

Traversal Strategies

Specification and Efficient
Implementation
(Graph Theory of OOP/OOD)

Contents

- Traversal strategies (graph theory for OOD/OOP)
- Coordination aspect
- Applications of AP to UML/OCL

Introduction

- Define subgraphs succinctly
- Define path sets succinctly
- Applications
 - writing adaptive programs
 - marshaling objects
 - storing objects, persistent objects

Leads to smaller and more evolvable software

Summary of lecture

- Concept of traversal strategies
- How to write traversal strategies
- Detailed meaning of strategies
- Complexity of compilation: polynomial in the size of strategy and class graph
- How to implement traversals manually
- Define concepts of class and object graph.

Summary of lecture

- Previous approaches: less general and their compilation algorithms were of exponential complexity.
- Show need for parameters in traversal methods.

Overview

- Use structure in graphs to express subgraphs and path sets in those graphs.
- Gain: writing programs in terms of strategies yields shorter and more flexible programs.

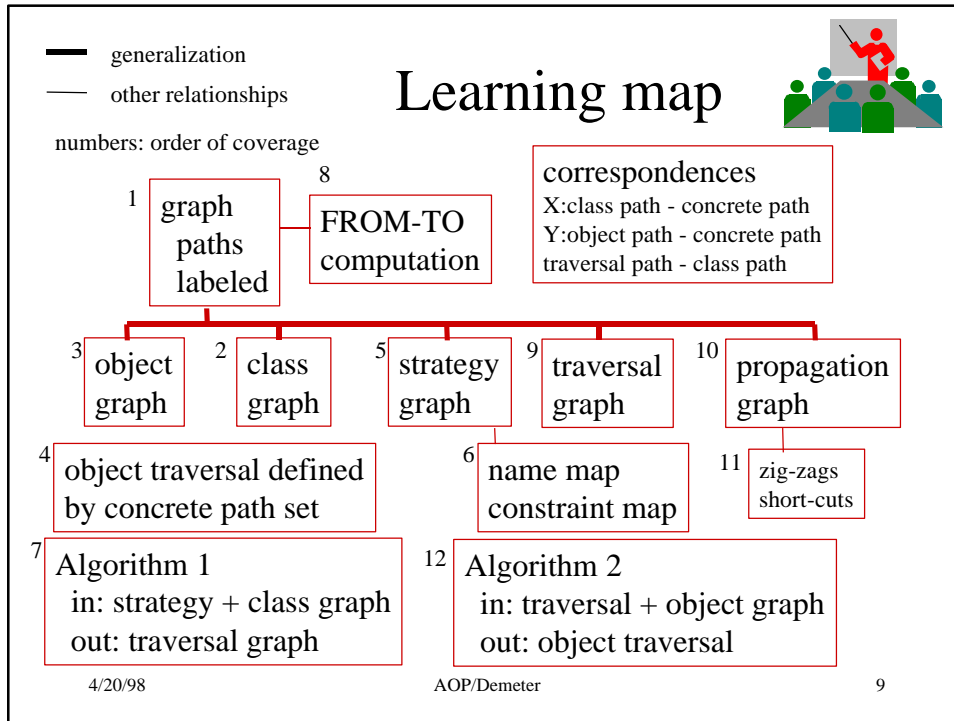
Connections

- strategy graphs, class graphs, object graphs
- simple class graphs, flat class graphs
- natural correspondence between paths in class graphs and object graphs
- compilation algorithm has some similarity with simulation of a non-deterministic automaton

Graphs used

- object graphs
- class graphs
- strategy graphs
- traversal graphs
- propagation graphs = folded traversal graphs

Therefore, introduce graph machinery for multiple use.



Graphs and paths

- Directed graph: (V, E) , V is a set of nodes, $E \subseteq V \times V$ is a set of edges.
- Directed labeled graph: (V, E, L) , V is a set of nodes, L is a set of labels, $E \subseteq V \times L \times V$ is a set of edges.
- If $e = (u, l, v)$, u is source of e , l is the label of e and v is the target of v .

Graphs and paths

- Given a directed labeled graph: (V, E, L) , a node-path is a sequence $p = \langle v_0 v_1 \dots v_n \rangle$ where $v_i \in V$ and $(v_{i-1}, l_i, v_i) \in E$ for some $l_i \in L$.
- A path is a sequence $\langle v_0 l_1 v_1 l_2 \dots l_n v_n \rangle$, where $\langle v_0 \dots v_n \rangle$ is a node-path and $(v_{i-1}, l_i, v_i) \in E$.

Graphs and paths

- In addition, we allow node-paths and paths of the form $\langle v_0 \rangle$ (called trivial).
- Unlabeled graphs have only node paths.
- First node of a path or node-path p is called the source of p , and the last node is called the target of p , denoted $Source(p)$ and $Target(p)$, respectively. Other nodes: interior.

Graphs and paths

- $P_G(u,v)$: set of all paths in G with source u and target v .
- concatenation of paths: If $p_1 = \langle v_0 \dots l_i v_i \rangle$ and $p_2 = \langle v_i l_{i+1} \dots v_n \rangle$ are paths, we define the concatenation $p_1 \cdot p_2 = \langle v_0 \dots l_i v_i l_{i+1} v_{i+1} \dots v_n \rangle$. Only one copy of meeting point v_i . Similar for node-paths.

Class graphs / object graphs

- Set of class names CC . Each class name is either abstract or concrete.
- Set of field names LL .
- Distinguished symbol \diamond not in LL for labeling subclass edges.

Class graphs / object graphs

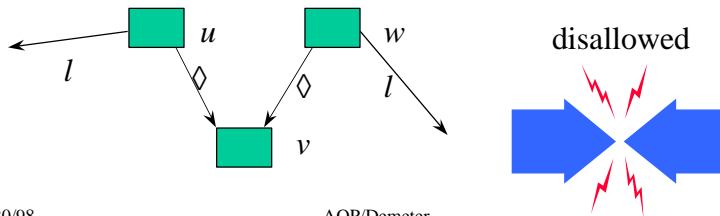
- Class graphs are graphs $G = (V, E, L)$ such that
 - V is a subset of CC (nodes are class names).
 - L is a subset of $LL \cup \{\diamond\}$.
 - For all v , field names of edges outgoing from v are distinct.
 - The set of edges labeled by \diamond is acyclic.

Class graphs / object graphs

- Edges labeled by field names are called reference edges, edges labeled by \diamond are called subclass edges.
- Reflexive notion of a superclass: Given a class graph $G = (V, E, L)$, $v \in V$ is a superclass of $u \in V$ if there is a possibly empty path of subclass edges from v to u .
- Ancestry of v : set of all superclasses of v .

Class graphs / object graphs

- Multiple inheritance conflicts are disallowed: we require that for all nodes v , if v has two superclasses u and w with outgoing edges labeled by the same field name, then either u is in the ancestry of w or w is in the ancestry of u .



4/20/98

AOP/Demeter

17

Class graphs / object graphs

- Induced references of a class: the set of all reference edges outgoing from its ancestry.
- Usual overriding rule: for each field name f used in edges outgoing from the ancestry of v , only the edge labeled f closest to v is in the induced references.
- Note: Induced references = direct references \cup inherited references

4/20/98

AOP/Demeter

18

Object graphs

- Model instantiations of class graphs.
- An object graph is a labeled directed graph $O = (V', E', L')$ where nodes are called objects and L' is a subset of LL .
- An object graph $O = (V', E', L')$ is an instance of a class graph $G = (V, E, L)$ under a function *Class* mapping objects to classes, if the following conditions hold:

4/20/98

AOP/Demeter

19

Object graphs

- For all objects $o \in V'$, $Class(o)$ is concrete.
- For each object $o \in V'$, the set of field names outgoing from o is exactly the set of field names of the induced references of $Class(o)$.
- For each edge $(o, f, o') \in E'$, $Class(o)$ has an induced reference edge (v, f, u) such that v is a superclass of $Class(o)$ and u is a superclass of $Class(o')$.

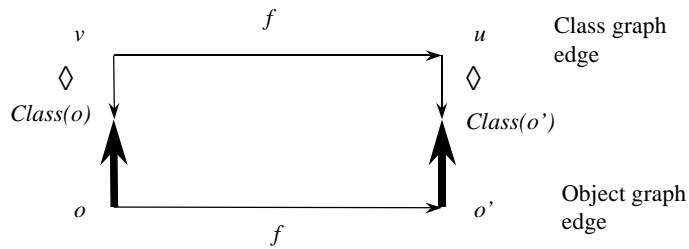
4/20/98

AOP/Demeter

20

Object graphs

For each edge $(o, f, o') \in E'$, $Class(o)$ has an induced reference edge (v, f, u) such that v is a superclass of $Class(o)$ and u is a superclass of $Class(o')$.



Natural correspondence

- So far, class graphs are very general: multiple inheritance is allowed, superclasses are not forced to be abstract.
- Without loss of generality: consider only a limited set of class graphs: simple class graphs. In simple class graphs: Easy mapping between class graph paths and object graph paths.

Simple class graphs

- We assume that class graphs are simple.
- A class graph $G = (V, E, L)$ is simple, if
 - for all edges $(u, f, v) \in E$, we have that $f = \hat{a}$ if and only if u is abstract, and
 - for all edges $(u, \hat{a}, v) \in E$, we have that v is concrete.

Simple class graphs

- for all edges $(u, f, v) \in E$, we have that $f = \hat{a}$ if and only if u is abstract, and
- Says that (1) all edges outgoing from abstract classes are subclass edges and (2) all edges outgoing from concrete classes are reference edges.
- (1) flatness
- (2) concrete classes have no subclasses

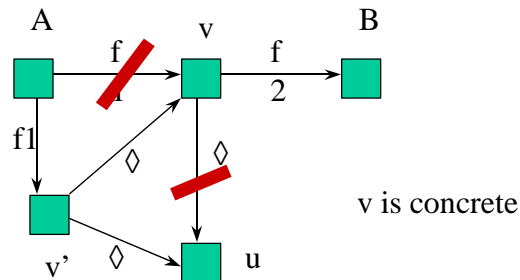
Simple class graphs

- for all edges $(u, \hat{a}, v) \in E$, we have that v is concrete.
- All subclass edges are incoming into concrete classes.

Simple class graphs

- Three situations are forbidden:
 - concrete superclasses
 - introduce abstract classes and rearrange
 - common parts
 - flatten
 - inheritance chains
 - for abstract v : find all concrete classes u reachable from v using subclass edges only. Add a subclass edge $(v \langle \rangle u)$ if one does not exist. Delete subclass edges leading to abstract classes.

Simple class graphs



4/20/98

AOP/Demeter

27

Proposition: nothing lost with simple class graphs

- Let $G = (V, E, L)$ be an arbitrary class graph. Then there exists a class graph $Simplify(G) = (V', E', L)$ such that an object graph O is an instance of G if and only if O is an instance of $Simplify(G)$. Moreover, $|V'| = O(|V|)$, $|E'| = O(|E|^2)$.
- Introduces multiple inheritance.

4/20/98

AOP/Demeter

28

Natural correspondence X

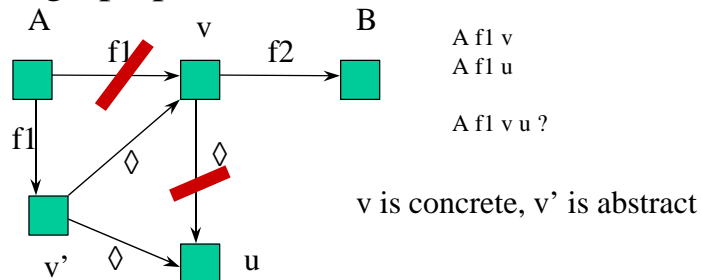
- A concrete path is an alternating sequence of concrete class names and field names (excluding \diamond).
- Natural correspondence X : map path p in class graph to concrete path $X(p)$ by omitting abstract classes and subclass edges.

Natural correspondence Y

- Map an object-graph path p' to a concrete path $Y(p')$ by taking the sequence of class names (under the *Class* function) and field names.
- Motivation: If p is a path in class graph G , then there is some object graph O which is an instance of G , and a path p' in O , such that $X(p) = Y(p')$.

Natural correspondence

- Simple class graphs have a simple relationship between class graph paths and object graph paths.



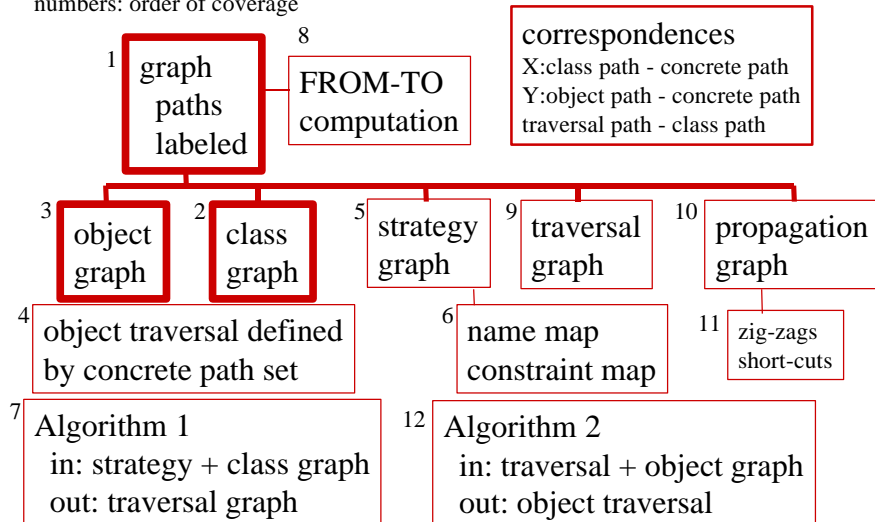
Learning map



— generalization

— other relationships

numbers: order of coverage



Definition of traversals

- For a set of sequences R :
 - $head(R) = \{x \mid \text{there exists } p: x.p \in R\}$
 - $tail(R,x) = \{p \mid x.p \hat{I} R\}$
- Assume a total order $<$ on set of field names LL .

Definition of traversals

- *traversing O from o guided by R produces H* = traversing the object graph O starting with o , and guided by a concrete path set R , yields traversal history H .
- Most of the time we can think of R as being defined by a subgraph of the original class graph.
- When object is visited, a method may be invoked.

Definition of traversals

If for i from 1 to n
traversing O from o_i guided by
 $\text{tail}(\text{tail}(R, \text{Class}(o)), l_i)$ **produces H_i** then
traversing O from o guided by R produces
 $H_1. \dots .H_n$ provided $\text{head}(\text{tail}(R, \text{Class}(o))) =$
 $\{l_i \mid i = 1 \dots n\}$, $(o, l_i, o_i) \hat{I} O$ and $l_j < l_k$ for
 $0 < j < k < n+1$.

Definition of traversals

- If $\text{tail}(R, \text{Class}(o)) = \text{empty set}$, then **traversing O from o guided by R produces ϵ** , where ϵ denotes the empty history.

Remarks about traversals

- If object graph is cyclic, traversal is not well defined.
- Traversals are opportunistic: As long as there is a possibility for success (i.e., getting to the target), the branch is taken.
- Traversals do not look ahead. Visitors must delay action appropriately.

Strategies: traversal specification

- Strategies select class-graph paths and then derive concrete paths by applying the natural correspondence.
- Traversals are defined in terms of sets of concrete paths.
- A strategy selects class graph paths by specifying a high-level topology which spans all selected paths.

Strategies

- A strategy SS is a triple $SS = (S, s, t)$, where $S = (C, D)$ is a directed unlabeled graph called the strategy graph, where C is the set of strategy-graph nodes and D is the set of strategy-graph edges, and $s, t \in C$ are the source and target of SS , respectively.

Strategies, name map

- Let $SS = (C, D)$ be a strategy graph and let $G = (V, E, L)$ be a class graph. A name map for SS and G is a function $N: C \rightarrow V$. If p is a sequence of strategy graph nodes, then $N(p)$ is the sequence of class nodes obtained by applying N to each element of p .
- Intuitively, strategy graph edge “ a to b ” represents paths from $N(a)$ to $N(b)$.

Strategies, expansion

- Given a sequence p , a sequence p' is an expansion of p if p' can be obtained by inserting elements between the elements of p .

Strategies, paths

- Let $SS = (S, s, t)$ be a strategy, let $G = (V, E, L)$ be a class graph, and let N be a name map for SS and G . The set of concrete paths $SS[G, N]$ is $\{X(p') \mid p' \hat{I} P_G(N(s), N(t)) \text{ and there exists } p \hat{I} P_S(s, t) \text{ such that } p' \text{ is an expansion of } N(p)\}$.

Strategies, constraint map

- Need negative constraints
- Given a class graph $G = (V, E, L)$, an element predicate EP for G is a predicate over $V \hat{E} E$. Given a strategy SS , a function B mapping each edge of SS to an element predicate is called a constraint map for SS and G .

Strategies, constraint map

- Let S be a strategy graph, let G be a class graph, let N be a name map and let B be a constraint map for S and G . Given a strategy-graph path $p = \langle a_0 a_1 \dots a_n \rangle$, we say that a class graph path p' is a satisfying expansion of p with respect to B under N if there exist paths p_1, \dots, p_n such that $p' = p_1 \cdot p_2 \dots p_n$ and:

Strategies, constraint map

- For all $0 < i < n+1$, $Source(p_i) = N(a_{i-1})$ and $Target(p_i) = N(a_i)$.
- For all $0 < i < n+1$, the interior elements of p_i satisfy the element predicate $B(a_{i-1}, a_i)$.

Strategies

- Many ways to decompose a path.
- Element constraints never apply to the ends of the subpaths.
- from A bypassing $\{A, B\}$ to B

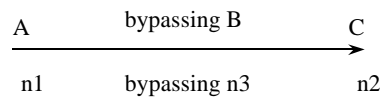
Strategies, paths

- Let $SS = (S, s, t)$ be a strategy, let $G = (V, E, L)$ be a class graph, and let N be a name map for SS and G and let B be a constraint map for S and G . The set of concrete paths $SS[G, N, B]$ is $\{X(p') \mid p' \hat{I} P_G(N(s), N(t)) \text{ and there exists } p \hat{I} P_S(s, t) \text{ such that } p' \text{ is an expansion of } N(p) \text{ w.r.t. } B\}$.

Strategies

- $SS[G, N] = SS[G, N, B_{TRUE}]$ for the constraint map B_{TRUE} which maps all strategy graph edges to the trivial element predicate that is always TRUE.
- Encapsulated strategies: want a clean separation between strategy graphs and class graphs.

Strategies



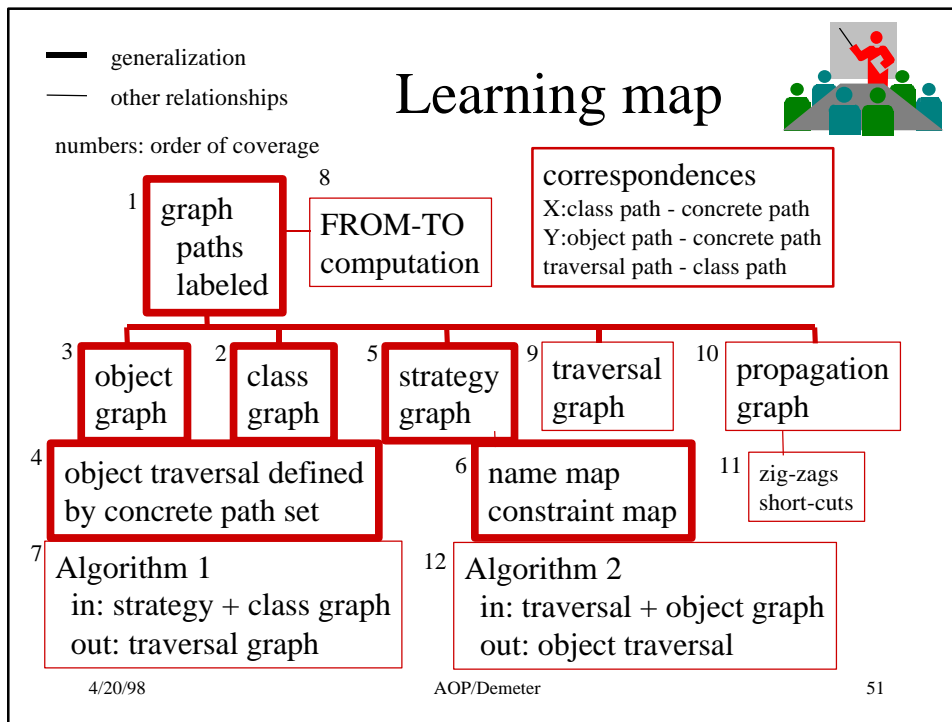
Name map:

n1	A
n2	C
n3	B
A	Company
B	Retirement
C	Salary

In Demeter/Java:
name map is identity

Strategies

- Are used in adaptive programs.
- Adaptive programs are expressed in terms of class-valued and relation-valued variables. Class graph not known when program is written.
- Wildcard notation in predicate specification: $\text{bypassing} (*, f, *)$.



Compilation of strategies

- Compilation problem
 - INPUT: A strategy $SS=(S,s,t)$, a simple class graph $G=(V,E,L)$, a name map N for S and G , and a constraint map B for S and G .
 - OUTPUT: A set of methods such that for any object graph O , invoking the traversal method at an object $o \in O$ yields a traversal history H satisfying *traversing O from o guided by $SS[G,N,B]$ produces H .*

What we tried.

- Path set is represented by subgraph of class graph, called propagation graph. Propagation graph is translated into a set of methods. Works in many cases. Two important cases which do not work:
 - short-cuts
 - zig-zags

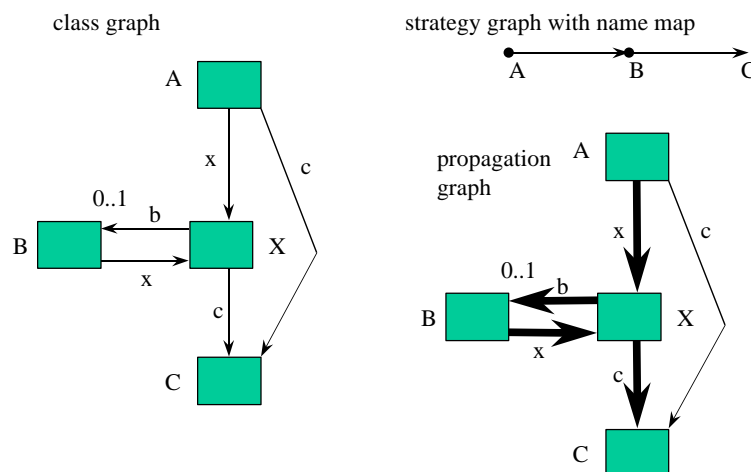
4/20/98

AOP/Demeter

53

Short-cut

```
strategy:  
{A -> B  
 B -> C}
```



4/20/98

AOP/Demeter

54

1+1=3 Short-cut

```
strategy:
{A -> B
 B -> C}
```

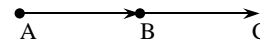
Incorrect traversal code:

```
class A {void t(){x.t();}}
class X {void t(){if (b!=null)b.t();c.t();}}
class B {void t(){x.t();}}
class C {void t(){}}
```

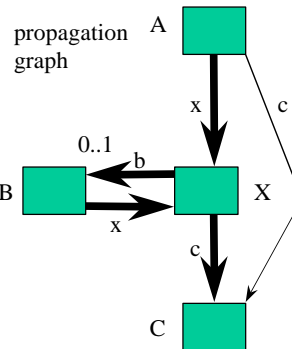
Correct traversal code:

```
class A {void t(){x.t();}}
class X {void t(){if (b!=null)b.t2();
              void t2(){if (b!=null)b.t2();c.t2();}}
}
class B {void t2(){x.t2();}}
class C {void t2(){}}
```

strategy graph with name map



propagation graph



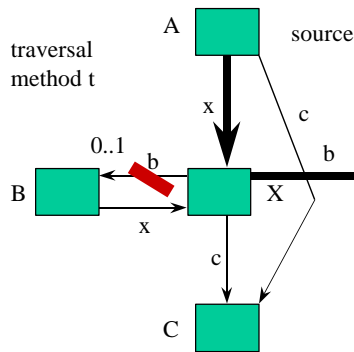
abstract representation
of traversal code



Short-cut

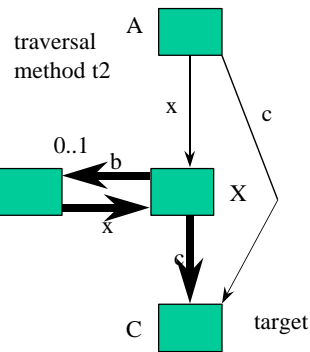
```
strategy:
{A -> B
 B -> C}
```

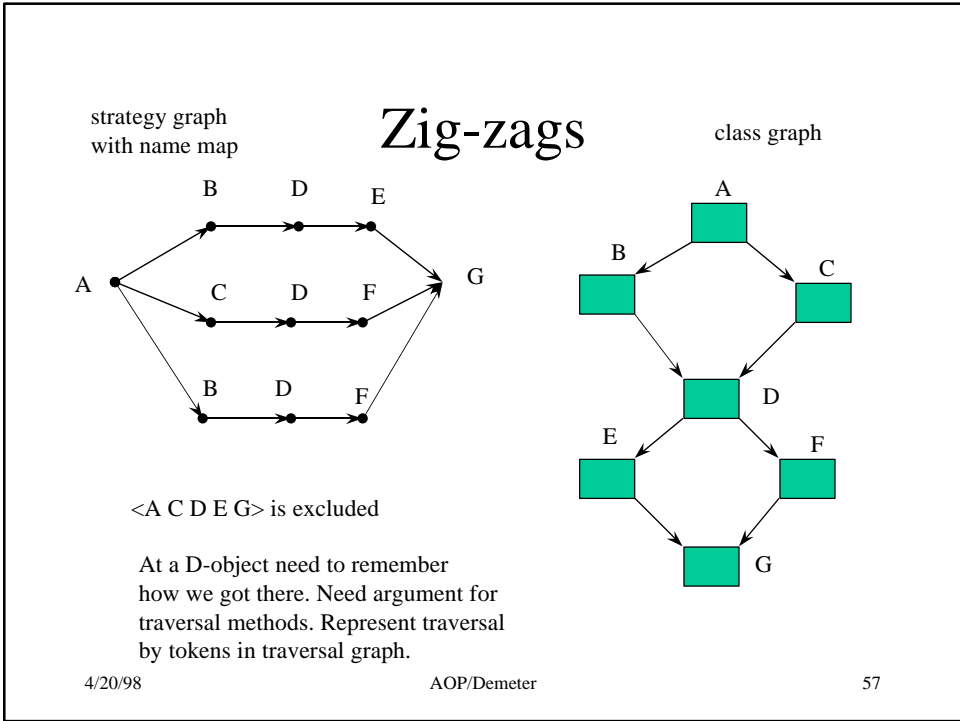
class graph



thick edges with incident nodes: traversal graph

class graph



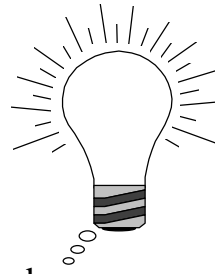


Compilation of strategies

- Two parts
 - construct graph which expresses the traversal $SS[G, N, B]$ in a more convenient way: traversal graph $TG(SS, G, N, B)$. Represents allowed traversals as a “big” graph.
 - code for traversal methods which uses $TG(SS, G, N, B)$.

4/20/98
AOP/Demeter
58

Compilation of strategies



- Idea of traversal graph:
 - Paths defined by `from A to B` can be represented by a subgraph of the class graph. Compute all edges reachable from A and from which B can be reached. Edges in intersection form graph which represents traversal.
 - Generalize to any strategies: Need to use *big* graph but above `from A to B` approach will work.

4/20/98

AOP/Demeter

59

Compilation of strategies

- Idea of traversal graph:
 - traversal graph is “big brother” of propagation graph
 - is used to control traversal
 - FROM-TO computation: Find subgraph consisting of all paths from A to B in a directed graph: Fundamental algorithm for traversals
 - Traversal graph computation is FROM-TO computation.

4/20/98

AOP/Demeter

60

Strategy behind Strategy

- Instead of developing a specialized algorithm to solve a specific problem, modify the data until a standard algorithm can do the work. May have implications on efficiency.
- In our case: use FROM-TO computation.

FROM-TO computation

- Problem: Find subgraph consisting of all paths from A to B in a directed graph.
 - Forward depth-first traversal from A
 - colored in red
 - Backward depth-first traversal from B
 - colored in blue
 - Select nodes and edges which are colored in both red and blue.

Variations

- reverse edges during first traversal in copy of graph
- could also use breadth-first traversal

Depth-first traversal

- Topological sorting
- Cycle checking
- Compute strongly-connected components
- Shortest paths

Traversal graph computation

Algorithm 1

- Let the strategy graph $S = (C, D)$ and let the strategy graph edges be $D = \{e_1, e_2, \dots, e_k\}$.
- 1. Create a graph $G' = (V', E')$ by taking k copies of G , one for each strategy graph edge. Denote the i th copy as $G^i = (V^i, E^i)$.
- The nodes in V^i and edges in E^i are denoted with superscript i , as in v^i, e^i , etc.

Why k copies?

- Mimics using k distinct traversal method names.
- Run-time traversals need enough state information.

Traversal graph computation

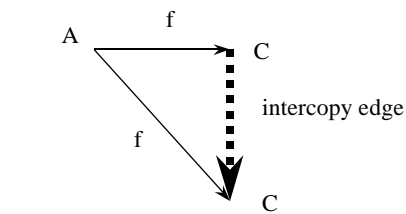
- Each class-graph node v corresponds to k nodes in V' , denoted v^1, \dots, v^k .
- Extend *Class* mapping to apply to nodes of G' by setting $Class(v^i) = v$, where $v^i \hat{I} V$ and $v \hat{I} V$.

Preview of step 2

- Link the copied class graphs through temporary use of intercopy edges.
- Each strategy graph node is responsible for additional edges in the traversal graph.
- If strategy graph node has one incoming and one outgoing edge, one edge is added.

Preview of step 2

- Redirection of edges from one copy to the next:



f may be \diamond

Traversal graph computation

- 2.a For each strategy-graph node $a \hat{I} C$: Let $I = \{ei_1, \dots, ei_n\}$ be the strategy-graph edges incoming into a , and let $O = \{eo_1, \dots, eo_m\}$ be the set of strategy graph edges outgoing from a . Let $N(a) = v \hat{I} V$. Add n times m edges v^j to v^l for $j=1, \dots, n$ and $l = 1, \dots, m$. Call these edges intercopy edges.

Traversal graph computation

- 2.b For each node $v^i \hat{I} G'$ with an outgoing intercopy edge: Add edges (u^i, f, v^j) for all u^i such that $(u^i, f, v^i) \hat{I} E^i$, and for all v^j which are reachable from v^i through intercopy edges only.
- 2.c Remove all intercopy edges added in step 2.a.

Preview of step 3

- Delete edges and nodes which we do not want to traverse.

Traversal graph computation

- 3. For each strategy-graph edge $e_i = \text{from } a \text{ to } b$: Let $N(a) = u$ and $N(b) = v$. Remove from the subgraph G^i all elements which do not satisfy the predicate $B(e_i)$, with the exception of u^i and v^i .
 - $V^i = \{v^i, u^i\} \hat{\cup} \{w^i \mid B(e_i)(w) = \text{TRUE}\}$, and
 - $E^i = \{(w^i, l, y^i) \mid B(e_i)(w, l, y) = B(e_i)(w) = B(e_i)(y) = \text{TRUE}\}$.

Preview of step 4

- Get ready for the FROM-TO computation in the traversal graph: need a single source and target.

Traversal graph computation

- 4.a Add a node s^* and an edge $(s^*, N(s)^i)$ for each edge e_i outgoing from s in the strategy graph, where s is the source of the strategy.
- 4.b Add a node t^* and an edge $(N(t)^i, t^*)$ for each edge e_i incoming into t in the strategy graph, where t is the target of the strategy.

Traversal graph computation

- 4.c Mark all nodes and edges in G' which are both reachable from s^* and from which t^* is reachable, and remove unmarked nodes and edges from G' . Call the resulting graph $G''=(V'',E'')$.
- The above is an application of the FROM-TO computation.

Traversal graph computation

- 5. Return the following objects:
 - The graph obtained from G'' after removing s^* and t^* and all their incident edges. This is the traversal graph $TG(SS, G, N, B)$.
 - The set of all nodes v such that (s^*, v) is an edge in G'' . This is the start set, denoted T_s .
 - The set of all nodes v such that (v, t^*) is an edge in G'' . This is the finish set, denoted T_f .

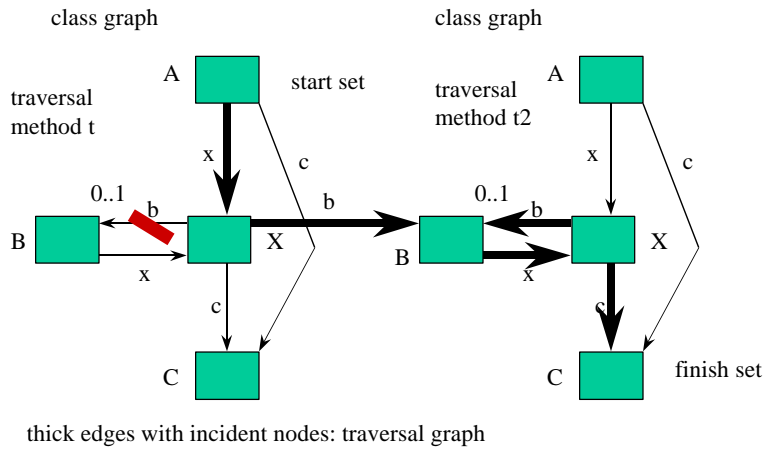
Traversal graph properties

- If p is a path in the traversal graph, then under the extended *Class* mapping, p is a path in the class graph. (Roughly: traversal graph paths are class graph paths.)

abstract representation
of traversal code

Short-cut

```
strategy:
{A -> B
 B -> C}
```



4/20/98

AOP/Demeter

79

Can now think in terms of a graph and need no longer path sets. But graph may be bigger.

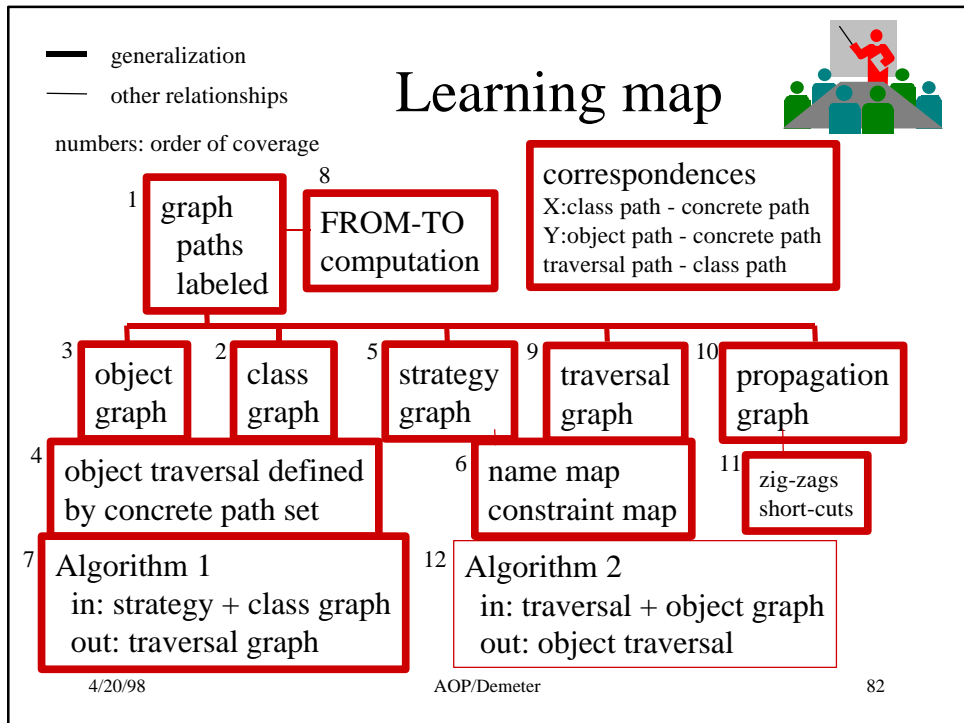
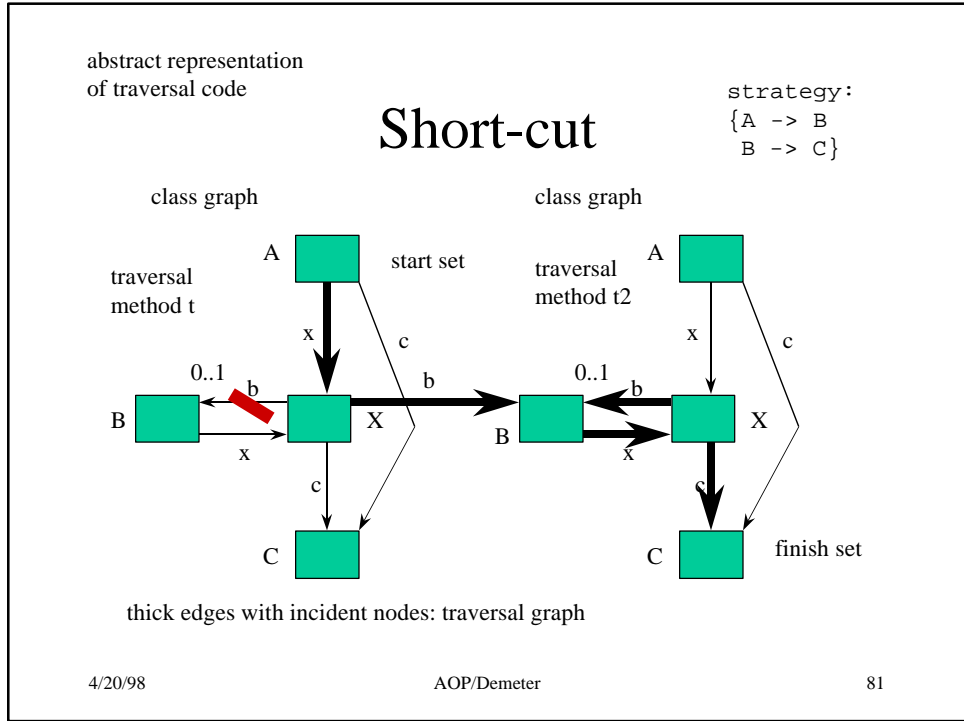
Traversal graph properties

- Let SS be a strategy, G a class graph, N a name map, and let B be a constraint map. Let $TG = TG(SS, G, N, B)$ be the traversal graph and let T_s be the start set and T_f the finish set generated by algorithm 1. Then $X(Class(P_{TG}(T_s, T_f))) = SS[G, N, B]$. (Roughly: Paths from start to finish in traversal graph are the paths selected by strategy.)

4/20/98

AOP/Demeter

80



Traversal methods algorithm

Algorithm 2

- Idea is to traverse an object graph while using the traversal graph as a road map.
- Maintain set of “tokens” placed on the traversal graph.
- May have several tokens: path leading to an object may be (under Y) a prefix of several distinct paths in $SS[G,N,B]$.

4/20/98

AOP/Demeter

83

Traversal method algorithm

- Traversal method $Traverse(T)$, where T a set of tokens, i.e., a set of nodes in the traversal graph.
- When $Traverse(T)$ invokes visit at an object, that object is added to traversal history.



4/20/98

AOP/Demeter

84

Traversal method algorithm

- $Traversal(T)$ is generic: same method for all classes.
- $Traversal(T)$ is initially called with the start set T_s computed by algorithm 1.

Traversal methods algorithm

- $Traverse(T)$, guided by traversal graph TG .
 - 1. define a set of traversal graph nodes T' by $T' = \{v \mid Class(v) = Class(this) \text{ and there exists } u \in T \text{ such that } u=v \text{ or } (u, \hat{a}, v) \text{ is an edge in } TG\}$.
 - 2. If T' is empty, return.
 - 3. Call `this.visit()`.



Traversal methods algorithm

- 4. Let Q be the set of labels which appear both on edges outgoing from a node in $T' \hat{I} TG$ and on edges outgoing from $this$ in the object graph. For each field name $l \hat{I} Q$, let

$$T_l = \{v / (u, l, v) \hat{I} TG \text{ for some } u \hat{I} T'\}.$$
- 5. Call $this.l.Traverse(T_l)$ for all $l \hat{I} Q$, ordered by “<“, the field ordering.

4/20/98

AOP/Demeter

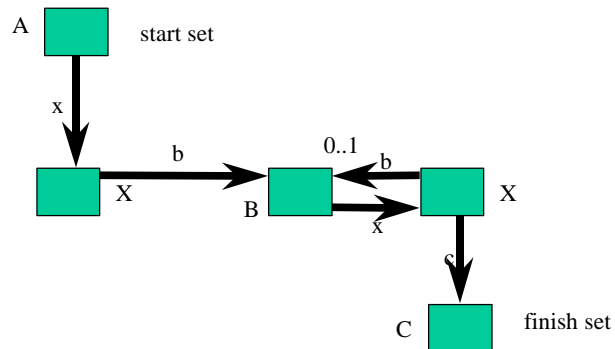
87

Object graph

```
A(
  <x> X(
    <b> B(
      <x> X(
        <c> C()))
    <c> C()))
```

Short-cut

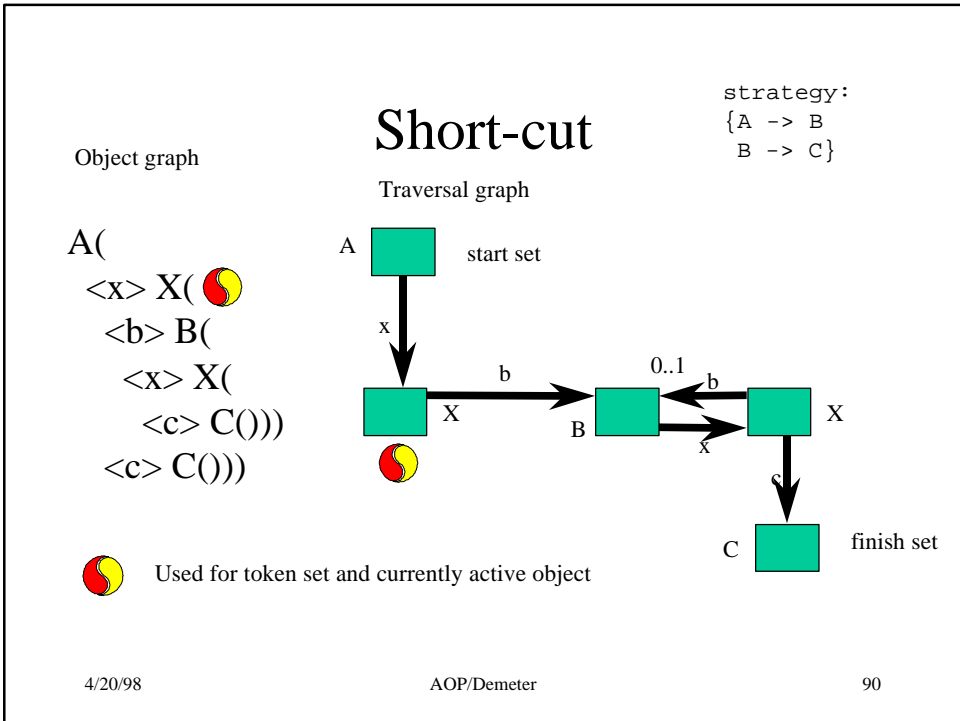
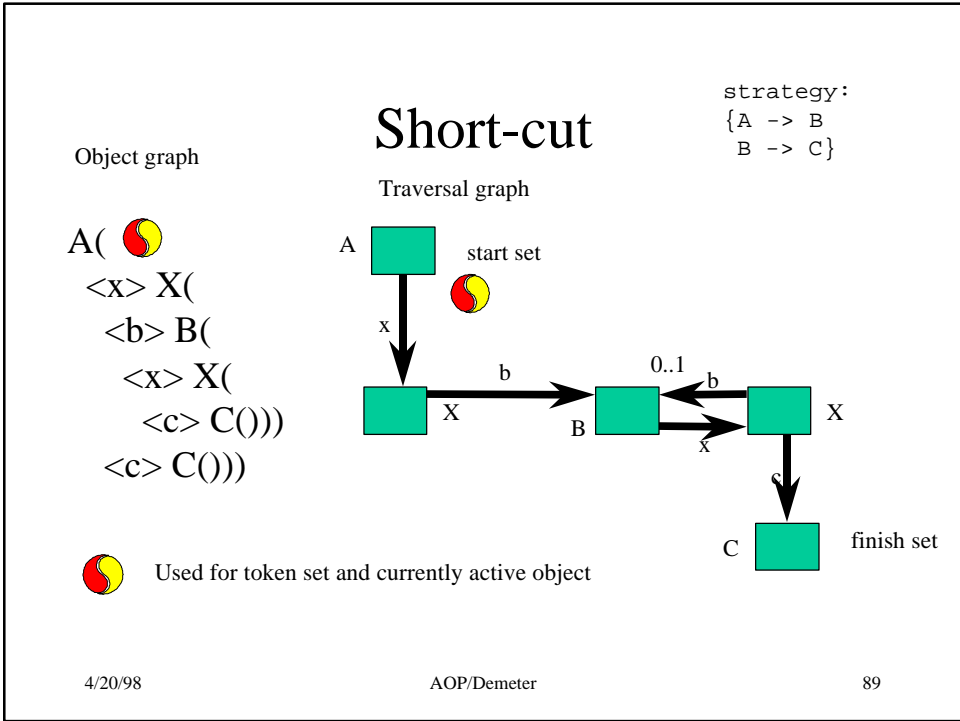
Traversal graph

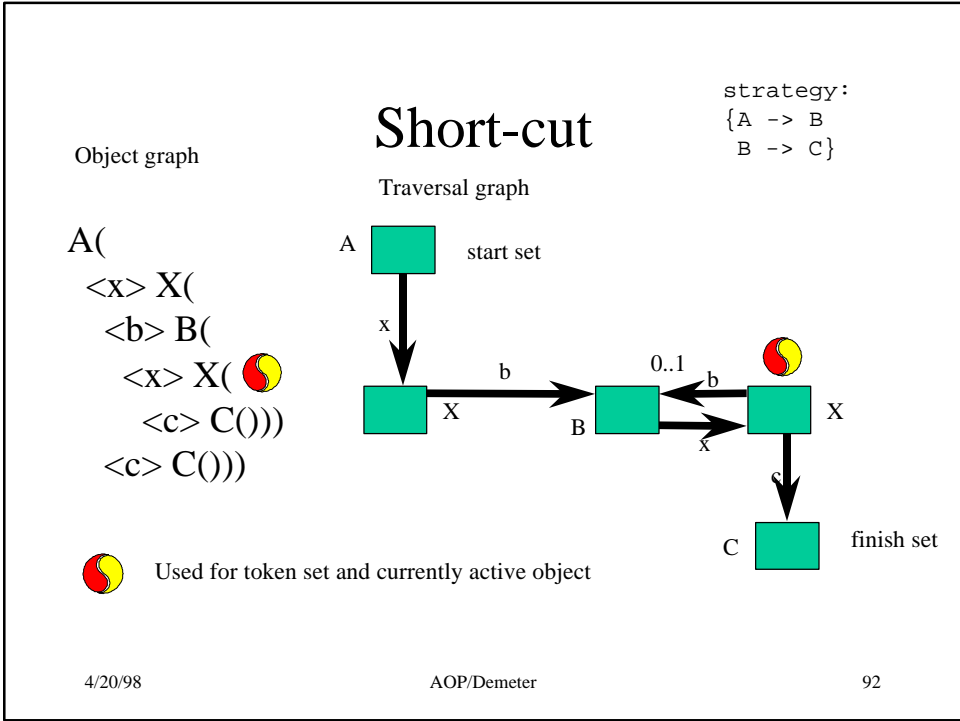
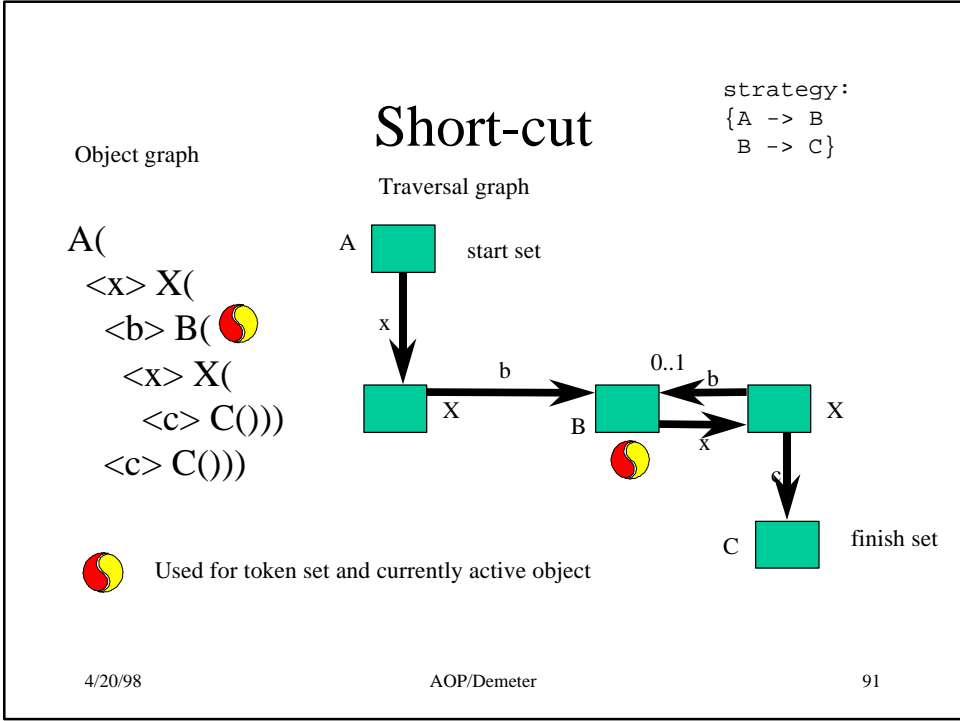


4/20/98

AOP/Demeter

88





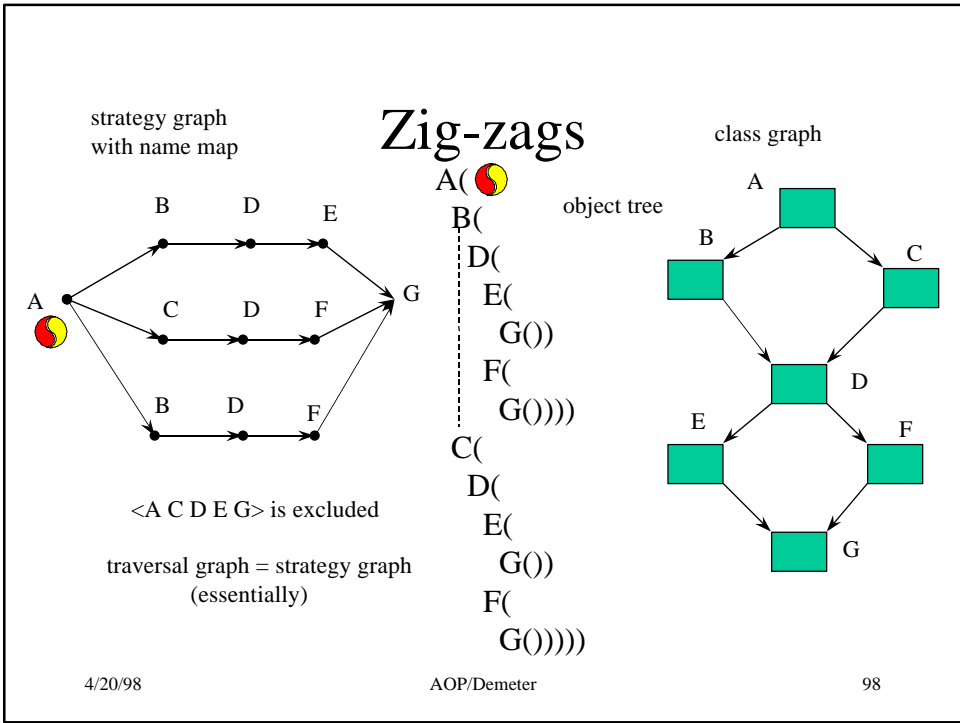
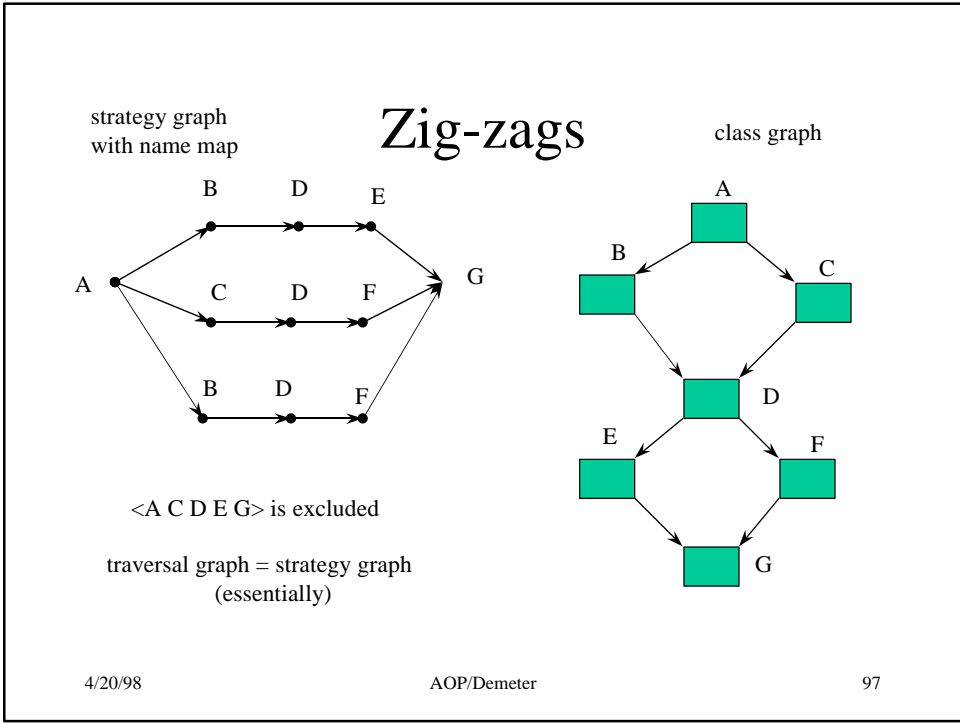
Traversal algorithm property

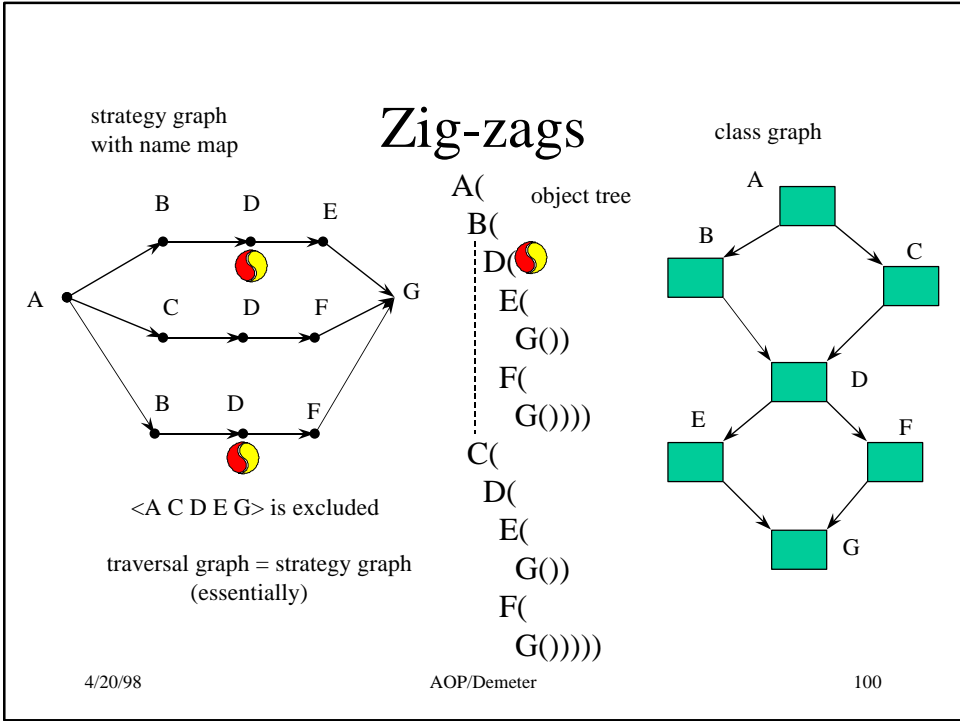
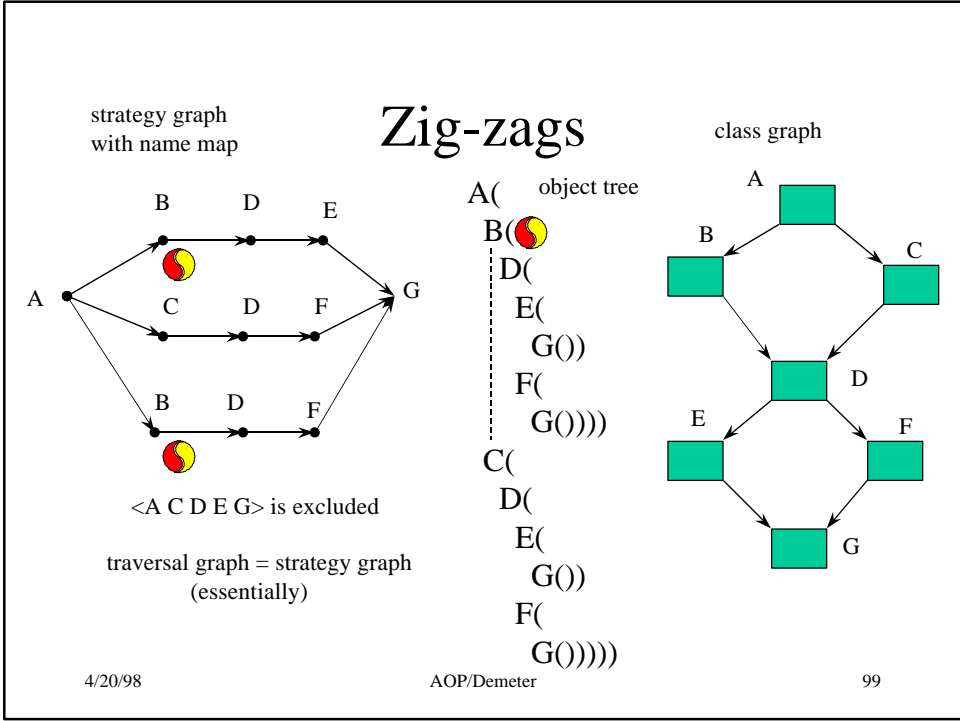
- Let O be an object tree and let o be an object in O . Suppose that the *Traverse* methods are guided by a traversal graph TG with finish set T_f . Let $H(o, T)$ be the sequence of objects which invoke visit while $o.Traverse(T)$ is active, where T is a set of nodes in TG . Then *traversing O from o guided by $X(P_{TG}(T, T_f))$ produces $H(o, T)$.*

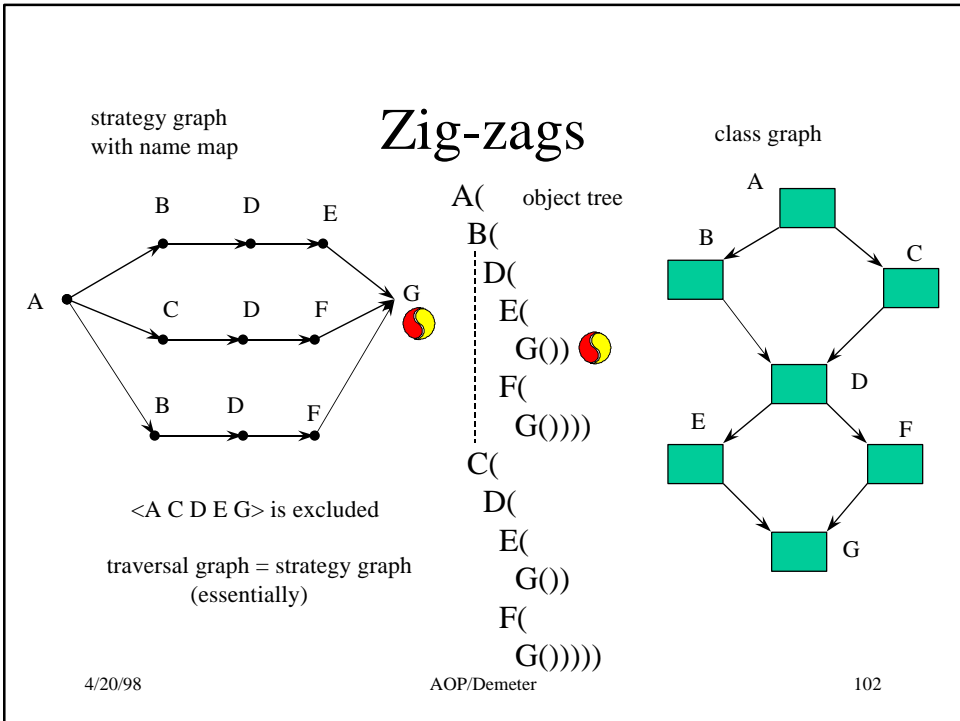
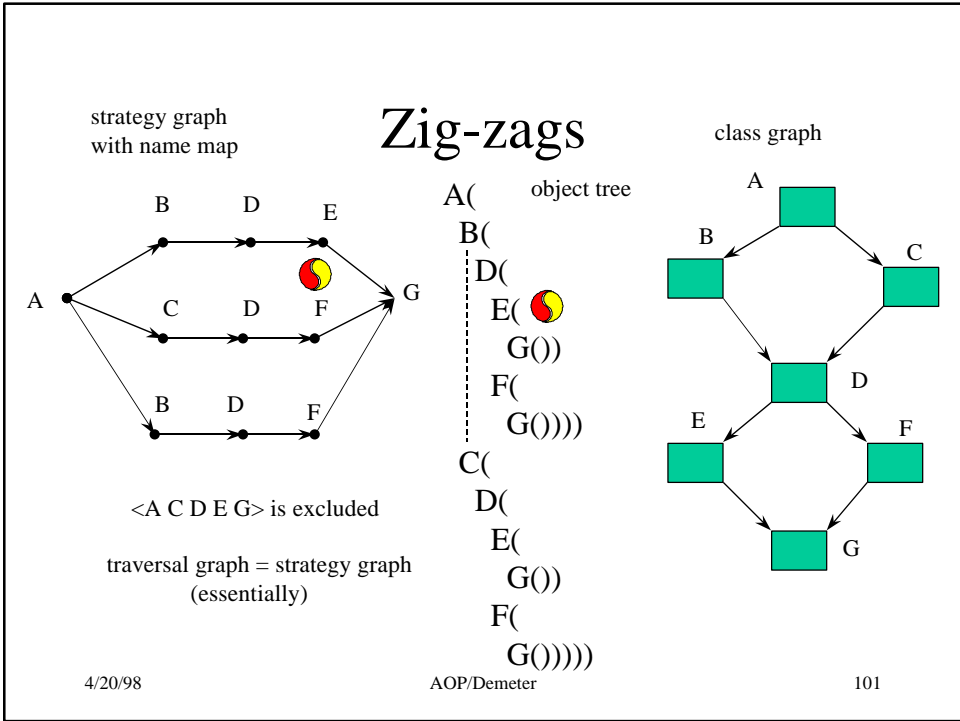


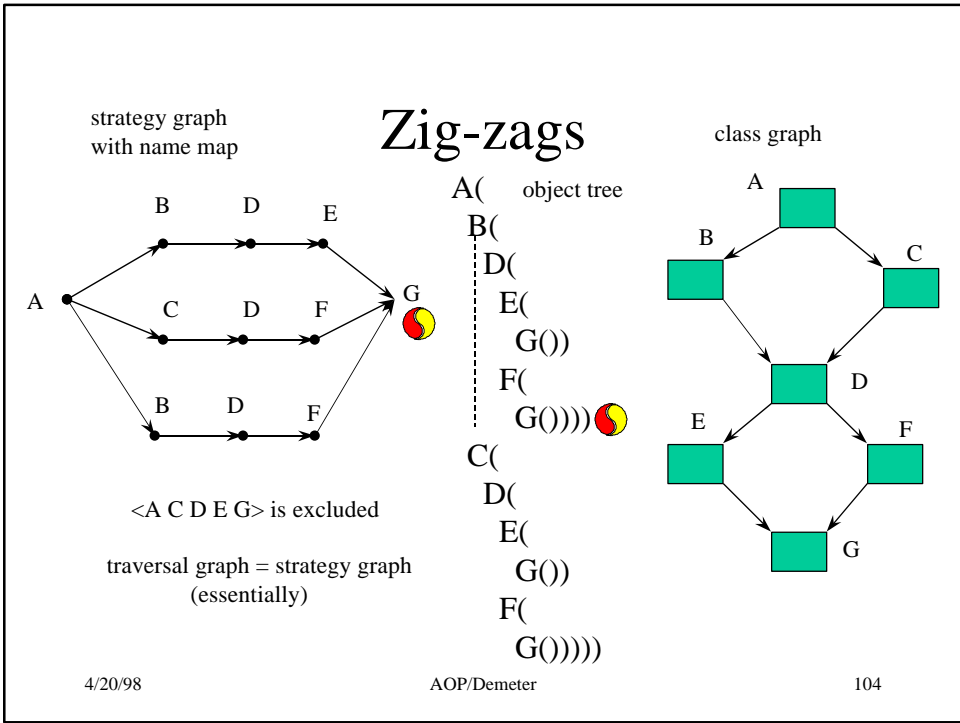
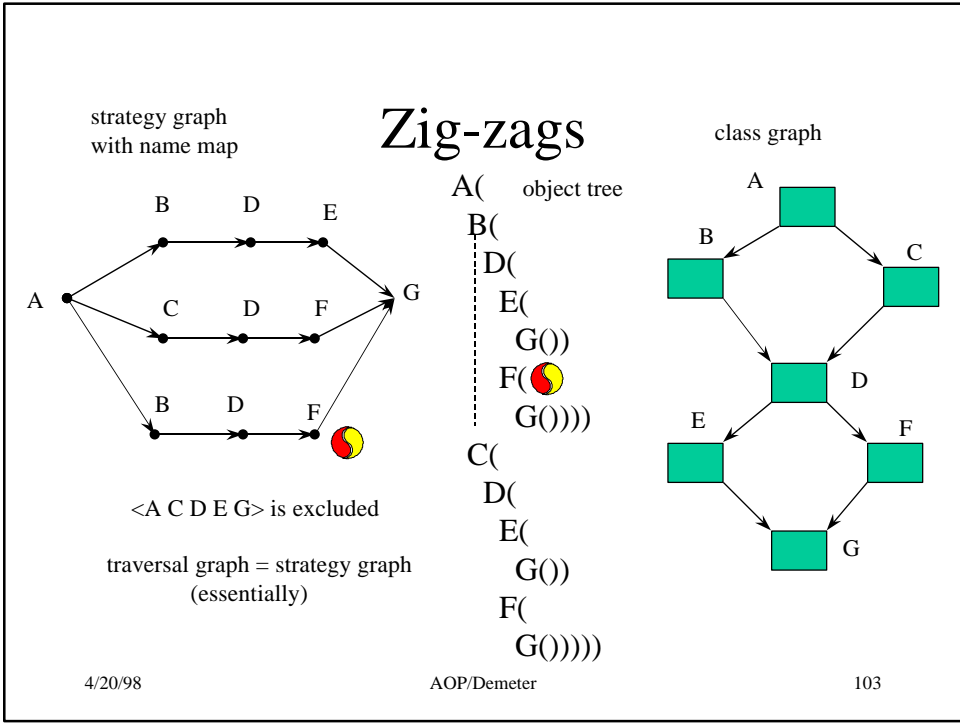
Example

- Using multiple tokens.
- Reuse zig-zag example.

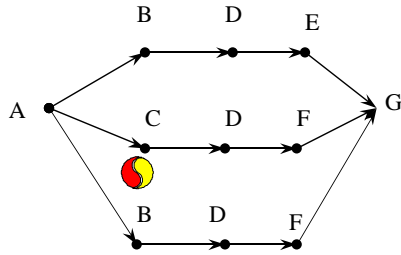








strategy graph
with name map



<A C D E G> is excluded

traversal graph = strategy graph
(essentially)

4/20/98

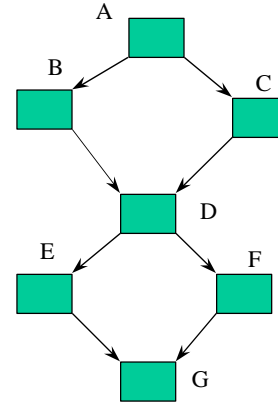
Zig-zags

object tree

```

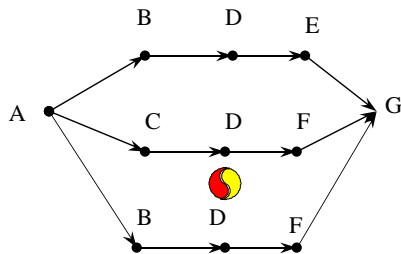
A(
  B(
    D(
      E(
        G())
      F(
        G()))
    C(
      D(
        E(
          G())
        F(
          G()))))
  
```

class graph



105

strategy graph
with name map



<A C D E G> is excluded

traversal graph = strategy graph
(essentially)

4/20/98

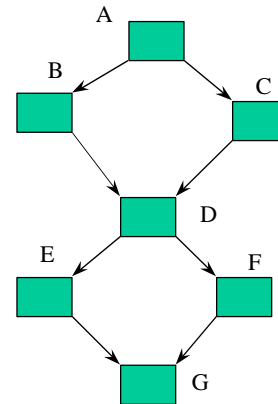
Zig-zags

object tree

```

A(
  B(
    D(
      E(
        G())
      F(
        G()))
    C(
      D(
        E(
          G())
        F(
          G()))))
  
```

class graph



106

Main Theorem

- Let SS be a strategy, let G be a class graph, let N be a name map, and let B be a constraint map. Let TG be the traversal graph generated by Algorithm 1, and let T_s and T_f be the start and finish sets, respectively.

Main Theorem (cont.)

- Let O be an object tree and let o be an object in O . Let H be the sequence of nodes visited when $o.Traverse$ is called with argument T_s , guided by TG . Then *traversing O from o guided by $SS[G,N,B]$ produces H .*

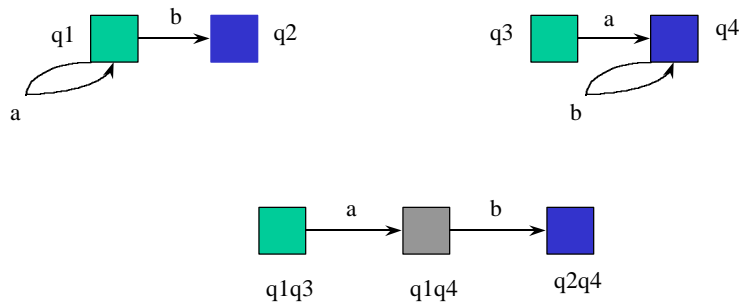
Complexity of algorithm

- Algorithm 1: All steps run in time linear in the size of their input and output. Size of traversal graph: $O(|S|^2 |G| d_0)$ where d_0 is the maximal number of edges outgoing from a node in the class graph.
- Algorithm 2: How many tokens? Size of argument T is bounded by the number of edges in strategy graph.

Evolution (continued)

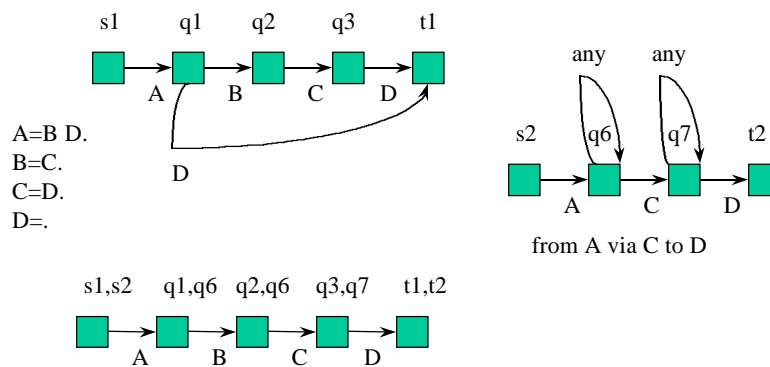
- Instead of using PL, use strategies: abstraction of class graph using regular expressions over class graph. We lose some of the flexibility of the traversal automata solution: strategies define only certain default traversals.
- Solution: use strategic traversal automata to gain flexibility.

Intersection of NDFA is similar to traversal graph construction



Intersection of two DFAs

Traversal Graph Construction



Integrated view of algorithms 1 and 2: for path existence

- Both are similar to the intersection of two NDFAs
 - Algorithm 1: NFA for strategy graph and NFA for class graph: results in NFA for traversal graph.
 - Algorithm 2: NFA for traversal graph and NFA for object graph: results in NFA which tells us whether there is a non-empty traversal

Recall: Intersection of NDFAs

- An NFA is a 5-tuple: $M=(S,A,d,p_0,F)$, S finite set of states, A is input alphabet, d is a state transition function which maps $A \times (S \cup \epsilon)$ to the set of subsets of S , p_0 is the initial state, and F is the set of final states.

Recall: Intersection of NDFAs

- $M1=(S1,A,d1,p0,F1)$ and $M2=(S2,A,d2,q0,F2)$.
- The NFA for $M1$ intersect $M2$ is $I=(S1 \times S2,A,d,(p0,q0),F1 \times F2)$, where for a in A , $(p2,q2)$ in $d((p1,q1),a)$ if and only if $p2$ in $d1(p1,a)$ and $q2$ in $d2(q1,a)$.

Simplifications of algorithm

- If no short-cuts and zig-zags, can use propagation graph. No need for traversal graph. Faster traversal at run-time.
- Presence of short-cuts and zig-zags can be checked efficiently (compositional consistency).
- See chapter 15 of AP book.

Extensions

- Multiple sources
- Multiple targets
- Intersection of traversals

Summary

- Abstract model behind strategy graphs.
- How to implement strategy graphs.
- How to apply: Precise meaning of strategies; how to write traversals manually (watch for short-cuts and zig-zags).

Where to get more information

- Paper with Boaz-Patt Shamir (strategies.ps in my FTP directory)
- Implementation of Demeter/Java shows you how algorithms are implemented in Demeter/Java (and Java). See Demeter/Java resources page.
- Chapter 15 of AP book.

Coordination aspect

- Review of AOP
- Summary of threads in Java
- COOL (COOrdination Language)
 - Design decisions
 - Implementation at Xerox PARC and for Demeter/Java

“To my taste the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects. ...

Quote taken from Gregor Kiczales' talk:
www.parc.xerox.com/aop

- Dijkstra, *A discipline of programming*, 1976
last chapter, **In retrospect**

4/20/98

AOP/Demeter

121

A few more viewgraphs taken from Gregor Kiczales' talk
www.parc.xerox.com/aop

4/20/98

AOP/Demeter

122

the goal is a clear separation of concerns

we want:

- natural decomposition
- concerns to be cleanly localized
- handling of them to be explicit
- in both design and implementation

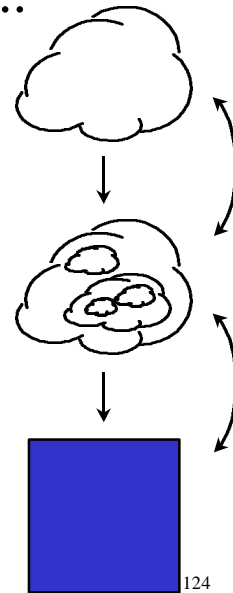


achieving this requires...

- synergy among
 - problem structure and
 - design concepts and
 - language mechanisms

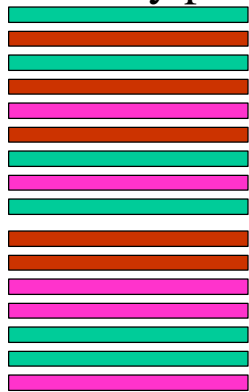
“natural design”

“the program looks like the design”

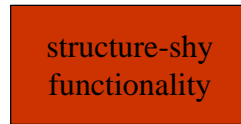


Cross-cutting of components and aspects

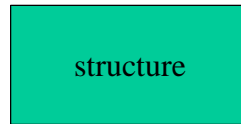
ordinary program



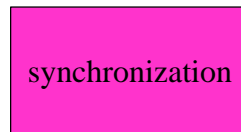
better program



Components

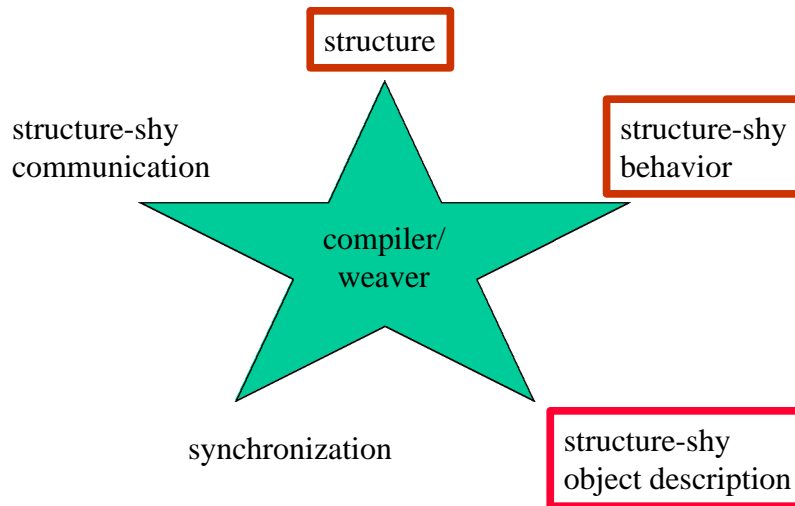


Aspect 1



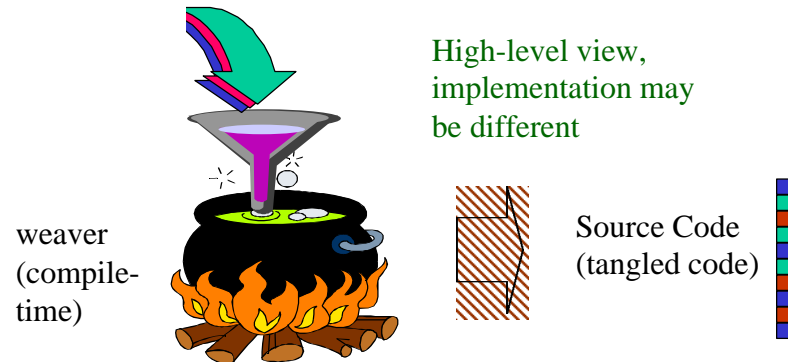
Aspect 2

Demeter



Aspect-Oriented Programming

components and aspect descriptions



4/20/98

AOP/Demeter

127

Technology Evolution

Object-Oriented Programming



Law of Demeter dilemma
Tangled structure/behavior

Adaptive Programming



Other tangled code

Aspect-Oriented Programming

4/20/98

AOP/Demeter

128

Components/Aspects of Demeter

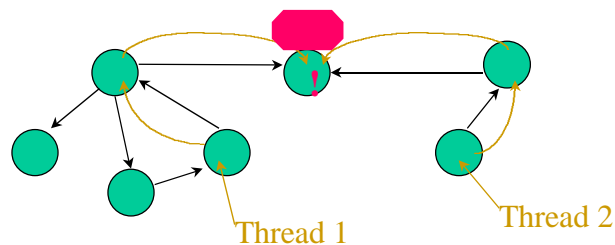
- Functionality (structure-shy)
 - Traversal (Traversal Strategies)
 - Functionality Modification (Visitors)
- Structure (UML class diagrams)
- Description (annotated UML class diagrams, class dictionaries)
- Synchronization

4/20/98

AOP/Demeter

129

Threads



4/20/98

AOP/Demeter

130

Coordination aspect

- Put coordination code about thread synchronization in one place.
- Threads are synchronized through methods.
- Method synchronization
 - Exclusion sets
 - Method managers

Java Threads

- Thread class in Standard Java libraries
- Thread worker = new Thread()
- start method spawns a new thread of control based on Thread object. start invokes the threads run method: active thread

Java Threads

- synchronized method: locks object. A thread invoking a synchronized method on the same object must wait until lock released. Mutual exclusion of two threads.

```
class Account {  
    synchronized double getBalance() { return balance;}  
    synchronized void deposit(double a) { balance += a;}  
}
```

Problem with synchronization code: it is tangled with component code

```
class BoundedBuffer {  
    Object[] array;  
    int putPtr = 0, takePtr = 0;  
    int usedSlots = 0;  
    BoundedBuffer(int capacity){  
        array = new Object[capacity];  
    }  
}
```

Tangling

```
synchronized void put(Object o) {  
    while (usedSlots == array.length) {  
        try { wait(); }  
        catch (InterruptedException e) {};  
    }  
    array[putPtr] = o;  
    putPtr = (putPtr + 1 ) % array.length;  
    if (usedSlots==0) notifyall();  
    usedSlots++;  
    // if (usedSlots++==0) notifyall();  
}
```

4/20/98

AOP/Demeter

135

Solution: tease apart basics and synchronization

- write core behavior of buffer
- write coordinator which deals with synchronization
- use weaver which combines them together
- simpler code
- replace `synchronized`, `wait`, `notify` and `notifyall` by coordinators

4/20/98

AOP/Demeter

136

Coordinator

method managers with *requires* clauses and *entry/exit* clauses

```
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; @)}  
}
```

exclusion sets

- selfex {f,g}
 - only one thread can call a selfex method
 - A and B are unrelated
- mutex {g,h,i} mutex {f,k,l}
 - if a thread calls a method in a mutex set, no other thread may call a method in the same mutex set.

Design decisions behind COOL

- The smallest unit of synchronization is the method.
- The provider of a service defines the synchronization (monitor approach).
- Coordination is contained within one coordinator.
- Association from object to coordinator is static.

4/20/98

AOP/Demeter

141

Design decisions behind COOL

- Deals with thread synchronization within each execution space. No distributed synchronization.
- Coordinators can access the objects' state, but they can only modify their own state. Synchronization does not “disturb” objects. Currently a design rule not checked by implementation.

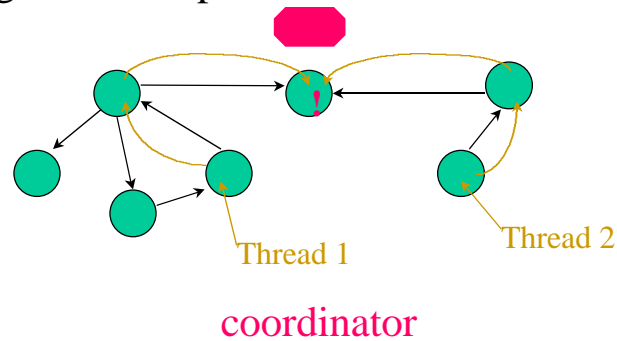
4/20/98

AOP/Demeter

142

COOL

- Provides means for dealing with mutual exclusion of threads, synchronization state, guarded suspension and notification



4/20/98

AOP/Demeter

143

COOL

- Identifies “good” abstractions for coordinating the execution of OO programs
 - coordination, not modification of the objects
 - mutual exclusion: sets of methods
 - preconditions on methods
 - coordination state (history-sensitive schemes)
 - state transitions on coordination

4/20/98

AOP/Demeter

144

plain Java

```
public class Shape {
    protected double x_ = 0.0;
    protected double y_ = 0.0;
    protected double width_ = 0.0;
    protected double height_ = 0.0;

    double x() { return x_(); }
    double y() { return y_(); }
    double width(){
        return width_();
    }
    double height(){
        return height_();
    }
    void adjustLocation() {
        x_ = longCalculation1();
        y_ = longCalculation2();
    }
    void adjustDimensions() {
        width_ = longCalculation3();
        height_ = longCalculation4();
    }
}
```

4/20/98

AOP/Demeter

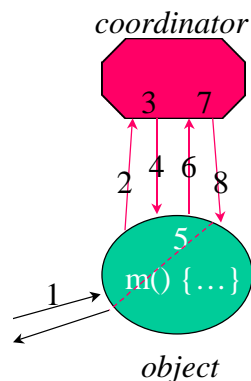
COOL Shape

```
coordinator Shape {
    selfex {adjustLocation,
            adjustDimensions}
    mutex {adjustLocation,x}
    mutex {adjustLocation,y}
    mutex {adjustDimensions,
            width}
    mutex {adjustDimensions,
            height}
}
```

145

Programming with COOL

Protocol object/coordinator:



4/20/98

AOP/Demeter

8. method returns

146

COOL View of Classes

- Stronger visibility:
 - coordinator can access:
 - all methods and variables of its classes, independent of access control
 - all non-private methods and variables of their superclasses
- Limited actions:
 - only read variables, not modify them
 - only coordinate methods, not invoke them

4/20/98

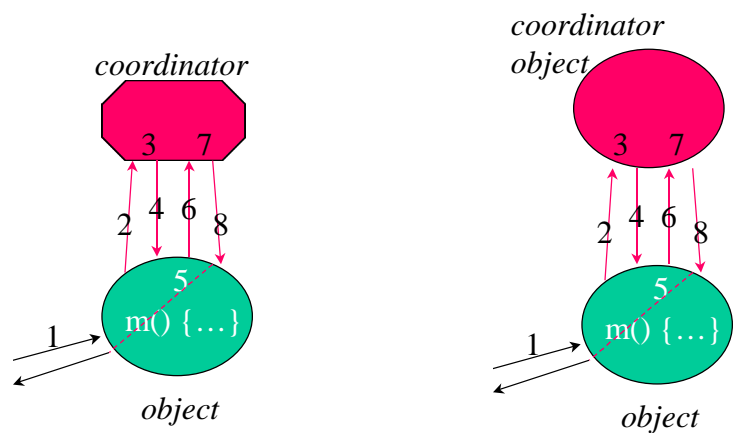
AOP/Demeter

147

Xerox PARC Implementation

Programming with COOL

Implementing COOL



Semantics

Implementation

4/20/98

AOP/Demeter

148

COOL

```
public class BoundedBuffer {
  private Object array[];
  private int putPtr = 0, takePtr = 0;
  private int usedSlots = 0;

  public BoundedBuffer(int capcty){
    array = new Object[capcty];
  }

  public void put(Object o) {
    array[putPtr] = o;
    putPtr = (putPtr+1)%array.length;
    usedSlots++;
  }

  public Object take() {
    Object old = array[takePtr];
    array[takePtr] = null;
    takePtr = (takePtr+1)%array.length;
    usedSlots--;
    return old;
  }
}
```

```
coordinator BoundedBuffer {
  selfex {put, take}
  mutex {put, take};
  boolean full=(@ false @),
    empty=(@ true @);
  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;
    @)
  }
}
```

4/20/98

AOP/Demeter

149

Acknowledgments

- Many of the viewgraphs prepared by Crista Lopes for her Ph.D. work supported by Xerox PARC.
- Implementation of COOL for Demeter/Java by Josh Marshall. Integration into Demeter/Java with Doug Orleans.
- ECOOP '94 paper on synchronization patterns by Lopes/Lieberherr.

4/20/98

AOP/Demeter

150

Applications to UML

UML
Model architecture
Object Constraint Language

4/20/98

AOP/Demeter

151

UML language architecture

- UML metamodel defines meaning of UML models
- Defined in a metacircular manner, using a subset of UML to specify itself
- UML metamodel bootstraps itself. Similar:
 - compiler compiles itself
 - grammar defines itself
 - class dictionary defines itself

4/20/98

AOP/Demeter

152

4 layer metamodel architecture

- UML metamodel one of the layers
- Why four layers?
- Proven architecture for complex models
- Validates core constructs by using them to define themselves

Four layer architecture

- meta-metamodel
 - language for specifying metamodels
- metamodel
 - language for specifying models
- model
 - language for specifying objects in some domain
- user objects

Four levels

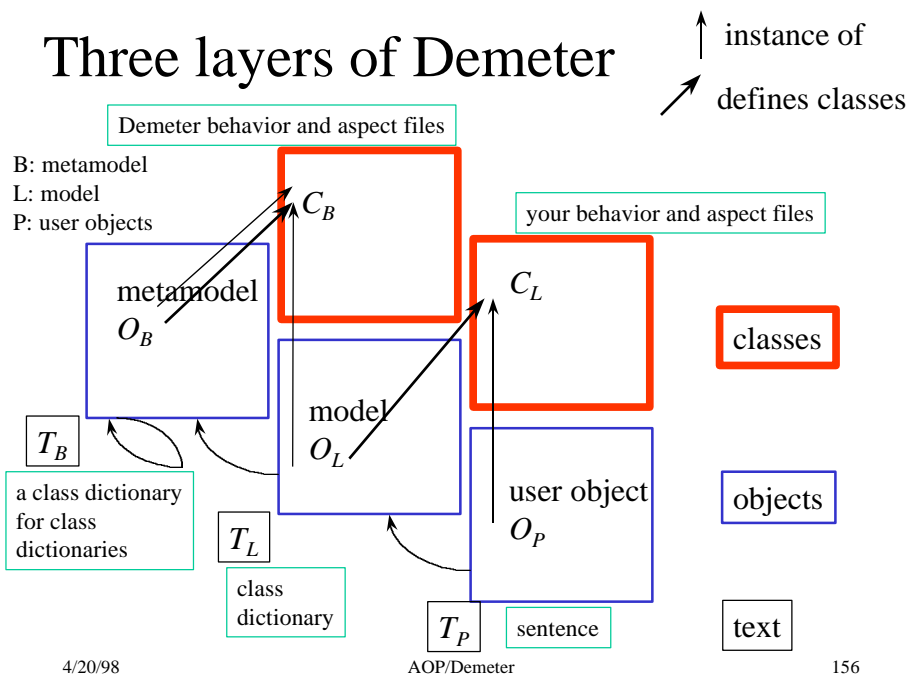
- User Objects in running system
 - check run-time constraints
- Model of System under design
 - specify run-time constraints
- Meta-model
 - specify constraints on use of constructs in model
- Meta-metamodel
 - data interchange between modeling tools

4/20/98

AOP/Demeter

155

Three layers of Demeter



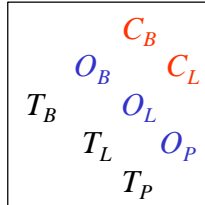
4/20/98

AOP/Demeter

156

Icon

Demeter Tiling



Use as reminder for
Demeter Tiling.

UML OCL

- Object Constraint Language
 - allows you to define side effect-free constraints for UML and other models (for example adaptive programs)
 - used in UML to defined well-formedness rules of the UML meta model (invariants for meta model classes)

Why OCL

- It is a formal mathematical language
- Tend to be hard to use by average modelers
- OCL is intended for average modelers
- Developed as business modeling language within IBM insurance division (has roots in Syntropy method)
- OCL is a pure expression language (side effect free)

4/20/98

AOP/Demeter

159

Companies behind OCL

- Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, Softeam

4/20/98

AOP/Demeter

160

Where to use OCL?

- Specify invariants for classes and types
- Specify pre- and post-conditions for methods
- As a navigation language

OCL properties

- LL(1) language
 - finally back to the Pascal days!
 - Grammar provided uses EBNF syntax
- Parser generated by JavaCC

What is OCL?

- Predicate calculus for objects
- Traditional predicate calculus:
 - individuals
 - variables, predicate and function symbols
 - terms (for all, Boolean connectives)
 - axioms and the theories they define (group theory, number theory, etc.)
- In OCL: individuals \rightarrow objects

Structured individuals

- some “structural” constraints imposed by UML class diagram; further constraints can be imposed by OCL expressions
- annotated UML class diagram defines textual representation

Connection to model

- Self. Each OCL expression is written in the context of an instance of a specific type.

Company

```
self.numberOfEmployees
```

c : Company

```
c.numberOfEmployees
```

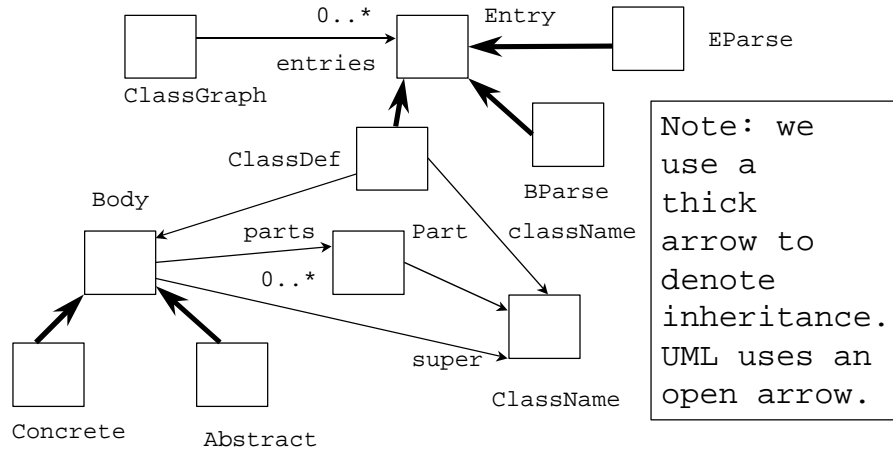
Connection to model

- Invariants of a type. An OCL expression stereotyped with <<invariant>>. An invariant must be true for all instances of the type at any time.

Person

```
self.age >= 0
```

Example: UML class diagram ClassGraph

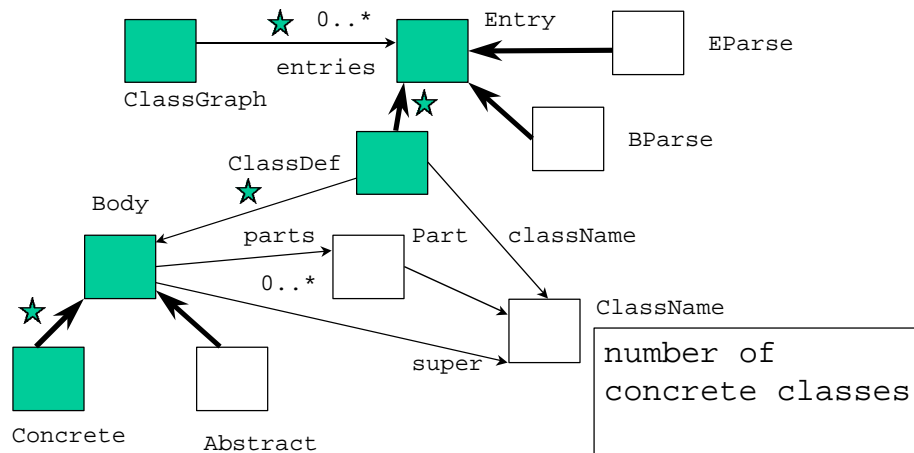


4/20/98

AOP/Demeter

167

UML class diagram ClassGraph



4/20/98

AOP/Demeter

168

Example

```
-> collection op  
select:subset  
collect:new set
```

- Number of concrete classes:

```
- ClassGraph self.entries->  
  select(c:Entry|c.  
    oclIsTypeOf(ClassDef))->  
    collect(body)->  
      select (b:Body|b.  
        oclIsTypeOf(Concrete))  
->size
```

Pre- and post-conditions

- constraints stereotyped with
<<precondition>> and <<postcondition>>
for an operation or method. Example:

```
Type::op(param1 : Type1 ...):  
  ReturnType  
  pre: param1 ...  
  Post: result = ...
```

Navigation over associations

- Company `self.manager`
object of type `Person` or `Set(Person)`
 - used as `Set(Person)`
`self.manager->size` -- result 1
 - used as `Person`
`self.manager.age`

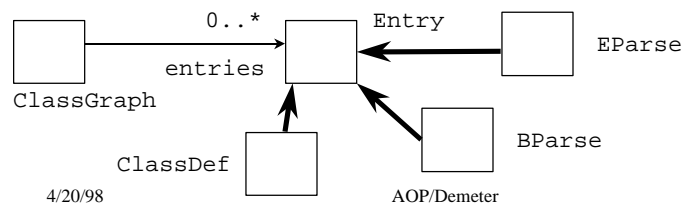
4/20/98

AOP/Demeter

171

Applications

- Number of class definitions:
 - ClassGraph `self.entries->size` **wrong**
 - ClassGraph `self.entries->`
`select(c:Entry|c.`
`oclIsTypeOf(ClassDef))->size`



4/20/98

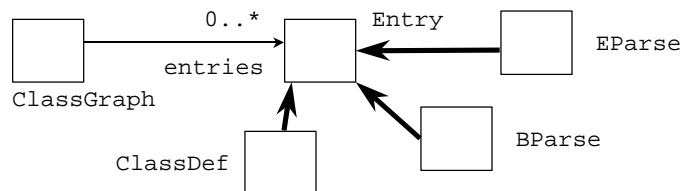
AOP/Demeter

172

Applications



- Number of class definitions: What about using strategies to define collections?
 - `ClassGraph self.{to ClassDef}`
`->size`



4/20/98

AOP/Demeter

173

Improve OCL: make adaptive

- OCL stresses the importance of collections
- Collections are best specified adaptively
- A strategy $SS = (S, B, s, t)$ with source s and target set t and name map N for class graph G defines a collection of objects contained in a $N(s)$ -object. The collection type CT is the union of $N(t_1)$ for t_1 in t .

4/20/98

AOP/Demeter

174

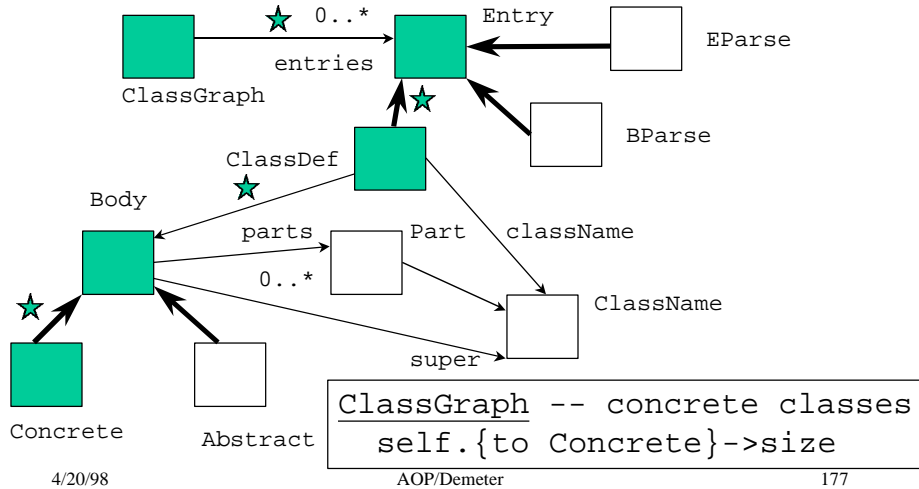
Improve OCL

- The collection consists of *CT*-objects reached during the traversal of the $N(s)$ object following strategy *SS*.

Properties

- In OCL
 - an attribute
 - an association end
 - an operation with *isQuery* true
 - a method with *isQuery* true
- Add for adaptive OCL
 - a strategy { ... } with a single source

UML class diagram ClassGraph

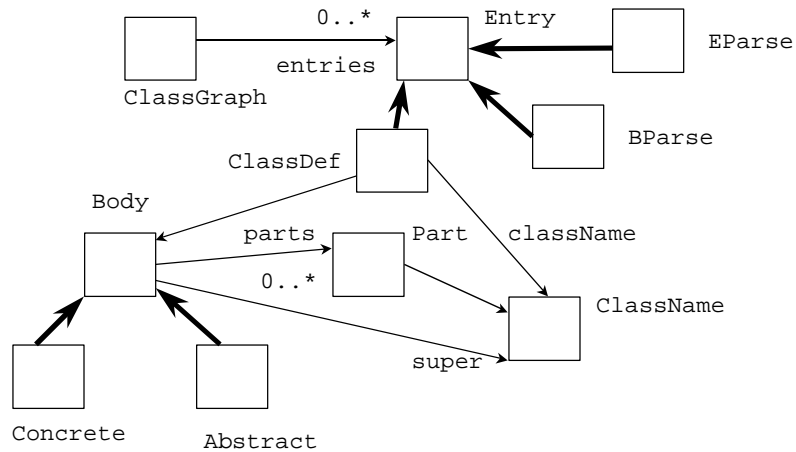


```
ClassGraph self.entries->
  select(c:Entry|c.
    oclIsTypeOf(ClassDef))->
    collect(body)->
      select (b:Body|b.
        oclIsTypeOf(Concrete))
    ->size -- count concrete classes
```

```
ClassGraph -- count concrete classes
self.{to Concrete}->size
```

Which one is easier to write?

UML class diagram ClassGraph



4/20/98

AOP/Demeter

179

Applications

- Terminal buffer rule

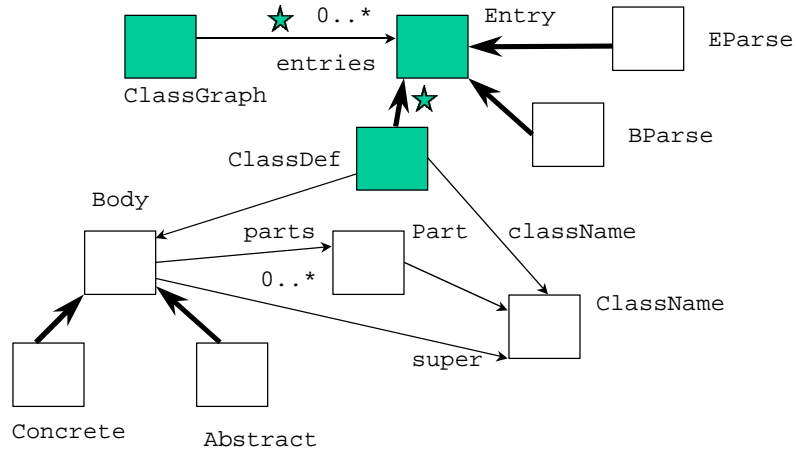
```
ClassGraph self.{to ClassDef}
->forall(r|r.termBProp())
ClassDef Boolean termBProp(){
  partCNS=self.{via Part to ClassName};
  result=if (partCNS->size)>1 then
    (partCNS->intersection(predefCNS))
    -> isEmpty
  else true endif}
```

4/20/98

AOP/Demeter

180

UML class diagram ClassGraph

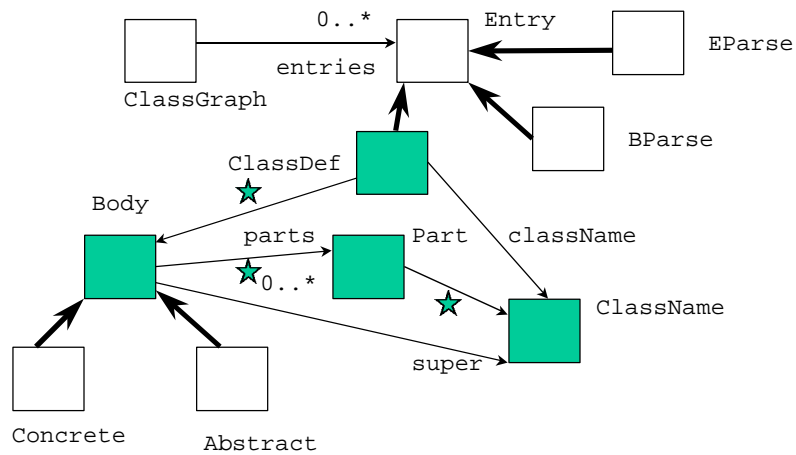


4/20/98

AOP/Demeter

181

UML class diagram ClassGraph

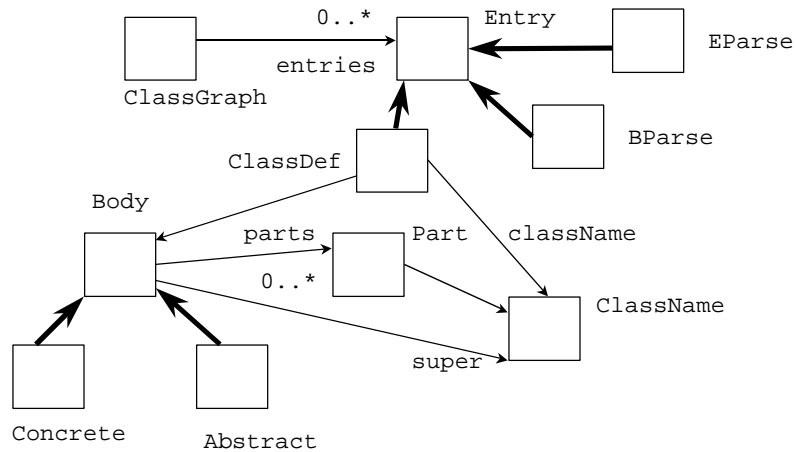


4/20/98

AOP/Demeter

182

UML class diagram ClassGraph



4/20/98

AOP/Demeter

183

Applications

- Class graph is flat

ClassGraph

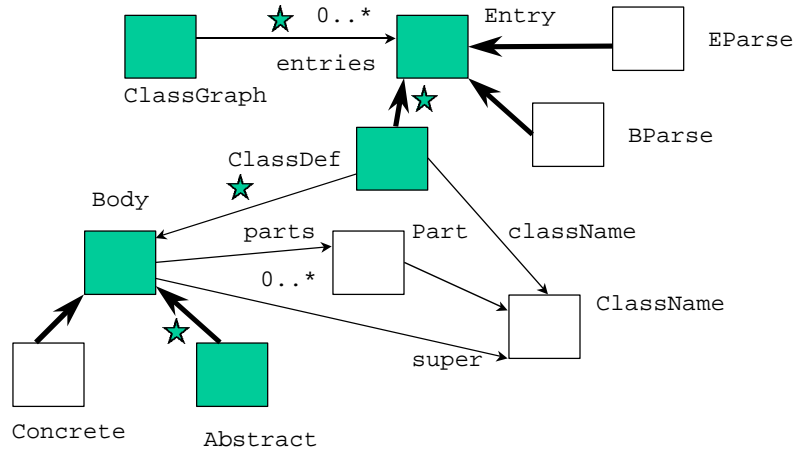
```
self.{to Abstract}->
  forAll(a|a.parts->size=0)
```

4/20/98

AOP/Demeter

184

UML class diagram ClassGraph

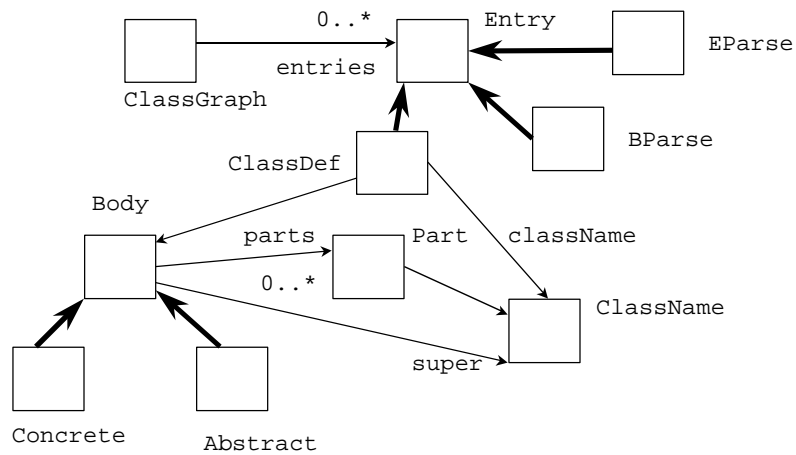


4/20/98

AOP/Demeter

185

UML class diagram ClassGraph



4/20/98

AOP/Demeter

186

Applications

- Abstract superclass rule

ClassGraph

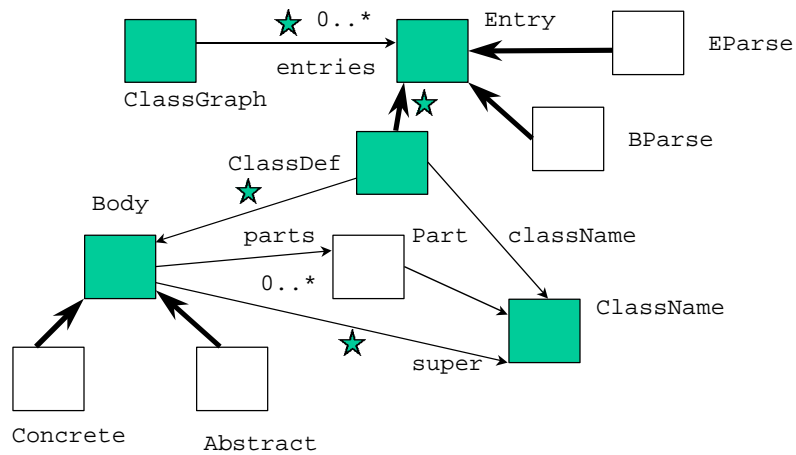
```
superCls =  
  self.{through->*,super,* to ClassName};  
self.{to ClassDef}->  
  forAll(c|  
    if (superCls->includes(c.className))  
    then c.{to Abstract}->size=1  
    else true  
  endif)
```

4/20/98

AOP/Demeter

187

UML class diagram ClassGraph

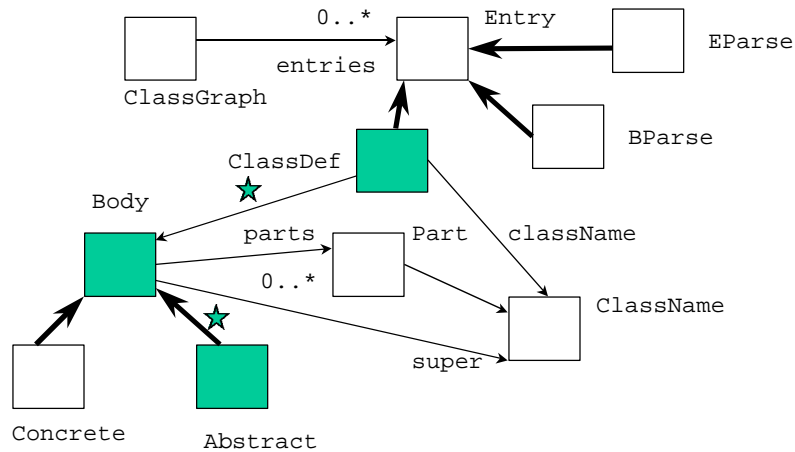


4/20/98

AOP/Demeter

188

UML class diagram ClassGraph



4/20/98

AOP/Demeter

189

Conclusions

- OCL is a suitable language for expressing object properties, class invariants and method pre- and post-conditions. (needs capability to define functions and auxiliary variables).
- OCL is NOT a good language for navigation but can be made into one by adding strategies.

4/20/98

AOP/Demeter

190

Further information

- www.rational.com contains latest information about UML, specifically OCL.
- www.ics.uci.edu/pub/arch/uml

Feedback

- Send email to demeter@ccs.neu.edu.