Demeter/Java notation for strategies

- Notations
  - line graph notation

```
from BookKeeping
  via Taxes via Business
to LineItem
```

  - strategy graph notation

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


Contents

- Traversal strategy notation
- Law of Demeter (revisited) and Demeter Method
- Class dictionary kinds
- AP and structural design patterns

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 }
```


Strategies by example

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

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

Propagation graph: From A to B

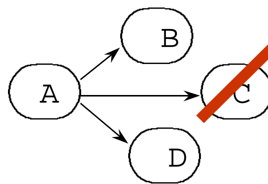
- Reverse all inheritance edges and call them subclass edges.
- Flatten all inheritance by expanding all common parts to concrete subclasses.
- Find all classes reachable from A and color them red including the edges traversed.

Propagation graph: From A to B

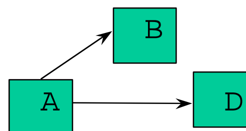
- Find all classes from which B is reachable and color them blue including the edges traversed.
- The group of collaborating classes is the set of classes and edges colored both red and blue.

Propagation graph controls traversal

- object graph



- propagation graph



Propagation graph and bypassing

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

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


Propagation graph and bypassing

- May bypass a set of classes

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

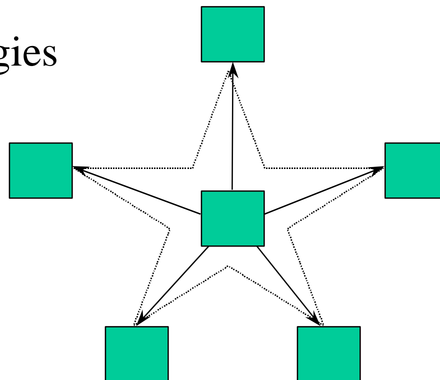
only-through

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

Star-graph strategies

- Multiple targets
- No-pitfall strategies

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



Star-graph strategies

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

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


Approximate meaning of multi-edge strategies

- Decompose strategy graph into single edges
- Propagation graphs for single edge strategies
- Take union of propagation graphs (merge graphs)
- May give wrong result in a few cases for pitfall strategies (Demeter/Java will do it right)

Basic join strategies

- Join two single edge strategies

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

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


Multiple join points

```
from Company
  through {Secretary, Manager}
to Salary

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

Edge-controlled strategies

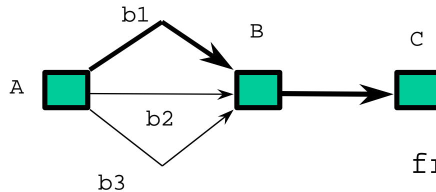
- Class-only strategies are preferred
- They do not reveal details about the part names
- Use whenever possible

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}
```

Needs edge-control




```
from A
   through -> A,b1,B
to C
{A -> B only-through
     -> A,b1,B
 B -> C}
```

```
from A
  bypassing
      {-> A,b2,B ,
       -> A,b3,B}
to C

{A -> C bypassing
   {-> A,b2,B ,
    -> A,b3,B}}
```


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.

Preventing Recursion

```
From Conglomerate
to-stop Company
```

equivalent to

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


simulating to-stop

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

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

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}}
- bypass edges into A and bypass edges out of B
- {A->A bypassing A}

Wysiwg strategies

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

Example: In-laws

Person = Brothers Sisters Status.
Status : Single | Married.
Single = .
Married = <marriedTo> Person.
Brothers ~ {Person}.
Sisters ~ {Person}.

Example: In-laws

```
{Person -> Married                  bypassing Person
  Married -> spouse:Person                  bypassing Person
  spouse:Person -> Brothers                 bypassing Person
  spouse:Person -> Sisters                  bypassing Person
   Brothers -> brothers_in_law:Person              bypassing Person
   Sisters -> sisters_in_law:Person                bypassing Person
}
```

Note: not yet implemented

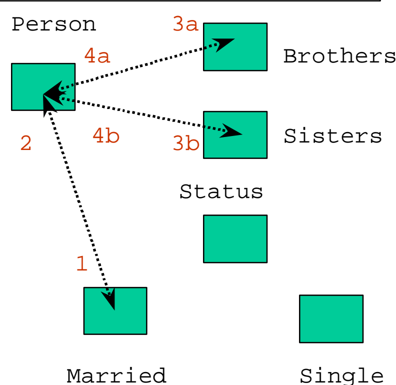
Traversals and naming roles (not implemented)

- Can use strategy graphs to name roles which objects play depending on when we get to them during traversal.

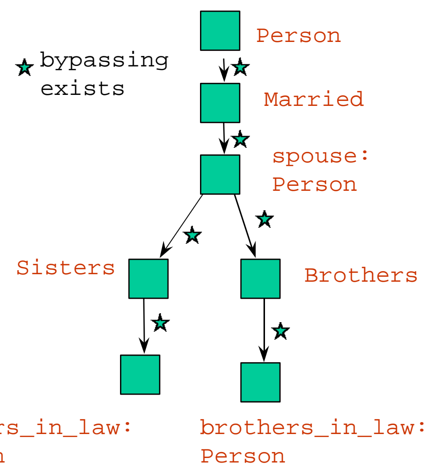
4/20/98 AOP / Demeter 27

Traversal dependent roles

Class graph with super-imposed strategy graph

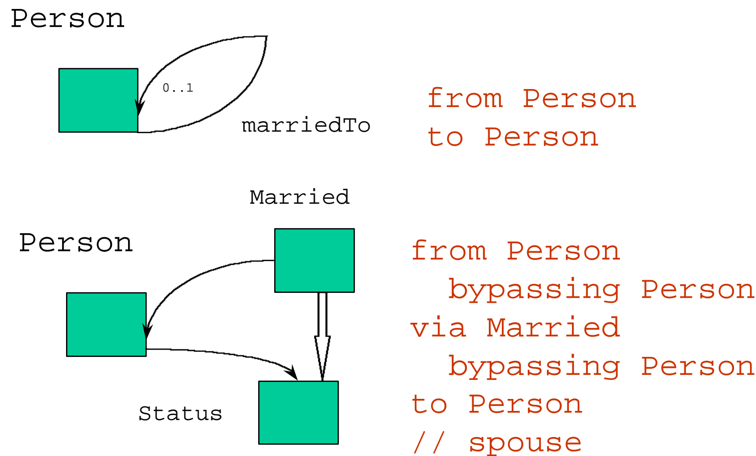


Strategy graph



4/20/98 AOP / Demeter 28

When to avoid strategies?

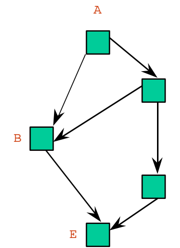


When to avoid strategies

- 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.

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`

General strategies

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

Constraints

- bypassing
  - `A -> B bypassing C`
    - if C ≠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

Constraints

- bypassing
  - `A -> B bypassing ->C,d,D`
    - delete edge `->C,d,D`

Constraints

- only-through
  - `A -> B only-through C`
    - delete edges not incident with C

Constraints

- only-through
  - `A -> B only-through ->C,d,D`
    - delete all edges except `->C,d,D`

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*

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.

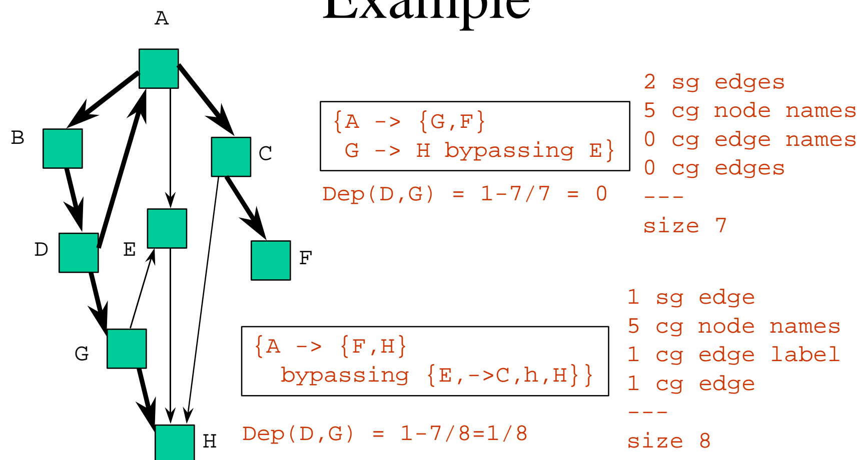
```
{A -> {G,F}
 G -> H bypassing E}
```

```
2 sg edges
5 cg node names
0 cg edge labels
0 cg edges
---
7 size
```

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)
- $Dep(D,G) = 1 - size(Dep_{min}(D,G))/size(D)$

Example



Finding strategies

- Input: class graph *G* and subgraph *H*
- Output: strategy *S* which selects *H*
- Algorithm (informal):
  - Choose a node basis of *H* and make the nodes 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.

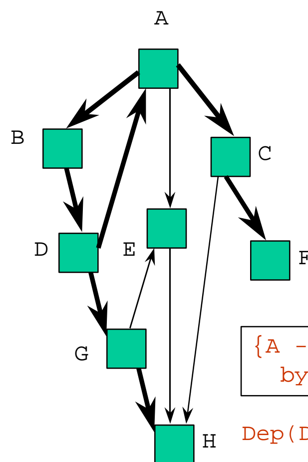
Finding strategies

- 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.

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 general strategy without negative constraints.
- Find precise strategy by adding negative constraints.

Example




```
{A -> {G,F}
 G -> H bypassing E}
Dep(D,G) = 1-7/7 = 0
```

```
2 sg edges
5 cg node names
0 cg edge names
0 cg edges
---
size 7
```

```
{A -> {F,H}
  bypassing {E,->C,h,H}}
Dep(D,G) = 1-7/8=1/8
```

```
1 sg edge
5 cg node names
1 cg edge label
1 cg edge
---
size 8
```

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

Robustness and dependency

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

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 of away of strategies through self-loops
- strategies useful for many other things

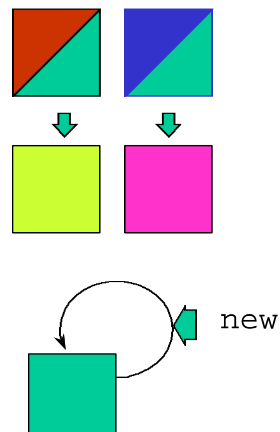
Topic switch

Demeter Method

- Law of Demeter
- Demeter process

Forms of adaptiveness

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



Law of Demeter

- Style rule for OOP
- Goals
  - promote good oo programming style
  - minimize coupling between classes
  - minimize change propagation
  - facilitate evolution

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`

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

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.

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

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();
  }
}
```

Context switch to Demeter/Method

Generic OO products

	Analysis	*Design*	*Implementation*
Behavior	Use cases	Collaboration Diagrams	Method Bodies
Structure	Analysis Class Diagram	Design Class Diagram	Classes

Traversal/Visitor OO products

	Analysis	*Design*	*Implementation*
Behavior	Use cases	Traversals Visitors	Method Bodies
Structure	Analysis Class Diagram	Design Class Diagram	Classes

Demeter/Java OO products

	Analysis	*Design*	*Implementation*
Behavior	Use cases	Visitors Strategies Ad. Methods	Method Bodies
Structure	Annotated Analysis Class Diagram	Annotated Design Class Diagram	Classes

tree objects represented as sentences

Decomposition of OOD

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

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

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.

Demeter/Java 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

Demeter/Java 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

Demeter/Java 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

Demeter/Java 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

Demeter/Java 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

Demeter/Java 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

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
```

Host/visitor diagrams

visitors

hosts

	Check	Sum	Initial
before Container *after*	?,?S,?I,!C		?S,!I !I
before Weight		?,!S	

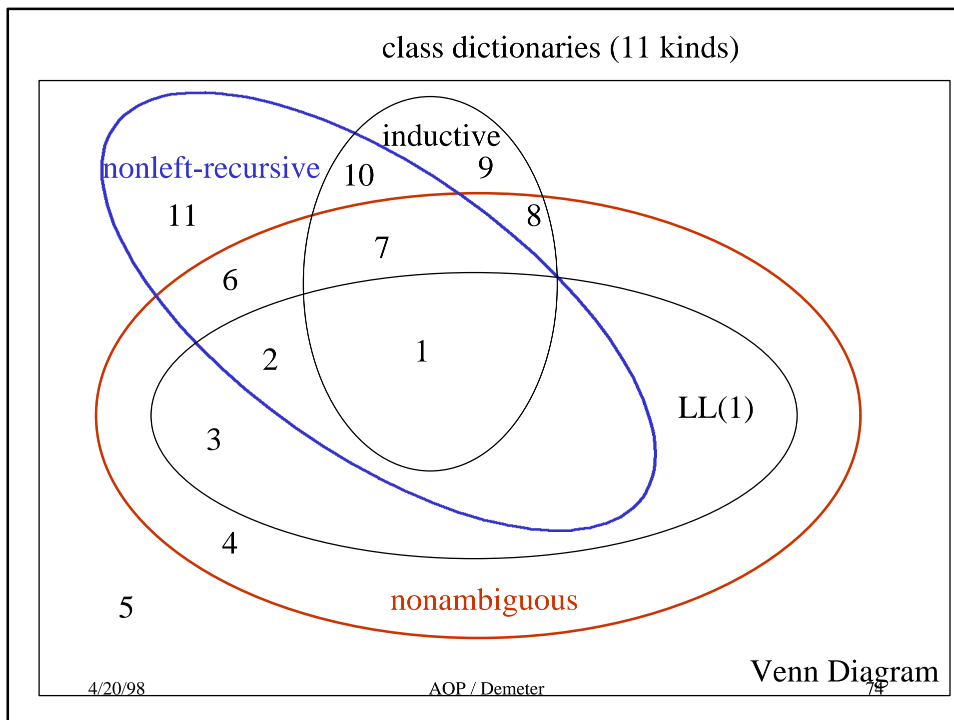
```
!   to host
?   from host
!V  to visitor V
?V  from visitor V
```


Managing Demeter/Java 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

Topic switch

class dictionaries (11 kinds)



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

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

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?

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.

Left-recursive class dictionaries

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

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


Ambiguous class dictionaries

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

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

But: undecidable

LL(1) class dictionaries

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

Style rule

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

Topic Switch

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

Composite Pattern

- Replace S by Composite(S)

```
Composite(S) : S | Compound(S).
Compound(S) =
  <s> List(Composite(S)).
```


Decorator Pattern

- Replace S by Decorator(S)

```
Decorator(S) : S | Decor(S).
Decor(S) :
 ScrollDecor(S) | Border(S)
  *common*
      <component> Decorator(S).
```

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

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!