AOSD 2002 Tutorial: Demeter

Aspect-Oriented Programming of Traversal-Related Concerns in Java

Demeter Research Group
Karl Lieberherr, Doug Orleans, Johan Ovlinger, John Sung, Mitchell Wand, Pengcheng Wu

Overview

- DJ introduction using AOP concepts
- How we got to AOP
- AspectJ and DJ

© 2002, by authors
Technology Evolution view of Demeter Research Group

Java  Object-Oriented Programming

Law of Demeter dilemma
Tangled traversal-related concerns

Java + DJ  Adaptive Programming (AP 1991)

Other tangled concerns: synchronization, data transfer, etc.

AspectJ  Aspect-Oriented Programming (AOP 1996)

Overview

• Aspect-oriented Programming in pure Java using the DJ library
  – AP concepts = DJ classes
  – Meaning of a traversal
  – Case study: semantic checking
  – Patterns for programming in DJ style
  – Generic programming with DJ
  – Adding traversals to AspectJ (with AspectJ team)
Modularization of crosscutting concerns

Scattering: count number of components to which color goes
 ordinary program aspect-oriented prog.

Concern 1: structure-shy functionality
Concern 2: object structure
Concern 3: synchronization
Problem addressed

• Encapsulation of traversal-related concerns
  – Those are concerns that involve a group of collaborating objects that are connected by has-a relationships or zero-argument methods.
  – Control scattering of traversal-related concerns and tangling with other concerns
• Construct useful programs without knowing exactly what data types are involved

How is the problem solved?

Use an AOP System

• Need to talk about traversal join points:
  – sets of points (called hooks or pointcuts) in the execution of a traversal where additional semantics will be specified
  – a means of specifying the semantics at those join points (called an enhancement or advice)

```java
public void before(Person host){ r++; }
public void before(Object host){ host.foo() }
```
An AOP System

- has 3 critical elements
  - what are the join points
  - means of identifying join points (pointcuts)
  - means of specifying semantics at join points (advice)

An AOP System (in more detail)

- what is the set of all join points
- means of identifying join points (pointcuts)
- means of specifying semantics at join points (advice)
- encapsulated units combining pointcuts and advice (aspects)
- method of attachment of units (weaves)

Pointcuts and advice are sometimes overlapping. Pointcuts might define an initial behavior plus a set of join points in that behavior.
DJ: set of all join points

- All points during the execution of a traversal algorithm on an object
- All nodes and edges of object graph slices for a fixed traversal algorithm
- An object graph slice is a subgraph of an object graph (selected by a traversal specification)

DJ pointcuts

- visitor method signatures
- the signatures define the points during the traversal when additional behavior needs to be executed.

```java
Object around(LetNode l, Subtraversal st) {
    Object[] bg = st.applyElement("bindings");
    Object[] bd = st.applyElement("body");
    return checkIfDeclared(bg, bd);
}
```
DJ advice

- each visitor method body is advice on the pointcut specified by the method signature.

```java
Object around(LetNode l, Subtraversal st) {
    Object[] bg = st.applyElement("bindings");
    Object[] bd = st.applyElement("body");
    return checkIfDeclared(bg, bd);
}
```

DJ aspects

- a visitor class is a package of pointcuts and advice, i.e., an aspect
- when you use a visitor in a traversal of an object graph (in traverse) then each pointcut is intersected with the traversal pointcut

```java
class MyVisitor extends Visitor {
    int r;
    public void before(Person host) { r++; }
    public void start() { r = 0; }
    public Object getReturnValue() { return new Integer(r); }
}
```
DJ weaves

- to attach an aspect you call traverse with an aspect (visitor).
- traverse expression attaches the aspect to an object graph slice (a subgraph of an object)

\[ \text{cg.traverse(this, WPS, v1)} \]

\text{what-to-traverse} \quad \text{where-to-go} \quad \text{what to do and when to do it}
Collaborating Classes

find all persons waiting at any bus stop on a bus route

BusRoute  
\( \xrightarrow{\text{busStops}} \)  
BusStopList

BusList  
\( \xrightarrow{0..*} \)  
BusStop

Bus  
\( \xrightarrow{0..*} \)  
Person

PersonList

OO solution: one method for each red class

DJ complete aspect example

class BusRoute {
int countPersons(ClassGraph cg) {
    String WPS="from BusRoute via BusStop to Person"
    Integer result = (Integer)
    cg.traverse(this, WPS, new Visitor(){
        int r;
        public void before(Person host){ r++; }
        public void start(){ r = 0; }
        public Object getReturnValue()
        {return new Integer (r);}
    }); return result.intValue();
}

continued

// Prepare the class graph
ClassGraph classGraph = new ClassGraph();
BusRoute aBusRoute = ...;
int r = aBusRoute.countPersons(classGraph);

How we got to Aspect-Oriented Programming

• Started simple: traversal-seeded programs:
  Law of Demeter dilemma.
• Talk generically about points in the execution
  of a traversal program.
• Generically means: parameterize program by
  an abstraction of its execution (class graph).
Why Traversal Strategies?

- Law of Demeter: a method should talk only to its friends:
  arguments and part objects (computed or stored)
  and newly created objects

- Dilemma:
  • Small method problem of OO (if followed) or
  • Unmaintainable code (if not followed)

- Traversal strategies are the solution to this dilemma

Law of Demeter Principle

- Each unit should only communicate with a limited set of other units: only units “closely” related to the current unit.
- “Each unit should only talk to its friends.” “Don’t talk to strangers.”
- Main Motivation: Control information overload. We can only keep a limited set of items in short-term memory.
Application to OO: generic application

- Unit = method
  - closely related =
    - methods of class of `this/self` and other argument classes
    - methods of immediate part classes (classes that are return types of methods of class of `this/self`)
Law of Demeter: talk only to your friends

**Object traversal**

**When:** select points in object traversal

**What:** actions to perform before and after node and edge visits

Scattering / Tangling

---

**Other aspect systems from our Research Group**

- COOL (1993) and RIDL (1994) by Crista Lopes
- Starting summer 1995: in collaboration with Xerox PARC (Gregor Kiczales)
- COOL and RIDL were the breeding ground for the first AspectJ weaver and the DemeterJ weaver.
thread synchronization

when: select method calls

what: actions to perform before and after method calls to maintain synchronization variables

Scattering / Tangling

COOL: Crista Lopes: 1993

COOL example

coordinator BoundedBuffer {
    self ex put, take;
    mutex {put, take}
    condition empty=true, full=false;

    put requires (!full) {
        on exit {empty=false;
            if (usedSlots==array.length)
                full=true; }
    }

    take requires (!empty) {
        on exit {full=false;
            if (usedSlots==0)
                empty=true; }
    }
}
**selective marshaling**

**when**: select method calls

**what**: actions to perform to transmit arguments and result

**Scattering / Tangling**

RIDL: Crista Lopes: 1994

RIDL example: Selective Marshaling

```
portal Library {
  BookCopy getBook(User u, String title) {
    return: copy (BookCopy bypass borrower,
                  Book bypass copies);
    u: copy (User bypass books; )
  }
  Book findBook(String title) {
    return: copy (Book bypass copies, ps; )
  }
}
```
Modularization of crosscutting concerns

Instead of writing this

During implementation separate issues are mixed together

During maintenance individual issues need to be factored out of the tangled code
Many functional concerns involve multiple objects that need to be traversed from a starting object. Such concerns are called traversal-related concerns.

- Subconcerns expressing
  - the traversal and
  - when to take action during the traversal and
  - what action to take

- Traversal related concerns deserve a traversal-specific aspect language which happens to fit into Java.
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 subcomputation 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>

Comparison of pointcuts: 
**AspectJ / DJ**

- target(a) && call(* t(*));
- **DJ: A a**
- this(a) && target(b) && call(* t1(*));
- **DJ: A a, B b**
AspectJ

- Xerox PARC: Gregor Kiczales et al.: lingua franca of AOP.
- One of the first versions: Crista Lopes (member of Demeter group): implementing both COOL and RIDL in a general purpose AO language (early AspectJ version).
- Model: join points, pointcuts, advice.

From Demeter to AspectJ

Demeter (for C++ or Java)              AspectJ

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

- **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
Java+DJ AspectJ

aspect name

class S{
    void collect(ClassGraph cg,
                 String constraint){
        String s = "from S" +
                   constraint + "to T";
        cg.traverse(this, s,
                     new Visitor(){ ... ;
                         public void before(T h){...}
                         public void start() { ... }});
    }
}

green: pointcut purple: advice

aspect Traversal_t { // t()
    // declare traversal t :
    // "from S" + c + "to T"; ...
    }
    aspect Collecting {
        pointcut start() : ...
        pointcut visitingT(T h):
            call(void t())
            && target(h);
        before():visitingT(){ ... }
        before():start(){ ... }
    }

Notice Difference

- In DJ
  - weave is done in Java code
  - cg.traverse(this,s,v);

- In AspectJ
  - weave is done on command line:
    - ajc Traversal_t.java Collecting.java ...

  but can use visitor in traversal specification!
**AspectJ**

- **aspect name**
- **pointcut** traced():
  - call(void D.update()) ||
  - call(void D.repaint());
- **before**: traced(){
  - println("Entering:" + thisJoinPoint);
}

**Java+DJ**

- **class** Source{
  - HashSet collect(ClassGraph cg, String constraint){
    - String s = "from Source" + constraint + "to Target";
    - return (HashSet) cg.traverse(this, s, new Visitor(){ ... ;
    - public void before (Target h) { ... }
    - public void start() { ... });
  }
}

**AspectJ:** refers to existing join points

**Demeter:** defines new join points in traversal

---

**AP**

- Late binding of data structures
- Programming without accidental data structure details yet handling all those details on demand without program change
- Reducing representational coupling
Concepts needed (DJ classes)

- **ClassGraph**
- **Strategy**
- **Visitor**
- **TraversalGraph**
- **ObjectGraph**
- **ObjectGraphSlice**

AOSD: not every concern fits into a component: **crosscutting**

<table>
<thead>
<tr>
<th>CR1</th>
<th>CM1</th>
<th>CM2</th>
<th>CM3</th>
<th>CM4</th>
<th>CM5</th>
<th>CM6</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CR2</th>
<th>x</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CR3</th>
<th></th>
<th>x</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CR4</th>
<th>x</th>
<th></th>
<th></th>
<th>x</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Goal: encapsulate “crosscutting” concerns
AP history

- Programming with partial data structures = propagation patterns
- Programming with participant graphs = high-level class graphs
- Programming with object slices
  - partial objects determined by traversal specifications

Collaborating Classes

find all persons waiting at any bus stop on a bus route

- BusRoute
  - buses
  - busStops
- Bus
  - 0..*
  - passengers
- BusList
  - 0..*
- Person
  - 0..*
- PersonList
  - waiting
- BusStop
  - 0..*
- BusStopList
  - OO solution: one method for each red class

4/7/02 Aspects and Demeter 45

4/7/02 Aspects and Demeter 46
find all persons waiting at any bus stop on a bus route

**Traversal Strategy**

from **BusRoute** via **BusStop** to **Person**

- BusRoute
  - buses
  - busStops
- BusList
  - 0..*
- Bus
  - passengers
- BusStopList
  - 0..*
- BusStop
  - waiting
- PersonList
  - 0..*
- Person

find all persons waiting at any bus stop on a bus route

**Robustness of Strategy**

from **BusRoute** via **BusStop** to **Person**

- BusRoute
  - villages
- BusList
  - 0..*
- Bus
  - passengers
- VillageList
- Village
- BusStopList
  - busStops
  - 0..*
- BusStop
  - waiting
- PersonList
  - 0..*
- Person
ObjectGraph: in UML notation

Route1: BusRoute
    busStops
        :BusStopList
            CentralSquare: BusStop
                waiting
                    :PersonList
                        Paul: Person
                        Seema: Person
    buses
        :BusList
            Bus15: Bus
                passengers
                    :PersonList
                        Joan: Person
                        Eric: Person

ObjectGraphSlice

Route1: BusRoute
    busStops
        :BusStopList
            CentralSquare: BusStop
                waiting
                    :PersonList
                        Paul: Person
                        Seema: Person
    buses
        :BusList
            Bus15: Bus
                passengers
                    :PersonList
                        Joan: Person
                        Eric: Person
Why crosscutting?

class BusRoute {
    int countPersons(ClassGraph cg) {
        String WPS="from BusRoute via BusStop to Person"
        Integer result = (Integer)
        cg.traverse(this, WPS, new Visitor()
        {
            int r;
            public void before(Person host){ r++; }
            public void start() { r = 0; }
            public Object getReturnValue()
            {
                return new Integer (r);
            }
        }); return result.intValue();
    }
}
Writing Adaptive Programs with Strategies (DJ=pure Java)

// Prepare the class graph
ClassGraph classGraph = new ClassGraph();
BusRoute aBusRoute = new BusRoute(...);
int r = aBusRoute.countPersons(classGraph);

Goal of DJ

- Focus on crosscutting traversal-related concerns: involve a group of collaborating objects which are manipulated to implement a behavior.
- Provide a Java library to cleanly encapsulate crosscutting traversal-related concerns whose ad hoc implementation would be scattered across many classes.
Solves Open Problem in AOP for Behavioral Aspects in Java

• The ClassGraph-Aspect-Freezing problem
  – When we have n aspects and the class graph changes, we potentially need to update all n aspects.
  – DJ allows us to loosely couple behavioral aspects to the class graph.
  – And this is all done in Java.

Writing Adaptive Programs with Strategies (DJ=pure Java)

String WPStrategy="from BusRoute via BusStop to Person"

class BusRoute {
    int countPersons(TraversalGraph WP) {
        Integer result = (Integer)
            WP.traverse(this, new Visitor{
                int r;
                public void before(Person host){ r++; }
                public void start() { r = 0; }
                public Object getReturnValue()
                    {return new Integer (r);}
            });
        return result.intValue();
    }
}
String WPStrategy="from BusRoute via BusStop to Person"

// Prepare the traversal for the current class graph
ClassGraph classGraph = new Y();
TraversalGraph WPTraversal = new TraversalGraph
    (WPStrategy, classGraph);

int r = aBusRoute.countPersons(WPTraversal);

class Utility {
    static
    int countPersons(ObjectGraphSlice countSlice){
        Integer result = (Integer)
            countSlice.traverse(new Visitor(){ int r;
                public void before(Person host){ r++; }
                public void start() { r = 0;}
                public Object getReturnValue()
                    {return new Integer (r);}
            });
        return result.intValue();
    }
}
find all persons waiting at any bus stop on a bus route

**TraversalGraph**

\[ \text{from BusRoute via BusStop to Person} \]

\begin{itemize}
  \item **BusRoute**
  \item **BusStopList**
  \item **BusList**
  \item **Bus**
  \item **PersonList**
  \item **Person**
\end{itemize}

Applications of Traversal Strategies

- Program Kinds in DJ
  - AdaptiveProgram\textsubscript{Traditional} (ClassGraph)
    - strategies are part of program: DemeterJ, Demeter/C++
  - AdaptiveProgram\textsubscript{Dynamic} (Strategies, ClassGraph)
    - strategies are a parameter. Even more adaptive.
  - AdaptiveProgram\textsubscript{TraditionalOptimized} (TraversalGraphs)
    - strategies are a parameter. Reuse traversal graphs.
  - AdaptiveProgram\textsubscript{DJ} (ObjectGraphSlices)
    - strategies are a parameter. Reuse traversal graph slices.
Example

• For data member access:
• C c = (C) Main.cg.fetch(this, “from A via B to C”);

Understanding the meaning of a strategy

• Classes involved: Strategy, ObjectGraph, ObjectGraphSlice, ClassGraph
• We want to define the meaning of a Strategy-object for an ObjectGraph-object as an ObjectGraphSlice-object (a subgraph of the ObjectGraph-object). Minimal attention necessary will be given to ClassGraph-object.
Searching for Reachable Objects

• Task: Given an object o1 of class c1 in an object graph, find all objects of type c2 that are reachable from o1.

• Assumptions: we know the class structure that describes the object graph, but we know nothing else about the object graph except the class of the current object.

Search using meta information

• we could visit the entire object but that
  – would be wasteful or
  – might lead to wrong results
### Classes and Objects: Basic Notations

- **Class c1 has a part e of type c2**
  - `c1` \(\rightarrow\) `c2`
- **Class c1 inherits from class c2**
  - `c1` \(\rightarrow\) `c2`
- **Object o1 is of class c1**
  - `o1` \(\rightarrow\) `c1`
- **Object o1 is of type c2 (i.e., its class is a subclass of c2)**
  - `o1` \(\rightarrow\) `c1` \(\rightarrow\) `c2`
- **Object o1 has a part e which is object o2**
  - `o1` \(\rightarrow\) `e` \(\rightarrow\) `o2`

### Finding the first step for the search

- **ObjectGraph-object**
- **Traversal Strategy:** from C1 to C2
- **Which arrows might lead to an object of type C2?**
  - `o1` \(\rightarrow\) `C1`
  - `o2`
  - `o3`
  - `o4`
Relations between Classes

- $e(C1,C2)$
- $C(C1,C2)$ (that is, $e(C1,C2)$ for some $e$)
- $C1 \leq C2$
- $\text{Class}(o1) = C1$
- Object $o1$ is of type $C2$: $\text{Class}(o1) \leq C2$

Relations between Objects

- $e(o1,o2)$
- $O(o1,o2)$ (that is, $e(o1,o2)$ for some $e$)
Operations on Relations

- $R.S = \{(x,z) \mid \text{exists } y \text{ s.t. } R(x,y) \text{ and } S(y,z)\}$
- $R^* = \text{reflexive, transitive closure of } R$

Possible edges in the object graph

- $e(o1, o2)$ implies $\text{class}(o1) \leq \text{class}(o2)$ in the class graph
- “up, over, and down”
- $O(o1, o2)$ implies $\text{class}(o1) \leq \text{class}(o2)$ in the class graph
Which edges to follow to C2?

From o1 of class C1, follow edge e iff there is some object graph O and some o2, o3 s.t.
• e(o1,o2),
• O*(o2,o3), and
• class(o3) <= C2

The existential quantifier “there is some object graph” represents our lack of knowledge about the rest of the object graph.

Example

from Basket to Orange

class graph

premature termination

object graph slice

4/7/02 Aspects and Demeter 71

4/7/02 Aspects and Demeter 72
Lack of Knowledge

- Objects of a given class may be very different.
- We want to go down edges without looking ahead!
- We don’t want to go down edges that are guaranteed to be unsuccessful (never reaching a target object).
Object graph conforms to class graph

- The object graph O must follow the rules of the class graph: the object graph cannot contain more information than the class graph allows.

For all edges \( e(o1,o2) \) in the object graph:
\( e(o1, o2) \) implies
\( \text{class}(o1) \ (\leq .e.\geq) \ \text{class}(o2) \) in the class graph

From dynamic to static characterization

From \( o1 \) of class \( c1 \), follow edge \( e \) iff there is some object graph \( O \) and some \( o2, o3 \) s.t.
- \( e(o1,o2) \),
- \( O^*(o2,o3) \), and
- \( \text{class}(o3) \leq c2 \)

From \( o1 \) of class \( c1 \), follow edge \( e \) iff there are classes \( c', c'' \) s.t.
- \( c1 \ (\leq .e.\geq) c' \)
- \( c' \ (\leq .C.\geq) c'' \) and
- \( c'' \leq c2 \)

Let \( c' \) be \( \text{class}(o2) \), \( c'' \) be \( \text{class}(o3) \)
from c1 to c2

Relational Formulation

From object o of class c1, to get to c2, follow edges in the set

\[ \text{POSS}(c1,c2,o) = \{ e \mid c1 \leq e \leq c2 \} \]

Can easily compute these sets for every c1, c2 via transitive-closure algorithms.

POSS = abbreviation for: following these edges it is still possible to reach a c2-object for some c1-object rooted at o.

Generalizations

- More complex strategies
- “from c1 via c2 to c3”
  - Use “waypoint navigation”; get to a c2 object, then search for a c3 object.
- More complex strategy graphs also doable in this framework
class dictionary
A = ["x" X] ["r" R].
B = ["b" B] D.
R = S.
S = ["t" T] C
C = D.
X = B.
T = R.
D = .

Example B

POSS(c1,c2,o)={e | c1 e.C* c2 }

class graph
A
R
S
T
C
D
X
B

A -> T
T -> D

Example B1

POSS(A,T,a1) = 1 edge
POSS(R,T,r1) = 1 edge
POSS(S,T,s1) = 0 edges

class graph
A
R
S
T
C
D
X
B

POSS(c1,c2,o)={e | c1 e.C* c2 }
Example B1

POSS(A,T,a1) = 1 edge
POSS(R,T,r1) = 1 edge
POSS(S,T,s1) = 0 edges

strategy
A -> T
T -> D

Example B2

POSS(A,T,a1) = 1 edge
POSS(R,T,r1) = 1 edge
POSS(S,T,s1) = 1 edge
POSS(T,D,t1) = 1 edge
POSS(R,D,r2) = 1 edge

strategy
A -> T
T -> D

object graph slice
POSS(c1,c2,o)={e | c1 e.C* c2 }

Aspects and Demeter
Example C

strategy SG:
\{A \rightarrow B
B \rightarrow C\}

Example C1

strategy SG:
\{A \rightarrow S
S \rightarrow C\}

Only node paths shown for space reasons
Example D

\[ S = \text{from BusRoute via Bus to Person} \]

Example D1

Only node paths shown for space reasons

\[ S = \text{from BusRoute via Bus to Person} \]
from c1  bypassing \{x_1, x_2, \ldots, x_n\}  to c2

Relational Formulation

From object o of class c1, to get to c2, follow edges in the set

$$\text{POSS}(c_1, c_2, o) = \{ e \mid c_1 \leq e, e \leq (\leq, C, \geq)^* \leq c_2 \}$$

POSS = abbreviation for: following these edges it is still possible to reach a c2-object for some c1-object rooted at o.

Delete x_1, x_2, \ldots, x_n and all edges incident with these nodes from the class graph (unless they are c1, c2).
Example D

\[ S = \text{from BusRoute bypassing Bus to Person} \]

```plaintext
BusRoute \rightarrow \text{busStops} \rightarrow \text{BusStopList}

BusList \rightarrow \text{NGasPowered} \rightarrow \text{BusStop}

\text{0..* passengers} \rightarrow \text{busStops} \rightarrow \text{busStops}

\text{waiting} \rightarrow \text{PersonList}

\text{DieselPowered} \rightarrow \text{Person} \rightarrow \text{0..*}
```

Example D

\[ S = \text{from BusRoute bypassing Bus to Person} \]

```plaintext
BusRoute \rightarrow \text{busStops} \rightarrow \text{BusStopList}

BusList \rightarrow \text{NGasPowered} \rightarrow \text{BusStop}

\text{0..* passengers} \rightarrow \text{busStops} \rightarrow \text{busStops}

\text{waiting} \rightarrow \text{PersonList}

\text{DieselPowered} \rightarrow \text{Person} \rightarrow \text{0..*}
```
Note

• Separation of concerns is also useful for defining programming language elements
  – separate subgraph selected from
  – how the subgraph is traversed (depth-first etc.)
• In earlier works: meaning of a traversal strategy for an object graph
  – was a traversal history
  – now it is a subgraph of the object graph. A traversal history can be defined ...

Programming Exercise

• Check whether all used entities are defined.
• Object structure, traversal (basically an introduction of methods), advice on traversal.
Crosscutting in Equation System

Class graph:
Find undefined things

definedThings = from System via -*,def,* to Thing
usedThings = from System via -*,body,* to Thing
Java Program: Adaptive Method with DJ

class System{
    String id = "from Thing to edu.neu.ccs.demeter.Ident";
    void repUndef(ClassGraph cg){
        checkDefined(cg, getDefThings(cg));
        HashSet getDefThings(ClassGraph cg){
            String definedThings =
                "from System via ->*,def,* to Thing";
            Visitor v = new Visitor(){
                HashSet return_val = new HashSet();
                void before(Thing v){
                    return_val.add(cg.fetch(v, id));
                }
                public Object getReturnValue(){return return_val;};
            return (HashSet)cg.traverse(this, definedThings, v);}
    }
}

void checkDefined(ClassGraph cg, final HashSet classHash){
    String usedThings =
        "from System via ->*,body,* to Thing";
    cg.traverse(this, usedThings, new Visitor(){
        void before(Thing v){ Ident vn = cg.fetch(v, vi);
            if (!classHash.contains(vn)){
                System.out.println("The object "+ vn + " is undefined.");
            }
    });}
}
Reengineer

• Reuse collection behavior twice.
• Much simpler to reengineer in this abstract form.
• Structure of program is not hardened yet by details of class graph.

class System{
    String id = "from Thing to edu.neu.ccs.demeter.Ident";
    HashSet collect(ClassGraph cg, String constraint){
        Visitor v = new Visitor(){
            HashSet return_val = new HashSet();
            void before(Thing v1){
                return_val.add(cg.fetch(v1, id));}
            public Object getReturnValue(){return return_val;}};
        cg.traverse(this, "from System"+constraint+"to Thing", v);
        return (HashSet)v.getReturnValue();}
    HashSet defined(ClassGraph cg){
        return (HashSet) this.collect(cg, "via ->*,def,*");}
    HashSet used(ClassGraph cg){
        return (HashSet) this.collect(cg, "via ->*,body,*");}}

4/7/02 Aspects and Demeter 97
M1: Equation System

EquationSystem

usedThings = from EquationSystem via Expression to Variable

EquationSystem

Equation_List

Equation

Variable

Expression

Expression_List

Compound

Fig. Eq3

Collect Things

System EquationSystem

Definition Equation

Body Expression

Thing Variable

definedThings = from System via -> * ,def,* to Thing

usedThings = from System via -> * ,body,* to Thing

= from System + constraint + to Thing
Example of Aspect-Oriented Programming

• Separating the following crosscutting concerns:
  – traversal-related concerns, for each one separate
    • Object Structure (detailed meta information)
    • Traversals via Objects (where to go)
    • Advice on Traversals (what to do)
  • Traversal-related concerns are common.

Loose Coupling

• Object Structure
  – does not have to know about traversals and advice on traversals

• Traversals
  – don’t have to know about advice on traversals

• Advice on Traversals
  – has to know minimally about object structure and traversals
Ad-hoc Implementation of three concerns

- Leads to lots of tangled code with numerous disadvantages
- The question is not how to eliminate the tangling but how to reduce it
- AOP is about tangling control for the implementation of crosscutting concerns
- Crosscutting will always lead to some tangling at code level

Name map

<table>
<thead>
<tr>
<th>Roles</th>
<th>CSI</th>
<th>CS2</th>
<th>MI</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>ClassG</td>
<td>Grammar</td>
<td>Equation-System</td>
</tr>
<tr>
<td>Body</td>
<td>Body</td>
<td>Body</td>
<td>Expression</td>
</tr>
<tr>
<td>Thing</td>
<td>ClassName</td>
<td>NonTerm</td>
<td>Variable</td>
</tr>
<tr>
<td>Definition</td>
<td>ClassDef</td>
<td>Production</td>
<td>Equation</td>
</tr>
</tbody>
</table>
High-level description

• It is useful to have a high-level description of the collaboration besides the Java source code. Useful documentation.
• Ultimately we are interested in the executable form of the collaboration (Java source code).

Collaboration with strategies

collaboration checkDef {
  role System {
    out repUndef(){(uses getDefThings, checkDefined)};
    getDefThings(){(uses definedThings)};
    checkDefined(){(uses usedThings)};
    in definedThings =
      from System bypassing Body to Thing;
    in usedThings =
      from System via Body to Thing; }
  role Body { }
  role Thing { }
  role Definition { }
}

4/7/02 Aspects and Demeter 105

4/7/02 Aspects and Demeter 106
Collaboration with strategies

collaboration COLLECT {
    role System {
        out HashSet collect(){};
        defined(){(uses collect, definedThings)};
        used(){(uses collect, usedThings)};
        in definedThings =
            from System bypassing Body to Thing;
        in usedThings =
            from System via Body to Thing; }
    role Body { }
    role Thing { }
    role Definition { }
}

Use of collaboration: Adapter

Need to provide the expected methods (in methods) and provide name map.

- name map:
  - System : EquationSystem
  - Definition : Equation
  - Body : Expression
  - Thing : Variable

- expected methods:
  - in definedThings // use default
  - in usedThings // use default
Ad-hoc Implementation of traversal-related concerns

- Leads to lots of tangled and scattered code with numerous disadvantages
- The question is not how to eliminate the tangling but how to reduce it
- AOP is about tangling control of the implementation of crosscutting concerns
- Crosscutting will always lead to some tangling at code level

Need more than localization of crosscutting concerns

- If we localize a crosscutting traversal-related concern in the standard way, we get a method that violates the Law of Demeter: it duplicates much class graph information
- In addition: Use traversal strategies to eliminate accidental noise in class graph
- Need AP to improve AOP
What is an aspect?

• An aspect is a modular unit of crosscutting implementation.
• A Java method is a modular unit.
• Can we make a Java method an aspect?
• Yes, we call such methods adaptive methods.
• They cut across many classes in an ad-hoc implementation.

Java Program: Adaptive Method

class System {
    String id = "from Thing to edu.neu.ccs.demeter.Ident";
    void repUndef(ClassGraph cg) {
        checkDefined(cg, getDefThings(cg));
    }
    HashSet getDefThings(ClassGraph cg) {
        String definedThings = "from System bypassing Body to Thing";
        Visitor v = new Visitor() {
            HashSet return_val = new HashSet();
            void before(Thing v1) {
                return_val.add(cg.fetch(v1, id));
            }
            public Object getReturnValue() { return return_val; }
        };
        cg.traverse(this, definedThings, v);
        return (HashSet)v.getReturnValue();
    }
}

4/7/02 Aspects and Demeter
Java Program: Adaptive Method

void checkDefined(ClassGraph cg, final HashSet.String usedThings = 
      "from System via Body to Thing";
    cg.traverse(this, usedThings, new Visitor()
      void before(Thing v){ Ident vn = cg.fetch(v, id);
      if (!classHash.contains(vn)){
          System.out.println("The object "+ vn + " is undefined.");
        }
    }});}

After applying name map

• For now we manually edit the Java program.
Java Program with less tangling

class EquationSystem{
    String id = "from Variable to edu.neu.ccs.demeter.Ident";
    void repUndef(ClassGraph cg)
    {
        checkDefined(cg, getDefThings(cg));
    }
    HashSet getDefThings(ClassGraph cg)
    {
        String definedThings =
            "from EquationSystem bypassing Expression to Variable";
        Visitor v = new Visitor()
        {
            HashSet return_val = new HashSet();
            void before(Variable v1)
            {
                return_val.add(cg.fetch(v1, id));
            }
            public Object getReturnValue()
            {
                return return_val;
            }
        }
        cg.traverse(this, definedThings, v);
        return (HashSet)v.getReturnValue();
    }
}

4/7/02 Aspects and Demeter

Java Program with less tangling

void checkDefined(ClassGraph cg, final HashSet classHash)
{
    String usedThings =
        "from EquationSystem via Expression to Variable";
    cg.traverse(this, usedThings, new Visitor()
    {
        void before(Variable v)
        {
            Ident vn = cg.fetch(v, id);
            if (!classHash.contains(vn))
            {
                System.out.println("The object "+ vn + " is undefined.");
            }
        }
    });
}

4/7/02 Aspects and Demeter 116
Tangling is localized
Scattering eliminated

- Instead of having code spread across several classes, it is localized in one class.
- Java program is describing the abstract pattern behind the computation of interest: checking whether used entities are defined.
- Tangling control through abstraction of patterns. We abstract away from structure.

definedThings = from ClassG bypassing Body to ClassName

CS1: UML class diagram ClassG
usedThings = from ClassG via Body to ClassName

**CS1: UML class diagram ClassG**

- **ClassG**
- **EParse**
- **BParse**
- **ClassDef**
- **Part**
- **ClassName**
- **Body**
- **Concrete**
- **Abstract**
- **Entry**
- **entries**

**M1: Equation System**

- **EquationSystem**
- **Equation**
  - **lhs**
  - **rhs**
- **Expression**
  - **Simple**
  - **Compound**
    - **args**
    - **op**
- **Expression_List**
- **Numerical**
- **Variable**
- **Ident**
- **Add**
- **Equation_List**
- **equations**

**Fig. Eq1**
definedThings = from EquationSystem Fig. Eq2 bypassing Expression to Variable

M1: Equation System

equations
Equation
Equation System
Equation_List
Expression
Expression List
Variable
Numerical
Add
Ident

usedThings = from EquationSystem via Expression to Variable

M1: Equation System

equations
Equation
Equation System
Equation_List
Expression
Expression List
Variable
Numerical
Add
Ident

4/7/02 Aspects and Demeter
Equation System Object

```
EquationSystem = <equations> List(Equation).
Equation = <lhs> Variable "=" <rhs> Expression ".".
Variable = Ident.
Expression : Simple | Compound.
Simple : Variable | Numerical.
Compound = "(" Op <args> List(Expression) ")".
Op : Add | Mul.
Add = "+".
Mul = "*".
Numerical = float.
```

Example:
```
x = 1.0.
y = (+ x 4.0).
z = (* x y).
```
M1: Equation System

EquationSystem = <equations> List(Equation).
Equation = <lhs> Variable "=" <rhs> Expression ".".
Variable = Ident.
Expression : Simple | Compound.
Simple : Variable | Numerical.
Compound = "(" Op <args> List(Expression) ")".
Op : Add | Mul.
Add = "+".
Mul = "*".
Numerical = float.

definedThings= from EquationSystem bypassing Expression to Variable

Example:
\[ x = 1.0 \]
\[ y = (+ x 4.0) \]
\[ z = (* x y) \]

CS1: UML class diagram Grammar

Fig. G1
definedThings = from Grammar bypassing Body to NonTerm

usedThings = from Grammar via Body to NonTerm
What DJ adds to AspectJ

- Pointcut definitions based on connectivity in class graph.
- Pointcut reduction (high-level point cut designator): free programmer from details of class graph.

Discussion with Gregor Kiczales at UBC

- Ontology of AOP
- Ontology is the study of what there is, an inventory of what exists. An ontological commitment is a commitment to an existence claim for certain entities.
basis of crosscutting

- a *join point model* (JPM) has 3 critical elements
  - what are the join points
  - means of identifying join points
  - means of specifying semantics at join points

### Range of AOP languages

<table>
<thead>
<tr>
<th>JPM</th>
<th>join points</th>
<th>means of identifying join points</th>
<th>specifying semantics at join points</th>
</tr>
</thead>
<tbody>
<tr>
<td>AspectJ dynamic JPM</td>
<td>points in execution call, get, set…</td>
<td>signatures w/ wildcards &amp; other properties of JPs</td>
<td>advice</td>
</tr>
<tr>
<td>static JPM</td>
<td>class members</td>
<td>signatures</td>
<td>add members</td>
</tr>
<tr>
<td>Demeter/C++</td>
<td>when traversal reaches object or edge</td>
<td>visitor method signatures</td>
<td>visitor method bodies</td>
</tr>
<tr>
<td>static JPM 1</td>
<td>class members</td>
<td>traversal spec. s class graph g</td>
<td>s + g (result = traversal implementation)</td>
</tr>
<tr>
<td>static JPM 2</td>
<td>class members</td>
<td>class names</td>
<td>add members</td>
</tr>
<tr>
<td>static JPM 3</td>
<td>class members</td>
<td>class graph</td>
<td>class graph with tokens=grammar (result = parsing and printing implementation)</td>
</tr>
</tbody>
</table>
Range of AOP languages

<table>
<thead>
<tr>
<th>JPM</th>
<th>join points</th>
<th>means of identifying join points</th>
<th>specifying semantics at join points</th>
</tr>
</thead>
<tbody>
<tr>
<td>AspectJ dynamic JPM</td>
<td>points in execution call, get, set...</td>
<td>signatures w/ wildcards &amp; other properties of JPs</td>
<td>advice</td>
</tr>
<tr>
<td>static JPM</td>
<td>class members</td>
<td>signatures</td>
<td>add members</td>
</tr>
<tr>
<td>DJ dynamic JPM 1</td>
<td>when traversal reaches object or edge (method traverse)</td>
<td>visitor method signatures</td>
<td>visitor method bodies</td>
</tr>
<tr>
<td>dynamic JPM 2</td>
<td>when traversal reaches object (methods fetch, gather, asList)</td>
<td>source and targets of traversal</td>
<td>method name (fetch, gather, asList)</td>
</tr>
<tr>
<td>static JPM 4</td>
<td>nodes in object graph o</td>
<td>trav. spec. s class graph g</td>
<td>s+g (result = traversal impl. = edges to traverse at nodes in object graph o)</td>
</tr>
</tbody>
</table>

Combining two join point models

- **Static JPM 1**: In Demeter we use traversal specifications and the class graph to define a traversal implementation (either static or dynamic)
- **Visitor JPM 1**: The result of static JPM 1 is used to define a second JPM:
  - The traversal implementation defines nodes and edge visits.
  - Visitor signatures define the nodes and edges where additional advice is needed: they are the means of identifying join points.
  - The means of specifying semantics at join points are the visitor bodies.
Pattern Language for Adaptive Programming (AP)

Structure-shy Traversal Pattern only

Structure-shy Traversal

• Intent
  – Succinctly represent a traversal to be performed on objects
  – Commit only to navigation strategy and specify navigation details later
Structure-shy Traversal

• Could also be called:
  – Adaptive Traversal
  – Structure-shy Walker

Structure-shy Traversal

• Motivation
  – Noise in objects for specific task
  – Focus on long-term intent
  – Don’t want to attach every method to a specific class explicitly. Leads to brittle programs.
  – Small methods problem (example: 80% of methods are two lines long or shorter)
Structure-shy Traversal

• Applicability
  – Need collaboration of at least two classes.
  – In the extreme case, each data member access is done through a succinct traversal specification.
  – Some subgraphs don’t have a succinct representation, for example a path in a complete graph. More generally: avoid well connected, dense graphs.

Structure-shy Traversal

• Solution
  – Use succinct subgraph specifications
  – Use succinct path set specifications
Structure-shy Traversal: Solution

• Traversal Strategy Graphs (Strategies)
  – *First stage*: A strategy is a graph with nodes and edges. Nodes are labeled with nodes of a class graph. Edges mean: all paths.
  – *Second stage*: label edges with constraints excluding edges and nodes in class graph

• Simplest useful strategy: One Edge. Possible syntax:
  • from Company to Salary or
  • {Company -> Salary}

• Line graph. Several edges in a line. Possible syntax:
  • From Company via Employee to Salary
  • {Company -> Employee Employee -> Salary}
Structure-shy Traversal: Solution

- Traversal Strategy Graphs (Strategies)
  - Star graph
    - From Company to {Personnel, Travel, Employee}

![Star Graph Diagram]

UML Class Diagram

- BusRoute
  - buses
  - busStops

- BusList
  - 0..*
  - passengers

- Bus
  - waiting

- PersonList
  - 0..*

- Person

- BusStopList
  - 0..*
  - waiting
find all persons waiting at any bus stop on a bus route

**Traversal Strategy**

*from BusRoute via BusStop to Person*

```
BusRoute --> BusStopList
  |_________busStops
<p>|
|---------------------|
| BusList             |
|                     |
| 0..*                |</p>
<table>
<thead>
<tr>
<th>buses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>0..*</td>
</tr>
<tr>
<td>passengers</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>BusStop</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>waiting</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>PersonList</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>0..*</td>
</tr>
<tr>
<td>Person</td>
</tr>
</tbody>
</table>
```

Robustness of **Strategy**

*from BusRoute via BusStop to Person*

```
BusRoute --> BusStopList
  |_________busStops
<p>|
|---------------------|
| BusList             |
|                     |
| 0..*                |</p>
<table>
<thead>
<tr>
<th>buses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>0..*</td>
</tr>
<tr>
<td>passengers</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
</tbody>
</table>
| VillageList         | 0..*
<p>| |
|                     |
| Village             |
|                     |</p>
<table>
<thead>
<tr>
<th>waiting</th>
</tr>
</thead>
<tbody>
<tr>
<td>PersonList</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>0..*</td>
</tr>
<tr>
<td>Person</td>
</tr>
</tbody>
</table>
```

4/7/02 Aspects and Demeter
Structure-shy Traversal

• Consequences
  – Programs become shorter and more powerful. A paradox. With less work we achieve more. Polya’s inventor paradox.
  – Program will adapt to many changes in class structure.

• Implementation
  – Many different models for succinct traversal specifications.
  – Best one: Strategies
  – Correct implementation of strategies is tricky. See paper by Lieberherr/Patt-Shamir strategies.ps in my FTP directory.
Structure-shy Traversal

- **Known Uses**
  - *Adaptive Programming*: Demeter/C++, DemeterJ, Dem/Perl, Dem/CLOS etc.
  - *Databases* (limited use): Structure-shy queries:
    - See Cole Harrison’s Master’s Thesis (Demeter Home Page)
  - XML: XPath
  - *Artificial Intelligence* (limited use): Minimal ontological commitment

More on DJ

- Including the connection to generic programming
ObjectGraphSlice

- The object graph slice starting with o1 is the slice built by following the edges POSS(Class(o1), t, o1) starting at o1 and continuing until every path terminates (at an object of type t or it terminates prematurely).

DJ

- An implementation of AP using only the DJ library (and the Java Collections Framework)
- All programs written in pure Java
- Intended as prototyping tool: makes heavy use of introspection in Java
- Integrates Generic Programming (a la C++ STL) and Adaptive programming
Integration of Generic and Adaptive Programming

- A traversal specification turns an object graph into a list.
- Can invoke generic algorithms on those lists. Examples: contains, containsAll, equals, isEmpty, contains, etc. add, remove, etc. throws operation not supported exception.
- What is gained: genericity not only with respect to data structure implementations but also with respect to class graph

Sample DJ code

```java
// Find the user with the specified uid
List libUsers =
    classGraph.asList(library,
    "from Library to User");
ListIterator li =
    libUsers.listIterator();
// iterate through libUsers
```
Methods provided by DJ

- On ClassGraph, ObjectGraph, TraversalGraph, ObjectGraphSlice: traverse, fetch, gather
- traverse is the important method; fetch and gather are special cases
- TraversalGraph
  - Object traverse(Object o, Visitor v)
  - Object traverse(Object o, Visitor[] v)

Traverse method: excellent support for Visitor Pattern

// class ClassGraph
Object traverse(Object o, Strategy s, Visitor v);
traverse navigates through Object o following traversal specification s and executing the before and after methods in visitor v
ClassGraph is computed using introspection
Fetch Method

• If you love the Law of Demeter, use fetch as your shovel for digging:
  – Part \( k1 = (K) \) classGraph.fetch(a,”from A to K”);
• The alternative is (digging by hand):
  – Part \( k1 = a.b().c().d().e().f().g().h().i().k(); \)
• DJ will tell you if there are multiple paths to the target (but currently only at run-time).

Gather Method

• Returns a list of objects.
• Object classGraph.gather(Object o, String s)
  – List \( ks = \) classGraph.gather(a,”from A to K”);
    returns a list of K-objects.
Using DJ

• `traverse(…)` returns the \( v[0] \) return value. Make sure the casting is done right, otherwise you get a run-time error. If “public Object getReturnValue()” returns an Integer and `traverse(…)` casts it to a Real: casting error at run-time.

• Make sure all entries of `Visitor[]` array are non-null.

Using multiple visitors

```java
// establish visitor communication
daV.set_cV(cV);
daV.set_sV(sV);
rV.set_aV(aV);

Float res = (Float) whereToGo.
    traverse(this,
        new Visitor[] {rV, sV, cV, aV});
```
### DJ binary construction operations

<table>
<thead>
<tr>
<th></th>
<th>cg</th>
<th>s</th>
<th>tg</th>
<th>o</th>
<th>og</th>
<th>ogs</th>
</tr>
</thead>
<tbody>
<tr>
<td>cg</td>
<td>*</td>
<td>tg, cg</td>
<td>*</td>
<td>og</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>s</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>ogs</td>
<td>*</td>
</tr>
<tr>
<td>tg</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>ogs</td>
<td>*</td>
</tr>
<tr>
<td>o</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>og</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>ogs</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Who has traverse, fetch, gather?**

(number of arguments of traverse)

<table>
<thead>
<tr>
<th></th>
<th>cg(3)</th>
<th>s</th>
<th>tg(2)</th>
<th>o</th>
<th>og(2)</th>
<th>ogs(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cg</td>
<td>*</td>
<td>tg, cg</td>
<td>*</td>
<td>og</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>s</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>ogs</td>
</tr>
<tr>
<td>tg</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>ogs</td>
</tr>
<tr>
<td>o</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>og</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>ogs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Methods returning an ObjectGraphSlice

• ClassGraph.slice(Object, Strategy)
• ObjectGraph.slice(Strategy)
• TraversalGraph.slice(Object)
• ObjectGraphSlice(ObjectGraph,Strategy)
• ObjectGraphSlice(ObjectGraph,TraversalGraph)

Blue: constructors

Traverse method arguments

• ClassGraph
  – Object, Strategy, Visitor
• TraversalGraph
  – Object, Visitor
• ObjectGraph
  – Strategy, Visitor
• ObjectGraphSlice
  – Visitor
Traverse method arguments.

Where is collection framework used?

- ClassGraph
  - gather(Object, Strategy) / asList(Object, Strategy)
- TraversalGraph
  - gather(Object) / asList(Object)
- ObjectGraph
  - gather(Strategy) / asList(Strategy)
- ObjectGraphSlice
  - gather() / asList()

Where is collection framework used?

- ObjectGraphSlice.asList()
  - a fixed-size List backed by the object graph slice. Is write-through: modifying list will modify object and modifying object will modify list. (Similar to Arrays.asList() in Java.)
- gather copies the pointers to the objects.
Interfaces

- Interface List
  - ListIterator listIterator()
  - Object set(int index, Object element)
- Interface ListIterator
  - void set(Object o): replaces the last element returned by next or previous with the specified element.
Why is asList useful?

• from Application to X: want to change all F-objects to D-objects.
• From Application to “predecessors” of D: put in a new object.
  – This is not structure-shy: all the predecessors of X also need to be mentioned.

Application = <es> List(E).
E = X D.
D = X F.
X : F | D.
F = .
D = List(X).
List(S) ~ {S}.

Sketch:
from Application to E:
aE.set_x(new D());
aE.get_d().set_x(new D());
from E to D: update list elements
from Application to X: set(new D())
Why is asList useful?

- From A to B: want to change a B-object that satisfies some property. Does not matter whether it is in a list.

\[ A = B \ C. \]

\[ C = \text{List}(B). \]

More on semantics of DJ with abstract classes

- What is the meaning of a visitor on an abstract class?
Traversals to Abstract Classes

- Class graph

```
• from A to E: b, c
• from A to B: b, c
• from A to C: b, c
• visitor:
  – void before (B){p("b");}
  – void before (C){p("c");}
```

Visitor Methods on Abstract Classes

```
• from A to E: b, c
• from A to B: b, c
• from A to C: b, c
• visitor:
  – void before (B){p("b");}
  – void before (C){p("c");}
```
Visitor Methods on Abstract Classes

- from A to E: b
- from A to B: b
- from A to C:
- visitor:
  - void before (B){p("b");}
  - void before (C){p("c");}

Visitor Rule

- When an object of class X is visited, all visitors of ancestor classes of X will be active.
- The before visitors in the downward order and the after visitors in the upward order.
**Traversals to Abstract Classes**

- From A to C

```plaintext
X

A

B

E

C

D

visitor:
void before (X host){p("x");}
void before (B host){p("b");}
void before (C host){p("c");}
```

**Guidelines**

If you use the combination of the following pairs and triples for multiple traversals, fetch or gather, introduce the following computation saving objects:

- \((cog,s,o)\rightarrow ogs\)
- \((cg,s)\rightarrow tg\)
- \((cg,o)\rightarrow og\)
- \((tg,o)\rightarrow ogs\)

Abbreviations:

- \(cg\): class graph
- \(s\): strategy
- \(tg\): traversal graph
- \(o\): object
- \(og\): object graph
- \(ogs\): object graph slice
- \(v\): visitor

In principle can express programs only with ClassGraph and Strategy and Visitor:

- \(cg\).traverse(o,s,v);
- \(cg\).fetch(o,s);
- \(cg\).gather(o,s);
- \(cg\).asList(o,s);
ClassGraph construction

- make a class graph from all classes in default package
  - ClassGraph()
    - include all fields and non-void no-argument methods. *Static members are not included.*
  - ClassGraph(boolean f, boolean m)
    - If f is true, include all fields; if m is true, include all non-void no-argument methods.
Dynamic features of DJ ClassGraph construction

• When a class is defined dynamically from a byte array (e.g., from network) `ClassGraph.addClass(Class cl)` has to be called explicitly. Class `cl` is returned by class loader.

• `ClassGraph()` constructor examines class file names in default package and uses them to create class graph.

• `ClassGraph.addPackage(String p)`
  – adds the classes of package `p` to the class graph. The package is searched for in the CLASSPATH. How can we control (f,m) options? Uses the same rule as used for the original class graph construction.

• Java has no reflection for packages. Motivates above solution.
Adding Nodes and Edges to ClassGraph

• addClass(Class cl)
  – add cl and all its members to the class graph, if it hasn’t already been added.

• addClass(Class cl, boolean aF, boolean aM)
  – add cl to the class graph. If aF, add all its non-static fields as construction edges. If aM, add all its non-static non-void methods with no arguments as derived construction edges.

• addConstructionEdge (Field f)
  – add f as a construction edge.

• addConstructionEdge (Method m)
  – add a no-args method as a construction edge.

addConstructionEdge may have in addition a String argument called source. For Impl.

• And also a Class argument called target. Also for Impl. Should not be public.
Add other repetition edges

- void ClassGraph.addRepetitionEdge(String source, String target)
  - add a repetition edge from source to target
- Questions
  - what about subclass and inheritance edges
  - what happens if class graph contains edges not in program. Error will occur.

Problem with Java

- What is coming is not about a problem of DJ but about a problem with Java: the lack of parameterized classes.
- The lack of parameterized classes forces the use of class Object which, as the mother of all classes, is too well connected.
- This leads to unnecessary traversals and traversal graphs that are too big.
Lack of parameterized classes in Java makes DJ harder to use

- Consider the traversal: from A to B
- Let’s assume that in the class graph between A and B there is a Java collection class. The intent is: A = List(B) which we cannot express in Java. Instead we have: A = Vector(Object). Object : A | B. Let’s assume we also have a class X=B.

Lack of parameterized classes in Java makes DJ harder to use

- We have: A = Vector(Object). Object : A | B | X. X = B.
- If the vector contains an X object it will be traversed!!!
No X-object is allowed to be in vector

Moral of the story

- If the Collection objects contain only the objects advertised in the nice class graph of the application the traversal done by DJ will be correct. But unnecessary traversals still happen.
- However, if the Collection objects contain additional objects (like an X-object) they will be traversed accidentally.
Moral of the story

- Java should have parameterized classes.
- Workaround: Use a JSR (Java Specification Request) 31 approach: use a schema notation with parameterization to express class graph and generate Java code from schema. For traversal computation, the schema will be used.

Size of traversal graph

- DJ might create big traversal graphs when collection classes are involved. DJ will plan for all possibilities even though only a small subset will be realized during execution.
- To reduce the size of the traversal graph, you need to use bypassing. In the example: from A bypassing \{A,X\} to B.
Adding Demeter Traversals to AspectJ

• Because the AspectJ join point model is a generalization of Demeter’s join point model, we can use the AspectJ pointcuts to express the traversal pointcuts.
• However, this would expose the traversal implementation.
• Extend AspectJ with two new pointcut primitives: traversal(t) and crossing(e).

Traversal(t) pointcut

• picks out all join points between the start and the end of the traversal that are needed for the traversal only, i.e. the node visits and the edge visits.
Note

- AspectJ is well-aligned with Demeter. Need only a small extension to add the Demeter traversals.
- Pointcuts become more verbose. Could also use traversals with visitors and extend them with AspectJ (use the args pointcut).

DAJ (temporary name for AspectJ with traversals)

- planned to be added to AspectJ
- provides an efficient implementation of traversals using the AP library
- we have also added traversals to Java but implementation is very slow (using reflection)
- AspectJ + traversals > Java + traversals because AspectJ > Java
Adding Demeter Traversals to AspectJ

• declare ClassGraph
• declare TraversalGraph
• declare Behavior

Self-Contained Example

• Java classes
• Traversal file
• Use traversal defined
Basket Example

Basket

Basket(Fruit f, Pencil p) { f = f; p = p; }
Fruit f;
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;
}

BasketMain.java

class Basket {
    Basket(Fruit _f, Pencil _p) { f = _f; p = _p; }
    Fruit f;
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 Weight{
    Weight(int _i) { i = _i; }
    int i;
    int get_i() { return i; }
}
BasketMain.java

class BasketMain {
    static public void main(String args[]) throws Exception {
        Basket b = new Basket(
            new Orange(
                new Color("orange"),
                new Weight(5)),
            new Pencil());
        int totalWeight = b.totalWeight();
        System.out.println("Total weight of basket = " +
            totalWeight);
    }
}

Traversals

• Count the total weight within the basket
• Traversal Strategy: “From Basket to Weight”
• Visitor: Add up all the values within Weight
Basket Example Traversal Graph

Two versions

• define visitor using
  – AspectJ pointcuts and advice
  – Java visitor class complete with initialization, finalization and return value processing
BasketTraversal.trv

// traversals for basket
aspect BasketTraversal {
    declare ClassGraph default;
    declare ClassGraph myClassGraph : default,
        "from Basket to * bypassing {->*,*,java.lang.String }");
    declare TraversalGraph t2 : myClassGraph, "from Basket to Weight";
}

public aspect BasketTraversal {
    // traversal t2 : {source: Basket -> target: Weight} with { }
    public void Basket.t2(){
        if (f != null) t2_crossing_f();
    }
    public void Basket.t2_crossing_f() { f.t2();}
    public void Fruit.t2() {
        if (w != null) t2_crossing_w();
    }
    public void Fruit.t2_crossing_w() { w.t2();}
    public void Weight.t2() {
    }
}

---

generated:
BasketTraversal.java

public aspect BasketTraversal {
    // traversal t2 : {source: Basket -> target: Weight} with { }
    public void Basket.t2(){
        if (f != null) t2_crossing_f();
    }
    public void Basket.t2_crossing_f() { f.t2();}
    public void Fruit.t2() {
        if (w != null) t2_crossing_w();
    }
    public void Fruit.t2_crossing_w() { w.t2();}
    public void Weight.t2() {
    }
}

for testing traversal code

pointcut pointcut_t2() : call(public void t2*());
before () : pointcut_t2() {
    System.out.println(thisJoinPoint);
}

} // BasketTraversal
BasketMainCount.java

// the aspect for counting the total weight of the basket
aspect BasketMainCount {
    static int returnVal;
    int Basket.totalWeight() {
        returnVal = 0;
        t2();
        return returnVal;
    }
    pointcut t2WeightPC(Weight weight) : call(* *t2*()) && target(weight);
    before(Weight weight) : t2WeightPC(weight) {
        returnVal += weight.get_i();
    }
}

BasketTraversal.trv

// traversals for basket
aspect BasketTraversal {
    declare ClassGraph default;
    declare ClassGraph myClassGraph : default,
        "from Basket to * bypassing -*,*,java.lang.String");
    declare TraversalGraph tg : myClassGraph, “from A to B”; // tg()
    declare Behavior void b1 : tg, Vis; //b1()
    declare Behavior Integer summing : myCg, “from A to B”, Vis; //summing()
    declare traversal Integer t2(myClassGraph, SumVis) :
        "from Basket to Weight”;
}
BasketMainCount.java

// the aspect for counting the total weight of the basket
aspect BasketMainCount {
  int Basket.totalWeight() {
    Integer retVal = summing();
    return retVal.intValue();
  }
  class SumVis {
    int retVal = 0;
    void before (Weight w) {retVal += weight.get_i();}
    Object returnVal() {return new Integer(retVal);}
  }
}

Related work

• www.ccs.neu.edu/research/demeter
• Or use google.com and search for “DJ DemeterJ”
• aosd.net
Specific DJ references

- Reflection 2001 paper
- Special issue on AOP of Comm. ACM Oct. 2001
- DAJ: Demeter AspectJ: John Sung’s Master’s Thesis 2002: provides fast implementation for AspectJ
Conclusions

- Traversal-related concerns are common
- Learning about traversal-related concerns is a good path to enter the world of AOP by using only Java
- Once the pointcuts and advice of traversals are mastered, it is easy to generalize to more pointcuts and the powerful world of AspectJ.