

Lecture 7

- Traversals and visitors on abstract classes
- Notations for strategies
- Metric for structure-shyness
- Demeter Method, including Law of Demeter
- Project steps
- Class dictionary kinds

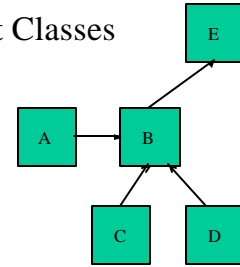
11/6/00

AOO/Demeter

1

Traversals / Visitor Methods on Abstract Classes

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



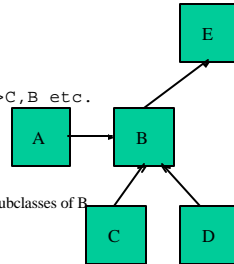
11/6/00

AOO/Demeter

2

Traversals to Abstract Classes

- from A to B =
- includes =>B,C and :>C,B etc.
- Motivation:
 - from A to E also includes subclasses of B



11/6/00

AOO/Demeter

3

Path concept

- Path from A to B
 - include construction edges (forward)
 - include inheritance edges (backward and forward). Backward inheritance edges are called subclass edges.
 - inheritance edge implies subclass edge in opposite direction.
 - follow rule: after an inheritance edge, you are not allowed to follow a subclass edge.

11/6/00

AOO/Demeter

4

Path concept

- Path from A to B:
 - EI implies EA in opposite direction
 - $(EC | EA | EI)^*$ but not EA followed by EI
 - $((EI^* EC) | EA)^* EI^*$
 - checking the eight allowed edge pairs: $(EI, EC), (EI, EI), (EC, EC), (EA, EI), (EC, EI), (EA, EA), (EC, EA), (EA, EC)$

11/6/00

AOO/Demeter

5

Path concept in flat class graphs

- Path from A to B:
 - $(EC | EA | EI)^*$ but not EI followed by EA
 - $((EI^* EC) | EA)^* EI^* = (EC|EA)^* EI^*$
- in flat class graph there is never a construction edge following an inheritance edge: $(EI^* EC) = EC$
- EI implies EA in opposite direction


11/6/00

AOO/Demeter

6

from A to C
from A to B
from A to E

from A to D



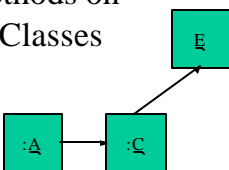
```

graph TD
    A[A] --> B[B]
    A --> C[C]
    B --> D[D]
    B --> E[E]
  
```

11/6/00 AOO/Demeter 7

Visitor Methods on Abstract Classes

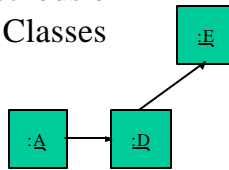
- from A to E: **b, c**
- from A to B: **b, c**
- from A to C: **b, c**
- visitor:
 - before (B){p("b");}
 - before (C){p("c");}



11/6/00 AOO/Demeter 8

Visitor Methods on Abstract Classes

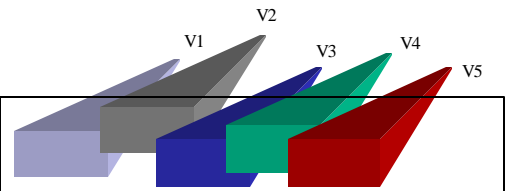
- from A to E: **b**
- from A to B: **b**
- from A to C: **b**
- visitor:
 - before (B){p("b");}
 - before (C){p("c");}



11/6/00 AOO/Demeter 9

View of AP

Multiple views of the same class graph



Application class graph

In DemeterJ: view = strategy graph or view = class graph with edges deleted.

11/6/00 AOO/Demeter 10

Apply idea again

- Each view has a class graph
- Define views of that view class graph
- Example: views by deletion
 - ClassGraph cg = new ClassGraph(true,false);
 - ClassGraph view1 = new ClassGraph(cg, s1);
 - s1 = "from Commands bypassing >*,tail,* to *";
 - ClassGraph view2 = new ClassGraph(view1, s2);
 - s2 also uses bypassing

11/6/00 AOO/Demeter 11

DemeterJ/DJ notation for strategies

- Notations via = through
 - line graph notation


```

from BookKeeping
via Taxes via Business
to LineItem
          
```
 - strategy graph notation


```

{BookKeeping -> Taxes
Taxes -> Business
Business -> LineItem }
          
```

11/6/00 AOO/Demeter 12

Bypassing

- line graph notation

```
from BookKeeping
  via Taxes bypassing HomeOffice
  via Business
to LineItem
```
- strategy graph notation

```
{BookKeeping -> Taxes
Taxes -> Business bypassing HomeOffice
Business -> LineItem }
```

11/6/00

AOO / Demeter

13

Strategies by example

- Single-edge strategies
- Star-graph strategies
- Basic join strategies
- Edge-controlled strategies
- The wild card feature
- Preventing recursion
- Surprise paths

11/6/00

AOO / Demeter

14

Single-edge strategies

- Fundamental building blocks of general strategies
- Can express any subgraph of a class graph
 - not expressive enough
- No-pitfall strategies
 - subgraph summarizes path set correctly

11/6/00

AOO / Demeter

15

Traversal graph: From A to B

- Reverse all inheritance edges and call them subclass edges.
- Flatten all inheritance by expanding all common parts to concrete subclasses.
- Find all classes reachable from A and color them red including the edges traversed.

11/6/00

AOO / Demeter

16

Traversal graph: From A to B

- Find all classes from which B is reachable and color them blue including the edges traversed.
- The group of collaborating classes is the set of classes and edges colored both red and blue.

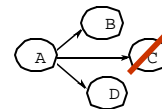
11/6/00

AOO / Demeter

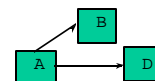
17

Traversal graph controls traversal

- object graph



- traversal graph



11/6/00

AOO / Demeter

18

Traversal graph and bypassing

- Take bypassed classes out of the class graph including edges incident with them

```
{ BusRoute -> Person  
  bypassing Bus }
```

11/6/00

AOO/Demeter

19

Traversal graph and bypassing

- May bypass a set of classes

```
{ BusRoute -> Person  
  bypassing {Bus, BusStop}  
}
```

11/6/00

AOO/Demeter

20

only-through

- is complement of bypassing
- ```
{A -> B
 only-through {-> A,b,B}}
```
- bypass all edges not in only-through set

11/6/00

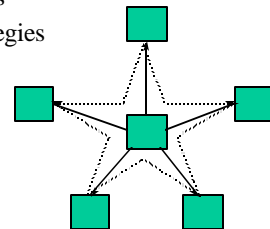
AOO/Demeter

21

## Star-graph strategies

- Multiple targets
- No-pitfall strategies

from A  
to {B,C,D,E,F}



11/6/00

AOO/Demeter

22

## Star-graph strategies

```
Company {
 ... bypassing {...} to {Customer, SalesAgent} ...
}

{Company -> Customer bypassing {...}
 Company -> SalesAgent bypassing {...}
}
```

11/6/00

AOO/Demeter

23

## Basic join strategies

- Join two single edge strategies  

```
from Company bypassing {...}
 through Customer
to Address

{Company->Customer bypassing{...}
 Customer->Address}
```

11/6/00

AOO/Demeter

24

## Multiple join points

from Company  
 through {Secretary, Manager}  
 to Salary

```
{ Company -> Secretary,
 Company -> Manager,
 Secretary -> Salary,
 Manager -> Salary }
```

11/6/00

AOO/Demeter

25

## Edge-controlled strategies

- Class-only strategies are preferred
- They do not reveal details about the part names
- Use whenever possible

11/6/00

AOO/Demeter

26

## Edge notation

- $-> A, b, B$  construction edge from A to B
- $=> A, B$  subclass edge from A to B
- set of edges:
 

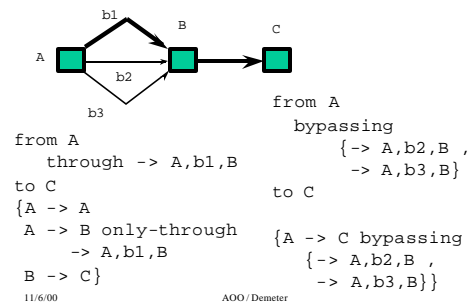
```
{ -> A, b, B ,
 -> X, Y, Y ,
 => R, S }
```

11/6/00

AOO/Demeter

27

## Need edge-control



11/6/00

AOO/Demeter

28

## Wild card feature

- For classes and labels may use \*
- line graph notation
 

```
from A bypassing B to *
```
- strategy graph notation
 

```
{A -> * bypassing B}
```
- Gain more adaptiveness; can talk about classes we don't know yet.

11/6/00

AOO/Demeter

29

## Preventing Recursion

```
From Conglomerate
to-stop Company
equivalent to
from Conglomerate
bypassing {-> Company, *, * ,
=> Company, *}
to Company
```

11/6/00

AOO/Demeter

30

## simulating to-stop

```
{Conglomerate -> Company
 bypassing {-> Company,**,
 => Company,*}
}
```

All edges from targets are bypassed.  
What is the meaning of: from A to-stop A

11/6/00

AOO / Demeter

31

## Surprise paths

- {A -> B B -> C}
- surprise path: A P C Q A B R A S C
- eliminate surprise paths:
  - {A->B bypassing {A,B,C}
  - B->C bypassing {A,B,C}}
- {A->A bypassing A}

11/6/00

AOO / Demeter

32

## Wysiwig strategies

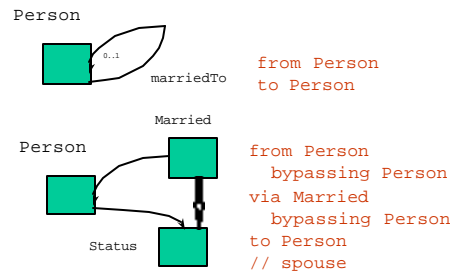
- Avoid surprise paths
- Bypass all classes mentioned in strategy on all edges of the strategy graph
- Some users think that wysiwig strategies are easier to work with
- For wysiwig strategies, if class graph has a loop, strategy must have a loop.

11/6/00

AOO / Demeter

33

## When to avoid strategies?



11/6/00

AOO / Demeter

34

## When to avoid strategies

- Either write your class graphs without self loops (a construction edge from A to A) by introducing additional classes or
- Avoid the use of strategies for traversing through a self loop. Reason: strategies cannot control how often to go through a self-loop; visitors would need to do that.

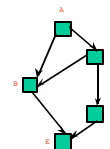
11/6/00

AOO / Demeter

35

## General strategies

```
{
A -> B //neg. constraint 1
B -> E //neg. constraint 2
A -> C //neg. constraint 3
C -> D //neg. constraint 4
D -> E //neg. constraint 5
C -> B //neg. constraint 6
}
```



may even contain loops

11/6/00

AOO / Demeter

36

## General strategies

- Negative constraints
  - either bypassing or
  - only-through
  - complement of each other for entire node or edge set

11/6/00

AOO/Demeter

37

## Constraints

- bypassing
  - A  $\rightarrow$  B bypassing C
    - if  $C \neq A, B$ : delete C and edges incident with C
    - if  $C = A$ : delete edges incoming into A
    - if  $C = B$ : delete edges outgoing from B
    - if  $C = A = B$ : delete edges into and out of A: sit at A

11/6/00

AOO/Demeter

38

## Constraints

- bypassing
  - A  $\rightarrow$  B bypassing  $\rightarrow C, d, D$ 
    - delete edge  $\rightarrow C, d, D$

11/6/00

AOO/Demeter

39

## Constraints

- only-through
  - A  $\rightarrow$  B only-through C
    - delete edges not incident with C

11/6/00

AOO/Demeter

40

## Constraints

- only-through
  - A  $\rightarrow$  B only-through  $\rightarrow C, d, D$ 
    - delete all edges except  $\rightarrow C, d, D$

11/6/00

AOO/Demeter

41

## Metric for structure-shyness

- A strategy  $D$  may be too dependent on a class graph  $G$
- Define a mathematical measure  $Dep(D, G)$  for this dependency
- Goal is to try to minimize  $Dep(D, G)$  of a strategy  $D$  with respect to  $G$  which is the same as maximizing structure-shyness of  $D$

11/6/00

AOO/Demeter

42

## Metric for structure-shyness

- $Size(D)$  = number of strategy edges in  $D$  plus number of distinct class graph node names and class graph edge labels plus number of class graph edges. Each \* occurrence counts as 1.

```

{A -> {G,F}
 G -> H bypassing E}
2 sg edges
5 cg node names
0 cg edge labels
0 cg edges

7 size

```

11/6/00

AOO / Demeter

43

## Metric for structure-shyness

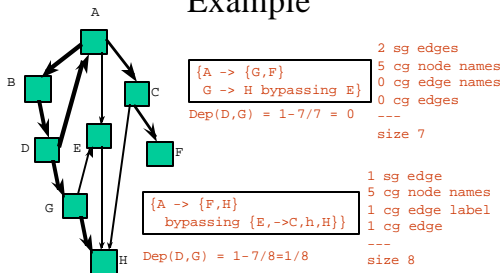
- Define  $Dep_{min}(D,G)$  as a strategy of minimal size among all strategies  $E$  for which  $TG(D,G)=TG(E,G)$  ( $TG$  is traversal graph)
- $Dep(D,G) = 1 - size(Dep_{min}(D,G))/size(D)$
- Ideal:  $Dep(D,G) = 0$ : not always desirable

11/6/00

AOO / Demeter

44

## Example



11/6/00

AOO / Demeter

45

## Finding strategies

- Input: class graph  $G$  and subgraph  $H$
- Output: strategy  $S$  which selects  $H$
- Algorithm (informal):
  - Choose a node basis  $B$  of  $H$  and make the nodes of  $B$  source nodes in the strategy graph. The node basis of a directed graph is a smallest set of nodes from which all other nodes can be reached.

11/6/00

AOO / Demeter

46

## Finding strategies

- Algorithm (continued):
  - Temporarily (for this step only) reverse the edges of  $H$  and choose a node basis of the reversed  $H$  and make the nodes target nodes in the strategy graph.

11/6/00

AOO / Demeter

47

## Finding strategies

- Approximate desired subgraph by single edge strategy (includes star-graphs) without negative constraints:
  - from {source vertex basis} to {target vertex basis}.
- Approximate by positive strategy without negative constraints.
- Find precise strategy by adding negative constraints.

11/6/00

AOO / Demeter

48

### Example

```

{A -> {H,F}
 bypassing -> A,e,E
 bypassing -> G,e,E
 bypassing -> C,e,E
 bypassing -> C,h,H
 bypassing -> A,f,F
}

```

11/6/00 AOO/Demeter 49

### How to find the negative constraints?

- Input: class graph  $G$  and subgraph  $H$
- Output: strategy  $S$  which selects  $H$
- Bypass all edges in  $G$  that
  - have the source in  $H$  but that do not belong to  $H$  and
  - are in the scope of "from source\_vertex\_basis to target\_vertex\_basis"

11/6/00 AOO/Demeter 50

### Not necessarily minimal

- Sometimes we can find an equivalent but shorter set of nodes/edges to bypass.
- Strategy obtained is correct but may not be very structure-shy.
- That is why we use multi-edge strategies.

11/6/00 AOO/Demeter 51

### Example

```

{A -> {H,F}
 bypassing -> A,e,E
 bypassing -> G,e,E
 bypassing -> C,e,E
 bypassing -> C,h,H
 bypassing -> A,f,F
}

{A -> {H,F}
 bypassing E
 bypassing -> C,h,H
 bypassing -> A,f,F
}

```

11/6/00 AOO/Demeter 52

### Robustness and dependency

- If for a strategy  $D$  and class graph  $G$ ,  $Dep(D,G)$  is not 0, it should be justified by robustness concerns.
- Conflicting requirements for a strategy:
  - succinctly describe paths that do exist
    - use minimal info about cd
  - succinctly describe paths that do NOT exist
    - use more than minimal info about cd

11/6/00 AOO/Demeter 53

### Robustness and dependency

- from Company to Money
- from Company via Salary to Money

11/6/00 AOO/Demeter 54

## Summary

- Strategies are good for painting your programs with traversal code
- Strategies allow you to assign roles to objects depending on when you visit them during a traversal
- stay away of strategies through self-loops
- strategies useful for many other things

11/6/00

AOO / Demeter

55

## Universal traversal

- A {void f() to \* (V1)}
- You can also use:  
A {void f() {V1 v1=new V1();  
universal\_trv0(v1);}}

11/6/00

AOO / Demeter

56

## Topic switch

11/6/00

AOO / Demeter

57

## Demeter Method

- Law of Demeter
- Demeter process

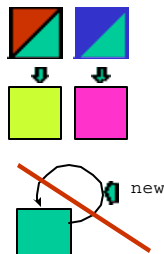
11/6/00

AOO / Demeter

58

## Forms of adaptiveness

- time
  - compile-time ←
  - run-time
- feedback
  - with
  - without ←



11/6/00

AOO / Demeter

59

## Law of Demeter

- Style rule for OOP
- Goals
  - promote good oo programming style
  - minimize coupling between classes; precursor of structure-shyness
  - minimize change propagation
  - facilitate evolution

11/6/00

AOO / Demeter

60

## Formulation (class form)

- Inside method *M* of class *C* one should only call methods attached to (preferred supplier classes)
  - the classes of the immediate subparts (computed or stored) of the current object
  - the classes of the argument objects of *M* (including the class *C* itself)
  - the classes of objects created by *M*

11/6/00

AOO / Demeter

61

## Metric: count number of violations of Law of Demeter

- class version can be easily implemented
- large number of violations is indicator of high maintenance costs
- class version allows situations which are against the spirit of the Law of Demeter

11/6/00

AOO / Demeter

62

## Formulation (object form)

All methods may have only preferred supplier objects.

Expresses the spirit of the basic law and serves as a conceptual guideline for you to approximate.

11/6/00

AOO / Demeter

63

## Preferred supplier objects of a method

- the immediate parts of *this*
- the method's argument objects (which includes *this*)
- the objects that are created directly in the method

11/6/00

AOO / Demeter

64

## Why object form is needed

A = B D E.  
B = D.  
D = E.  
E = .

```
class A {
 void f() {
 this.get_b().get_d().get_e();
 }
}
```

11/6/00

AOO / Demeter

65

## Context switch

11/6/00

AOO / Demeter

66

## Generic OO products

|                  | <i>Analysis</i>        | <i>Design</i>          | <i>Implementation</i> |
|------------------|------------------------|------------------------|-----------------------|
| <i>Behavior</i>  | Use cases              | Collaboration Diagrams | Method Bodies         |
| <i>Structure</i> | Analysis Class Diagram | Design Class Diagram   | Classes               |

11/6/00

AOO / Demeter

67

## Traversal/Visitor OO products

|                  | <i>Analysis</i>        | <i>Design</i>          | <i>Implementation</i> |
|------------------|------------------------|------------------------|-----------------------|
| <i>Behavior</i>  | Use cases              | Traversals<br>Visitors | Method Bodies         |
| <i>Structure</i> | Analysis Class Diagram | Design Class Diagram   | Classes               |

11/6/00

AOO / Demeter

68

## DemeterJ OO products

|                  | <i>Analysis</i>                  | <i>Design</i>                         | <i>Implementation</i> |
|------------------|----------------------------------|---------------------------------------|-----------------------|
| <i>Behavior</i>  | Use cases                        | Visitors<br>Strategies<br>Ad. Methods | Method Bodies         |
| <i>Structure</i> | Annotated Analysis Class Diagram | Annotated Design Class Diagram        | Classes               |

tree objects represented as sentences

11/6/00

AOO / Demeter

69

## Decomposition of OOD

- C = class graph
- G = grammar
- M = method, including adaptive method
- S = strategy
- V = visitor
- $OOD = CD + GD + MD + SD + VD$

11/6/00

AOO / Demeter

70

## Software process

- Development process itself can be described as informal program
- Refine process based on experience
- Adapt process to specific domains
- Could use a process description language

11/6/00

AOO / Demeter

71

## Demeter Method with Visitors

- use case: a typical use of the software to be built.
- Derive from uses cases:
  - analysis class dictionary. Defines vocabulary used in use cases.
  - detailed class dictionary.
  - derive interfaces, traversals, visitors and host/visitor diagrams.

11/6/00

AOO / Demeter

72

## DemeterJ/DJ software process

- For each use case
  - focus on subgraphs of collaborating classes
  - express clustering in terms of strategies and transportation visitors
  - express strategies robustly, focussing on long-term intent

11/6/00

AOO / Demeter

73

## DemeterJ/DJ software process

- Fundamental problem of method design
  - Identify collaborating objects
  - Identify suitable traversals and visitors to collect them
  - Minimize number of methods not calling traversals

11/6/00

AOO / Demeter

74

## DemeterJ/DJ software process

- Fundamental problem of class dictionary design
  - Structural/Behavioral: Arrange the classes so that it is easy to use strategies to collect the collaborating objects needed for behaviors
  - Structural/Grammar: Arrange the classes so that there is a syntax extension which produces natural, English-like descriptions of tree objects

11/6/00

AOO / Demeter

75

## DemeterJ/DJ software process

- Fundamental problem of strategy design
  - Given a group of collaborating classes C, write a strategy which captures the long-term intent behind C

11/6/00

AOO / Demeter

76

## DemeterJ/DJ software process

- Fundamental problem of visitor design
  - What are the classes which do the interesting work for a given task?
  - Decompose into multiple visitors, each one doing a simple task which might be reusable
  - Compose visitors based on the communication needs

11/6/00

AOO / Demeter

77

## DemeterJ/DJ software process

- Fundamental problem of visitor design
  - Separate the core behavioral pieces of an application from their interconnections
  - Two-tiered approach to connection: traversal strategies and class diagrams

11/6/00

AOO / Demeter

78

## Host/visitor diagram

- summarizes important object interactions
- rows consist of host classes
- columns consist of visitor classes
- communication primitives

```
! to host
? from host
!V to visitor V
?V from visitor V
```

11/6/00

AOO/Demeter

79

## Host/visitor diagrams

|           | visitors    |      |         |
|-----------|-------------|------|---------|
|           | Check       | Sum  | Initial |
| before    |             |      | ?S,!I   |
| Container |             |      | !!      |
| after     | ?,?,S,!I,!C |      |         |
| before    |             | ?,!S |         |
| Weight    |             |      |         |

```
! to host
? from host
!V to visitor V
?V from visitor V
```

11/6/00

AOO/Demeter

80

## Managing DemeterJ/DJ projects

- Job categories
  - Visitor designers and implementors
    - Forces: features requested, cd infra structure
  - Class dictionary designers
    - Forces: IO, data structures, cd infra structure req.
  - Feature integrators
    - Forces: use cases, available visitors and class diagrams

11/6/00

AOO/Demeter

81

## Topic switch

11/6/00

AOO/Demeter

82

Read chapter 2 of UML Distilled:  
An Outline Development Process

## Your Project

- Inception
- Elaboration
- Construction consisting of iterations
  - each iteration builds tested and integrated software for a subset of use cases
- Transition

11/6/00

AOO/Demeter

83

## Elaboration

- Risks
  - requirements
  - technological
  - skills
  - political

11/6/00

AOO/Demeter

84

## Elaboration

- Use cases
  - Def: A typical interaction that a user has with the system
  - Provide basis of communication between sponsors and developers
- Domain model (class diagram)
- Design model (class diagram, important strategies and visitors)

11/6/00

AOO / Demeter

85

## Elaboration

- When finished? Takes about 1/5 of total time.
  - Feel comfortable providing estimates
  - Significant risks have been identified
- Planning
  - Assign use cases to iterations, Growth Plan
    - High risk use cases early
  - Commitment schedule

11/6/00

AOO / Demeter

86

## Construction

- Documentation: confine to areas where it helps
- Document patterns in your project
- Use patterns for documentation

11/6/00

AOO / Demeter

87

## Transitions

- Optimization
- More bug fixes
- Time between beta release and final release

11/6/00

AOO / Demeter

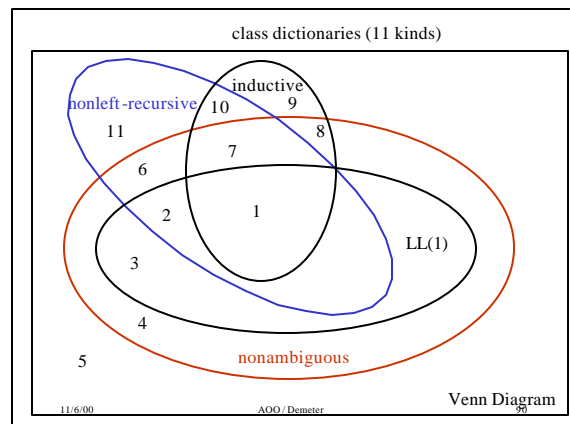
88

## Topic switch

11/6/00

AOO / Demeter

89



## 11 kinds of class dictionaries

- Why 11 and not 16?
  - Four properties: nonambiguous, LL(1), inductive, non-left recursive: 16 sets if independent
  - But: implication relationships
    - LL(1) implies nonambiguous: 12 left
    - LL(1) and inductive imply nonleft-recursive: 11 left

11/6/00

AOO / Demeter

91

## Inductive class dictionaries

- inductiveness already defined for class graphs
  - contains only good recursions: recursions that terminate

```
Car = Motor.
Motor = <belongsTo> Car.
```

bad recursion, objects must be cyclic,  
cannot use for parsing: useless nonterminals

11/6/00

AOO / Demeter

92

## Inductive class dictionaries

- A node  $v$  in a class graph is inductive if there is at least one tree object of class  $v$ .
- A class graph is inductive if all its nodes are inductive.

```
Car = Motor Transmission.
Motor = <belongsTo> Car.
Transmission = .
```

Which nodes are inductive?

11/6/00

AOO / Demeter

93

## Inductiveness style rule to follow

- Maximize the number of classes which are inductive.
- Reasons: cyclic objects
  - cannot be parsed directly from sentences.
  - require visitors to break infinite loops.
  - it is harder to reason about cyclic objects.

11/6/00

AOO / Demeter

94

## Left-recursive class dictionaries

- Bring us back to the same class without consuming input.

```
A : B | C.
B = "b".
C = A.
```

11/6/00

AOO / Demeter

95

## Ambiguous class dictionaries

- cannot distinguish between objects. Print is not injective (one-to-one).

```
Fruit : Apple | Orange.
Apple = "a".
Orange = "a".
```

But: undecidable

11/6/00

AOO / Demeter

96

## LL(1) class dictionaries

- A special kind of nonambiguous class dictionaries. Membership can be checked efficiently.

11/6/00

AOO / Demeter

97

## Style rule

- Ideally, make your class dictionaries LL(1), nonleft-recursive and inductive.

11/6/00

AOO / Demeter

98

## Topic Switch

11/6/00

AOO / Demeter

99

## AP and structural design patterns

- Show how adaptiveness helps to work with structural design patterns
- Focus on Composite and Decorator
- Opportunity to learn two more design patterns

11/6/00

AOO / Demeter

100

## Composite Pattern

- Replace S by Composite(S)

Composite(S) : S | Compound(S).

Compound(S) =

<s> List(Composite(S)).

11/6/00

AOO / Demeter

101

## Decorator Pattern

- Replace S by Decorator(S)

Decorator(S) : S | Decor(S).

Decor(S) :

ScrollDecor(S) | Border(S)

\*common\*

<component> Decorator(S).

11/6/00

AOO / Demeter

102

## Evolution steps for drawing program

- Sketch = <shape> X. Have drawing progr.
  - replace X by Box
  - replace X by Composite(Box) : no change
  - replace X by Decorator(Box)
  - replace X by Composite(Decorator(Box))
  - replace X by Decorator(Composite(Box)):
    - 7 additional classes, need code only for two
- need only code for decorator classes

11/6/00

AOO / Demeter

103

## Program is soft

- Have draw program which works “correctly” in all 5 cases
- The draw program works correctly in infinitely many other class graphs not resulting from applications of Composite and Decorator.
- Focus on essence and not on noise!

11/6/00

AOO / Demeter

104