A Short Introduction to Adaptive Programming (AP) with collaborations and adapters

Northeastern Team

Problem addressed

- “To what extent is it possible to construct useful programs without knowing exactly what data types are involved?”
- First sentence of paper: “Generic functional programming with types and relations” by Richard Bird, Oege De Moor, Paul Hoogendijk, J. Functional Programming, 6(11): 1-28, January 1996.

Approaches

- In paper mentioned:
  - Generalize specific algorithms to make them adaptive for different data types: pattern matching, maximum segment sum.
- In AP:
  - Provide programming tools for writing adaptive algorithms in general. Current focus on Java: DJ.

Example: The Publisher-Subscriber Protocol

- Have two collaborating participants: Publisher and Subscriber
- All subscribers are reachable from publisher. We call this a “complete” version of the pattern (no attach or detach).
- Describe a specific version of the pattern.

Publisher-Subscriber-complete

```
collaboration PublisherSubscriberProtocol  {
  participant Publisher {
       in TraversalGraph subscribers();
       in-out void changeOp(*) {
          expected(); List subscrs = subscribers().asList(this);
          for (int i = 0; i < subscrs.size(); i++)
            {((Subscriber)subscrs.elementAt(i)).
              newUpdate(this);}}
  participant Subscriber {
      in void subUpdate(Publisher publ);
      protected Publisher publ;
      out void newUpdate(Publisher aPubl) {
         publ = aPubl;
         subUpdate(publ);}
      }
   }
}
```

Meta variable: bold
Keywords: underscored

Classes for deployment

```
Class Point {
  void set(…)
}

class InterestedInPointChange {
  void public notifyChange() {
    System.out.println("CHANGE ...");
  }
}
```

Deployment 1

```
adapter PubSubConn1 {
  Point is Publisher with
     { changeOp = set*;}
     { TraversalGraph subscribers() {
        ClassGraph cg = new ClassGraph(true, false);
        String s = "from Point via B bypassing C to
                    InterestedInPointChange";
        return new TraversalGraph(s,cg); }
  InterestedInPointChange is Subscriber with {
      void subUpdate(Publisher publ) {
         notifyChange();
         System.out.println("on Point object " +
           ((Point) publ).toString());
       }
   }
 }
```

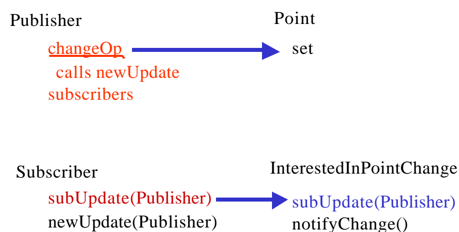

Adaptive Deployment Qosketeer

```
adapter PubSubConn1 {
  Point is Publisher with
     { changeOp = set*;}
     { TraversalGraph subscribers() {
        ClassGraph cg = new ClassGraph(true, false);
        String s = // computed from a System property
        return new TraversalGraph(s,cg); }
  InterestedInPointChange is Subscriber with {
      void subUpdate(Publisher publ) {
         notifyChange();
         System.out.println("on Point object " +
           ((Point) publ).toString());
       }
   }
 }
```


Red: expected replaced

Blue: from adapter

Deployment 1



Deployment 2

```
adapter PubSubConn2 {
  TicTacToe is Publisher with {
    changeOp = {startGame, newPlayer, putMark,
     endGame}
    TraversalGraph subscribers() {
        ClassGraph cg = new ClassGraph(true, false);
        String s = "from TicTacToe via B bypassing C to
                    {BoardDisplay, StatusDisplay}";
        return new TraversalGraph(s,cg); }
};
  {BoardDisplay, StatusDisplay} is Subscriber with {
       void subUpdate(Publisher publ) {
          setGame((Game) publ);
          repaint();
        }
     };
 }
```


Overview

- DJ introduction
- AspectJ and DJ
- Aspect-oriented Programming in pure Java using the DJ library

AP

- Late binding of data structures
- Programming without accidental data structure details yet handling all those details on demand without program change

Concepts needed (DJ classes)

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

Adaptive Programming

Bold names refer to DJ classes.

Strategy

is use-case based abstraction of

ClassGraph

defines family of

ObjectGraph

Adaptive Programming

Strategy

defines traversals of

ObjectGraph

plus Strategy defines

ObjectGraphSlice

Adaptive Programming

Strategy

guides and informs

Visitor

Software Design and Development with DJ (very brief)

- Functional decomposition into generic behavior
  - Decomposition into methods
  - Decomposition into formal traversal graphs
  - Decomposition into visitors
- Adaptation of generic behavior
  - Identify class graph
  - Identify traversal strategies

AspectJ	DJ
• Abstract pointcut – set of execution points – where to watch • advice – what to do • Concrete pointcut – set notation using regular expressions	• Abstract object slice – set of entry/exit points – where to go • visitor – what to do • Actual object slice – path set notation using traversal strategies

AspectJ: Observer Pattern: Abstract Pointcut

```
public abstract aspect Subject { ...
   abstract pointcut stateChanges();
   after(): stateChanges() {
      for (int i = 0; i < observers.size(); i++)
         { ((Observer)observers.elementAt(i)).update(); } }
…}
```


AspectJ: Observer Pattern: Concrete Pointcut

```
aspect ColoredNumberAsSubject extends Subject
  of eachobject(instanceof(ColoredNumber)) {
    pointcut stateChanges():
      (receptions(void setValue(..)) ||
       receptions(void setColor(..))); …}
```


DJ: Counting Pattern: Abstract Pointcut

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


DJ: Counting Pattern: Concrete Pointcut

```
// Prepare the traversal for the current class graph
ClassGraph classGraph = new ClassGraph();
TraversalGraph WPTraversal = new TraversalGraph
  ("from BusRoute via BusStop to Person", classGraph);

int r = aBusRoute.countPersons(WPTraversal);
```

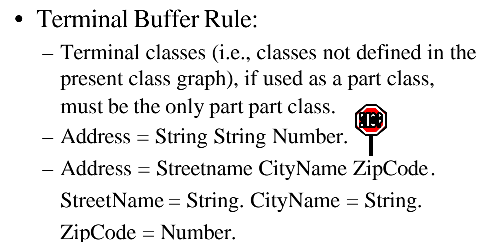

Example : Count Aspect Pattern

```
collaboration Counting {
   participant Source {
        import TraversalGraph getT();
     public int count (){ // traversal/visitor weaving
          getT().traverse(this, new Visitor(){ int r;
             public void before(Target host){ r++; }
             public void start() { r = 0;} …) } }
   participant Target {}
}
```

Base:
Meta variable: bold
Keywords: underscore

DJ Example: TBR

- Terminal Buffer Rule:
  - Terminal classes (i.e., classes not defined in the present class graph), if used as a part class, must be the only part part class.
  - Address = String String Number.



  - Address = Streetname CityName ZipCode. StreetName = String. CityName = String. ZipCode = Number.

Terminal Buffer Rule (TBR)

```
//class Cd_graph
public void TBRchecker(
  TraversalGraph definedClassNamesT,
  TraversalGraph allPartsT,
  TraversalGraph fetchIdentT) {
    // definedClassNamesT defines
    // the part of the object graph
    // that is relevant for finding
    // all defined classes.
    // allPartsT defines
    // the part of the object graph
    // that is relevant for checking
    // the TerminalBufferRule
```


TBR: generic behavior

```
// find defined classes
    DefinedClassVisitor v1 =
      new DefinedClassVisitor
        (fetchIdentT);
    Vector definedClasses =
      (Vector) definedClassNamesT.
         traverse( this, v1);
    // check for violations
    TBRVisitor v2 =
      new TBRVisitor(definedClasses);
    allPartsT.traverse(this, v2);
}
```

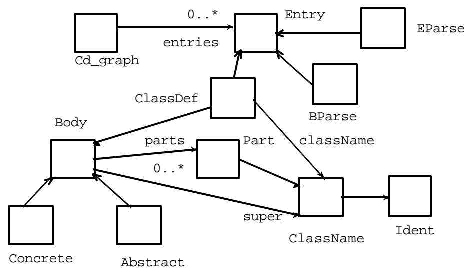

TBR: DefinedClassVisitor

```
public class DefinedClassVisitor
    extends Visitor { ...
  private Vector vNonTerminals = new Vector();
  public void before(Adj o) {
    Ident idCurrentAdj = fetchIdentT.fetch(o);
    vNonTerminals.addElement(idCurrentAdj);}
  public Object getReturnValue() {
    return vNonTerminals;
  }
}
```


TBR: Adaptation to a concrete Structure 1

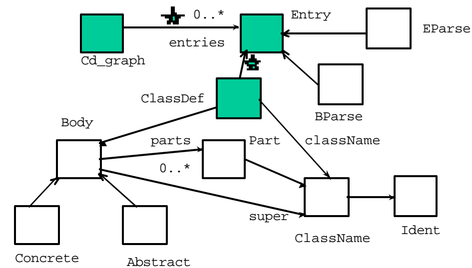
```
ClassGraph cg = new ClassGraph();//reflection
TraversalGraph tg1 = new TraversalGraph(
  "from Cd_graph to Adj", cg);
// The purpose of traversal tg2
// is to visit all parts of all classes
TraversalGraph tg2 = new TraversalGraph(
  "from Cd_graph via Construct to Vertex", cg);
TraversalGraph tg3 = …
  "from Adj through Vertex to Ident" …;
CdGraph cdGraph = new CdGraph(...);
cdGraph.TBRchecker(tg1,tg2,tg3);
```


Example 2: Cd_graph



Example 2: Adaptation

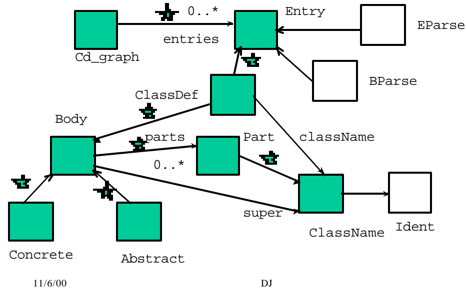
```
definedClassNamesT "from Cd_graph to ClassDef"
```



Example 2: Adaptation

allPartsT

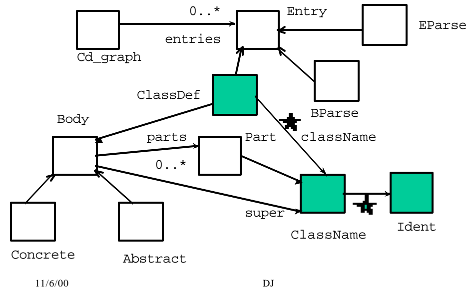
"from Cd_graph through Part to ClassName"



Example 2: Adaptation

fetchIdentT

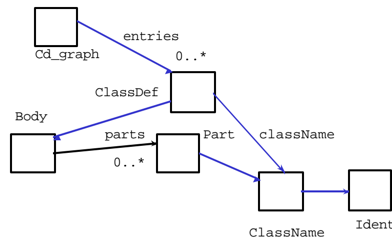
"from ClassDef through ->className to Ident"



TBR: Adaptation to a concrete Structure 2

```
ClassGraph cg = new ClassGraph();//reflection
TraversalGraph tg1 = new TraversalGraph(
  "from Cd_graph to ClassDef", cg);
// The purpose of traversal tg2
// is to visit all parts of all classes
TraversalGraph tg2 = new TraversalGraph(
 "from Cd_graph through Part to ClassName", cg);
TraversalGraph tg3 = …
 "from ClassDef through ->className to Ident" …;
CdGraph cdGraph = new CdGraph(...);
cdGraph.TBRchecker(tg1,tg2,tg3);
```


Participant Graph: connections are elastic

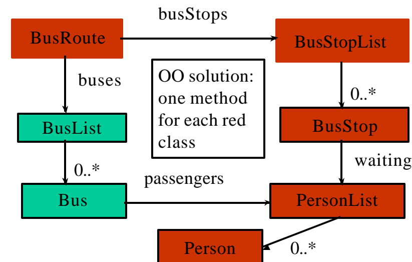


AP history

- Programming with partial data structures = propagation patterns
- Programming with participant graphs
- Programming with object slices
  - partial data structures = all the constraints imposed by visitors

Collaborating Classes

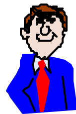
find all persons waiting at any bus stop on a bus route

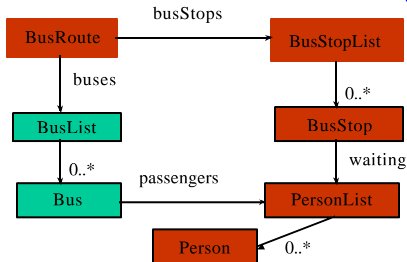


find all persons waiting at any bus stop on a bus route

Traversal Strategy

from BusRoute through BusStop to Person

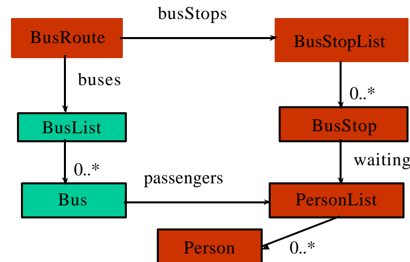




find all persons waiting at any bus stop on a bus route

Traversal Strategy

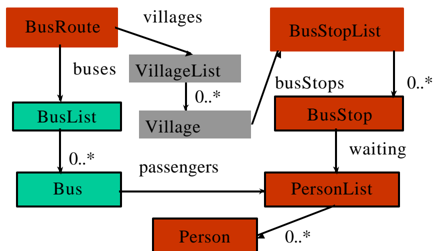
from BusRoute through BusStop to Person



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

Robustness of Strategy

from BusRoute through BusStop to Person



Writing Adaptive Programs with Strategies (DJ=pure Java)

String WPStrategy="from BusRoute through BusStop to Person"

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


Writing Adaptive Programs with Strategies (DJ=pure Java)

String WPStrategy="from BusRoute through BusStop to Person"

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

int r = aBusRoute.countPersons(WPTraversal);
```


Writing Adaptive Programs with Strategies (DJ=pure Java)

String WPStrategy="from BusRoute through BusStop to Person"

```
class BusRoute {
  int countPersons(TraversalGraph WP) {
    Integer result = (Integer)
     WP.traverse(this, new Visitor(){...});
  return result.intValue();}
}
```

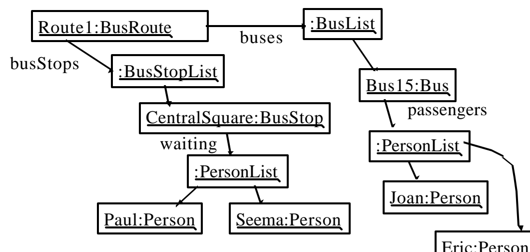
WP.traverse(this,visitor) <===>

```
ObjectGraph objectGraph =
  new ObjectGraph(this, classGraph);
ObjectGraphSlice objectGraphSlice =
  new ObjectGraphSlice(objectGraph, WP);
objectGraphSlice.traverse(visitor);
```

ObjectGraph

```
BusRoute (
  <busStops> BusStopList {
    BusStop(
      <waiting> PersonList {
        Person()
        Person()}))}
  <buses> BusList{
    Bus(
      <passengers> PersonList {
        Person()
        Person()}))})
```

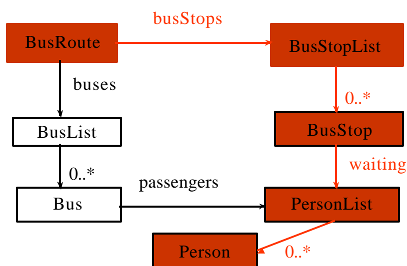

ObjectGraph: in UML notation



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

TraversalGraph

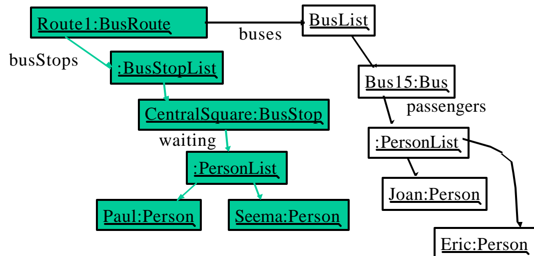
from BusRoute through BusStop to Person



ObjectGraphSlice

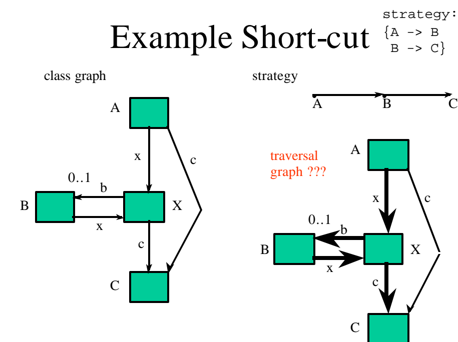
```
BusRoute (
  <busStops> BusStopList {
    BusStop(
      <waiting> PersonList {
        Person()
        Person()}))}
  <buses> BusList{
    Bus(
      <passengers> PersonList {
        Person()
        Person()}))})
```

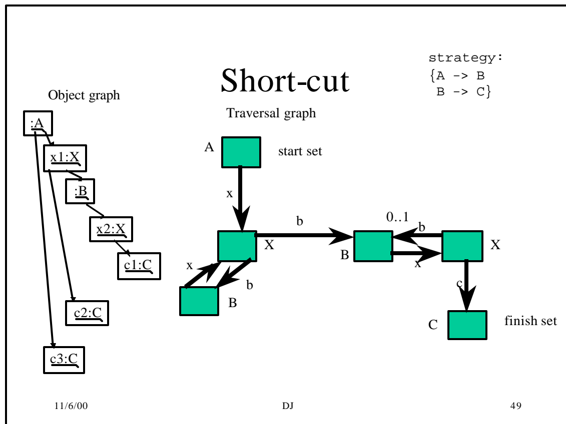

ObjectGraphSlice



Example Short-cut

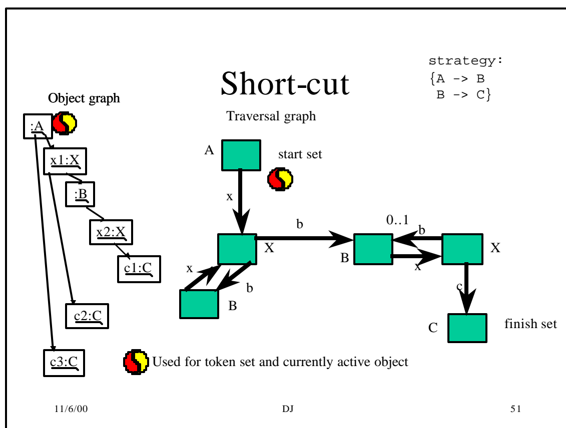
```
strategy:
{A -> B
 B -> C}
```

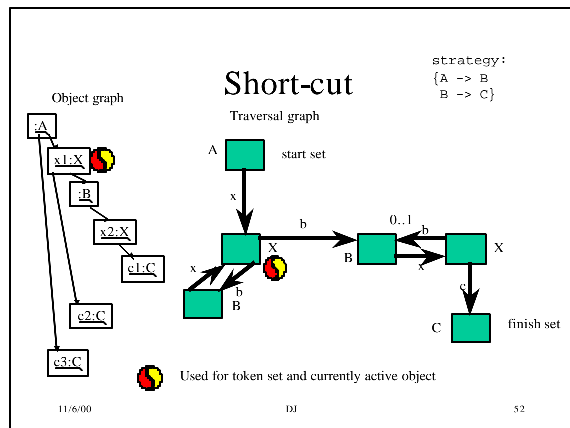


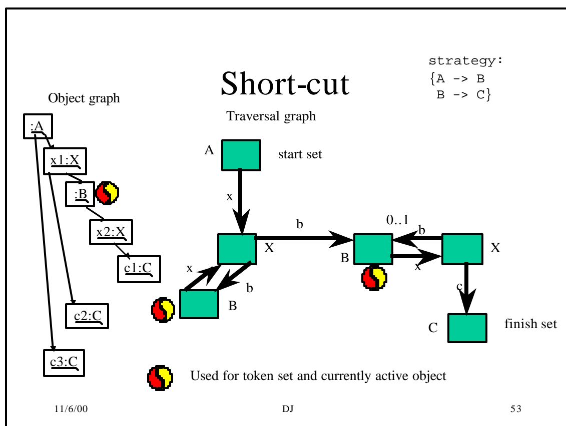


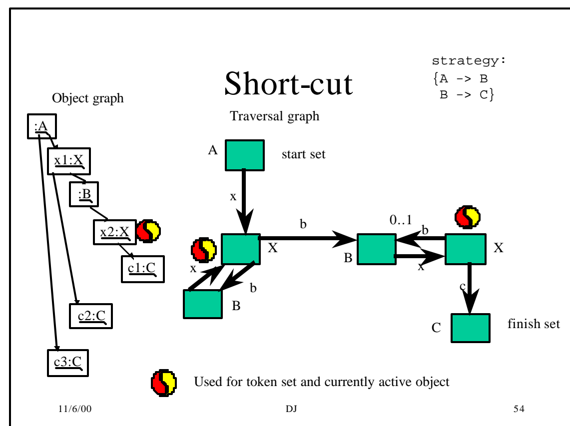
Short-cut

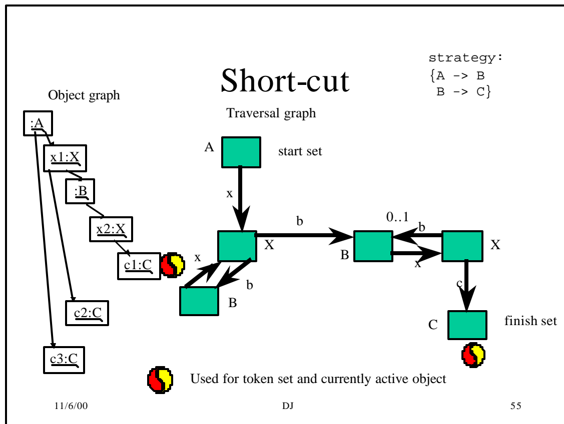
- ObjectGraphSlice
  - static view
    - ObjectGraph, TraversalGraph, start nodes, finish nodes
  - dynamic view
    - when `traverse` method gets called
    - traversal history: node sequence of nodes in object graph
    - subgraph of object graph (but only part of the story)
    - visualize it by a movie of `traverse`

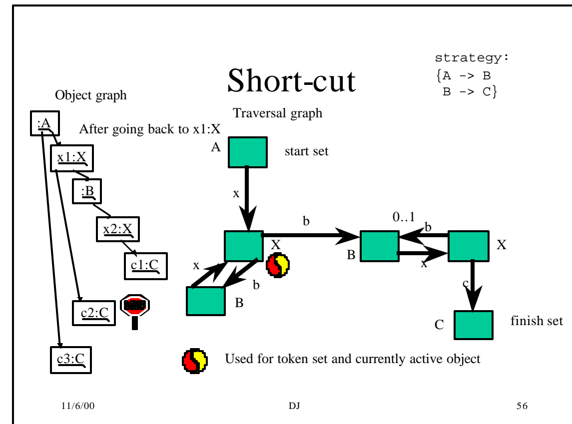


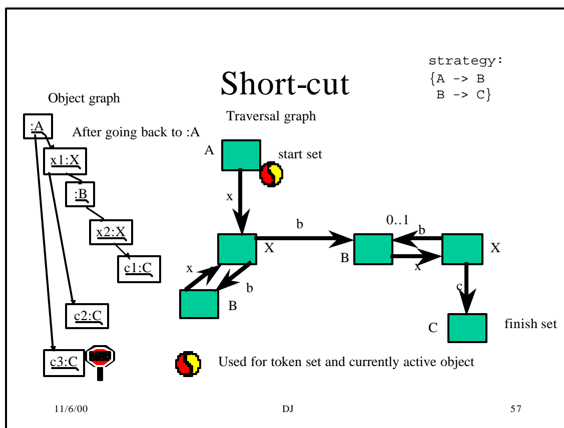


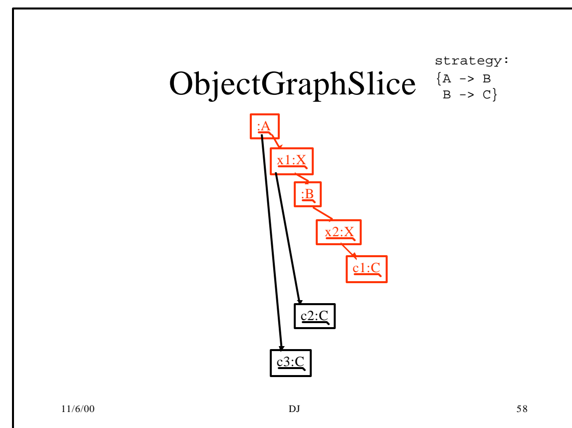












Applications of Traversal Strategies

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

Example

- For data member access:
- new TraversalGraph("from A via B to C", Main.cg).fetch(host);

Graphs and paths

- Directed graph: *(V,E)*, *V* is a set of nodes, *E* Í *V* ´ *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* Í *V* ´ *L* ´ *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 *e*.

Graphs and paths

- Given a directed labeled graph: *(V,E,L)*, a node-path is a sequence $p = <v_0 v_1 \ldots v_n>$ 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 $<v_0\ l_1\ v_1\ l_2 \ldots l_n\ v_n>$, where $<v_0 \ldots v_n>$ is a node-path and $(v_{i-1}, l_i, v_i) \in E$.

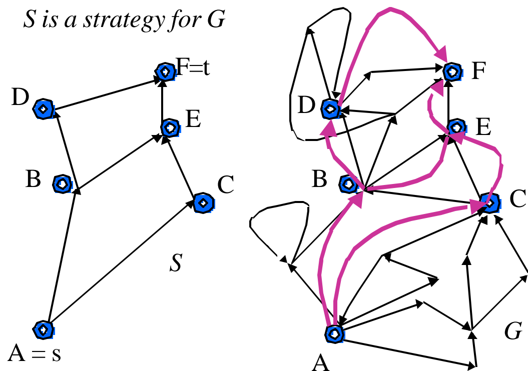
Graphs and paths

- In addition, we allow node-paths and paths of the form $<v_0>$ (called trivial).
- 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.

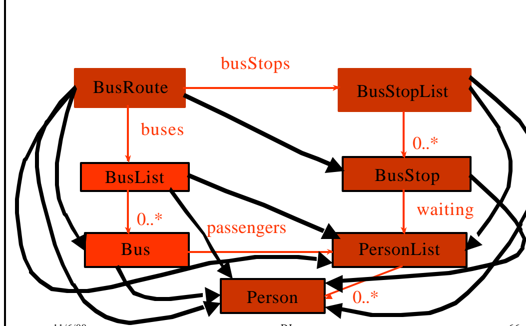
Strategy definition: embedded, positive strategies

- Given a graph G, a strategy graph S of G is any subgraph of the transitive closure of G.
- The transitive closure of G=(V,E) is the graph G*=(V,E*), where E*={(v,w): there is a path from vertex v to vertex w in G}.

S is a strategy for G



Transitive Closure



Strategy graph and base graph are directed graphs

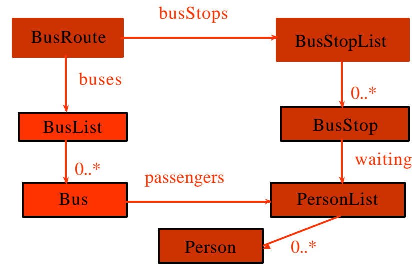
Key concepts

- Strategy graph *S* with source *s* and target *t* of a base graph *G*. *Nodes(S)* subset *Nodes(G)* (Embedded strategy graph).
- A path *p* is an ***expansion*** of path *p'* if *p'* can be obtained by deleting some elements from *p*.
- *S* defines ***path set*** in *G* as follows: $PathSet_{st}(S,G)$ is the set of all *s-t* paths in *G* that are expansions of any *s-t* path in *S*.

`S = from BusRoute through BusStop to Person`

Expansion

(BR BSL BS PL P) is an expansion of (BR BS P)

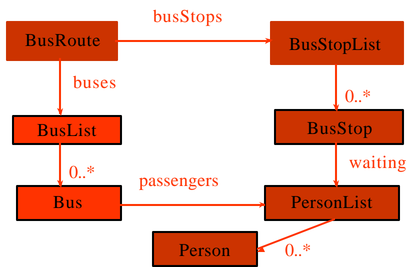


`S = from BusRoute to Person`

PathSet

BR → P

$PathSet_{BusRoute,Person}$ (S,G)={(BR BL B PL P),(BR BSL BS PL P)}

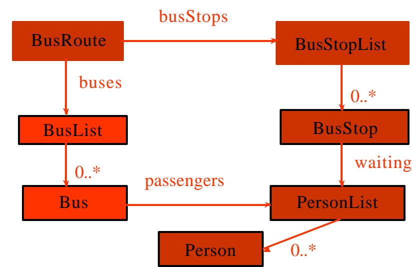


`S = from BusRoute through BusStop to Person`

PathSet

BR → BS → P

$PathSet_{BusRoute,Person}$ (S,G)={(BR BSL BS PL P)}

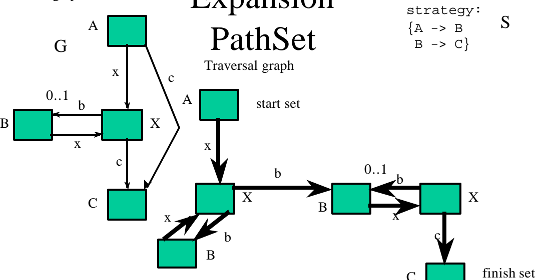


class graph

Expansion PathSet

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

S



(A X B X C) is an expansion of (A B C)

$PathSet_{A,C}$(S,G) = {(A X B X (B X)* C)}

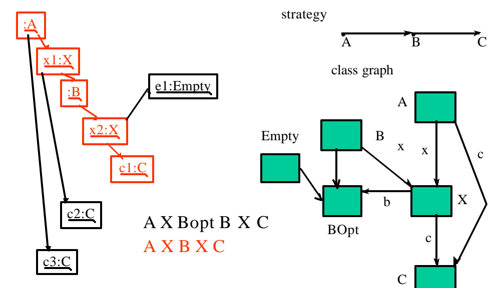
Correspondence

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

Object graph

strategy

A → B → C



From Path Set to Object Slices

- Effect of a strategy
  - at class graph level: defines a path set that in general cannot be described by a subgraph of the class graph.
  - at object graph level: defines a subgraph of the object graph, called an object graph slice

From Path Set to Object Slice: Assume class graph is flat

- An edge e of an object graph belongs to the object graph slice determined by strategy S=(SS,s,t), if the path from s to e is an alternation reduction of a path in the path set of the class graph and S.
- An alternation reduction is a path with alternation nodes and edges and inheritance edges deleted.

DJ

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

Integration of Generic and Adaptive Programming

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

Sample DJ code

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


Methods provided by DJ

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

Traverse method: excellent support for Visitor Pattern

```
// class ClassGraph
Object traverse(Object o,
  Strategy s, Visitor v);
```

traverse navigates through Object o following traversal specification s and executing the before and after methods in visitor v

ClassGraph is computed using introspection

Fetch Method

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

Gather Method

- Returns a list of objects.
- Object ClassGraph.gather(Object o, String s)
  - List ks = classGraph.gather(a,"from A to K"); returns a list of K-objects.

Using DJ

- traverse(…) returns the v[0] return value. Make sure the casting is done right, otherwise you get a run-time error. If "`public Object` getReturnValue()" returns an Integer and traverse(…) casts it to a Real: casting error at run-time.
- Make sure all entries of Visitor[] array are non-null.

Using multiple visitors

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

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


DJ binaryconstruction operations

	cg	s	tg	o	og	ogs
cg	*	tg,cg	*	og	*	*
s		*	*	*	ogs	*
tg			*	*	ogs	*
o				*	*	*
og					*	*
ogs						*

Who has traverse, fetch, gather? (number of arguments of traverse)

	cg(3)	s	tg(2)	o	og(2)	ogs(1)
cg	*	tg,cg	*	og	*	*
s		*	*	*	ogs	*
tg			*	*	ogs	*
o				*	*	*
og					*	*
ogs						*

Methods returning an ObjectGraphSlice

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

Blue: constructors

Traverse method arguments

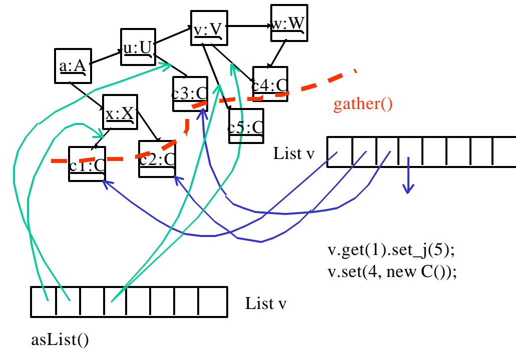
- ClassGraph
  - Object, Strategy, Visitor
- TraversalGraph
  - Object, Visitor
- ObjectGraph
  - Strategy, Visitor
- ObjectGraphSlice
  - Visitor

Traverse method arguments. Where is collection framework used?

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

Where is collection framework used?

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



Interfaces

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

Why is asList useful?

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

Why is asList useful?

- `from Application to X:` want to change all X-objects to D-objects.

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

Sketch:
from Application to E:
aE.set_x(new D());
aE.get_d().set_x(new D());
from E to D: update list elements

from Application to X: set(new D())

Why is asList useful?

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

A = B C.

C = List(B).

Some more theory

- So far:
  - strategy *S* for a class graph *G: S* is subgraph of the transitive closure of *G*.
  - *S* defines ***path set*** in *G* as follows: $PathSet_{st}(S,G)$ is the set of all *s-t* paths in *G* that are expansions of any *s-t* path in *S*.
  - A path *p* is an ***expansion*** of path *p'* if *p'* can be obtained by deleting some elements from *p*.

What is a path?

- If we only have concrete classes: no problem.
- Use the standard graph theoretic definition.

A Path

- (A,B,E)

E
A
B

Traversals with Abstract Classes

- from A to E
- from A to B
- from A to C
- from A to D

E
A
B
C
D

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

E
A
B
C
D

Path concept

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

Path concept

EI: inheritance or is-a edges
EA: subclass or alternation edges
EC: construction or has-a edges

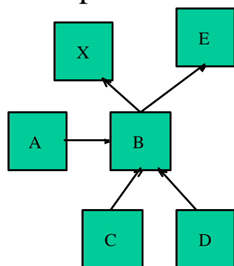
- Path from A to B:
  - EI implies EA in opposite direction
  - (EC | EA | EI)* but not EA followed by EI
  - ((EI* EC) | EA)* EI*
  - checking the eight allowed edge pairs: (EI,EC),(EI,EI),(EC,EC),(EA,EI),(EC,EI), (EA,EA),(EC,EA),(EA,EC)

Equivalence of Regular Expressions

- EA* (EI* EC EA*)* EI*
- ((EI* EC) | EA)* EI*

Problem with path concept of DJ

- from A to X
- from X to C
- from A via X to C



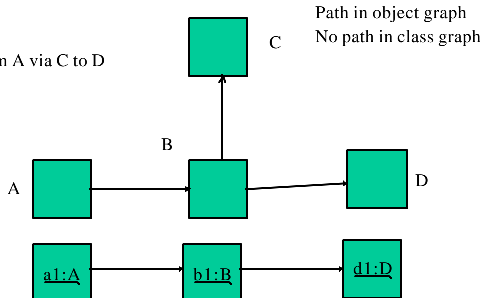
Path concept in flat class graphs

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

Legal Path

From A via C to D

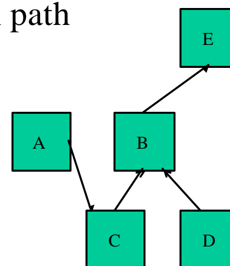
Path in object graph
No path in class graph



Legal path

From A to D

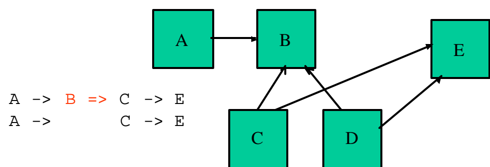
No path in object graph
No path in class graph



Requirements: for flat class graphs

- Paths in class graph turn into paths in object graph by deleting alternation edges and nodes and inheritance edges (Def. of corresponding path).

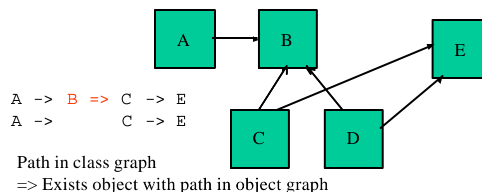
```
A -> B => C -> E
A ->      C -> E
```



Requirements: for flat class graphs

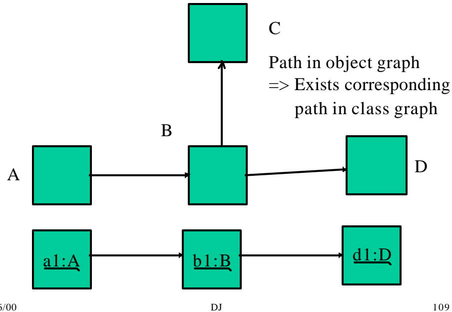
- If there is a path in the class graph then there is a legal object of the class graph that has the same corresponding path.

```
A -> B => C -> E
A ->      C -> E
```



Path in class graph
=> Exists object with path in object graph

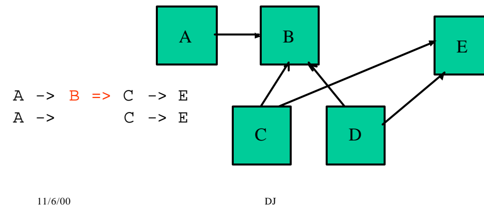
Requirements: for flat class graphs



Requirements: for flat class graphs

- For a class graph G and legal object O, if there is a path p in the object graph then there is a path P in the class graph such that p corresponds to P.

```
A -> B => C -> E
A ->      C -> E
```

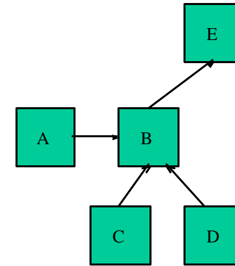


More on semantics of DJ with abstract classes

- What is the meaning of a visitor on an abstract class?

Traversals to Abstract Classes

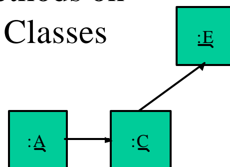
- `Class graph`



visitor:
before (B){p("b");}
before (C){p("c");}

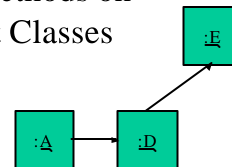
Visitor Methods on Abstract Classes

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



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");}

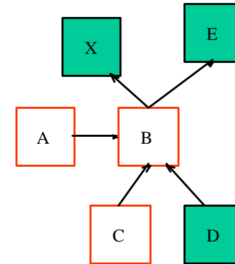


Visitor Rule

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

Traversals to Abstract Classes

- `From A to C`



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

Alternative Visitor Rule: not what programmers want

- When an object of class X is visited, all visitors of ancestor classes of X ~~*and in the path set of the current traversal*~~ will be active.
- The before visitors in the downward order and the after visitors in the upward order.

Guidelines

```
IF you use the combination of the following pairs and triples
   for multiple traversals, fetch or gather, introduce the
   following computation saving objects:
(cg,s,o)->ogs
(cg,s)->tg
(cg,o)->og
(tg,o)->ogs

cg      class graph
s       strategy
tg      traversal graph
o       object
og      object graph
ogs     object graph slice
v       visitor
```

Abbreviations

In principle can express programs only with ClassGraph and Strategy and Visitor:
cg.traverse(o,s,v);
cg.fetch(o,s);
cg.gather(o,s);
cg.asList(o,s);

DJ unary construction operations

- Class graph from Strategy
  - ClassGraph(Strategy): make a class graph with just the classes and edges in traversal graph determined by strategy. Interpret path set as a graph. Maintain several class graphs cg1, cg2, cg3 that are suitable for expressing what you need.
- Class graph from all classes in package

ClassGraph construction

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

Dynamic features of DJ ClassGraph construction

- When a class is defined dynamically from a byte array (e.g., from network) ClassGraph.addClass(Class cl) has to be called explicitly. Class cl is returned by class loader.
- ClassGraph() constructor examines class file names in default package and uses them to create class graph.

Dynamic features of DJ ClassGraph construction

- ClassGraph.addPackage(String p)
  - adds the classes of package p to the class graph. The package is searched for in the CLASSPATH. How can we control (f,m) options?
- Java has no reflection for packages. Motivates above solution.

Adding Nodes and Edges to ClassGraph

- addClass(Class cl)
  - add cl and all its members to the class graph, if it hasn't already been added.
- addClass(Class cl, boolean aF, boolean aM)
  - add cl to the class graph. If aF, add all its non-static fields as construction edges. If aM, add all its non-static non-void methods with no arguments as derived construction edges.

Adding Nodes and Edges to ClassGraph

- Part addConstructionEdge(Field f)
  - add f as a construction edge.
- Part addConstructionEdge(Method m)
  - add a no-args method as a construction edge.
- addConstructionEdge may have in addition a String argument called source ???
- And also a Class argument called target ???

Add other repetition edges

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

Design and Implementation

- Until summer 1999: Josh Marshall
- Since then: Doug Orleans
- Available on the Web from DJ home page
- Quite complex
- Viewing an object as a list is done through coroutines to simulate continuation

Problem with DJ

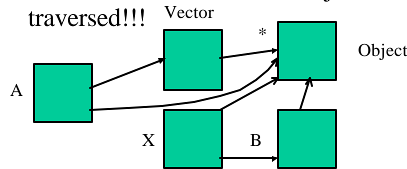
- What is coming is not about a problem of DJ but about a problem with Java: the lack of parameterized classes.
- The lack of parameterized classes forces the use of class Object which, as the mother of all classes, is too well connected.
- This leads to unnecessary traversals and traversal graphs that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

Lack of parameterized classes in Java makes DJ harder to use

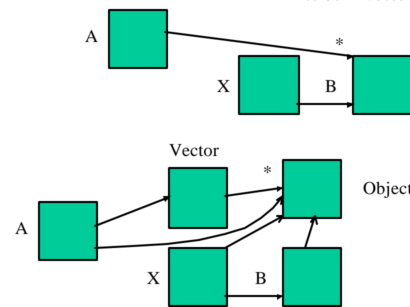
- Consider the traversal: from A to B
- Let's assume that in the class graph between A and B there is a Java collection class. The intent is: A = List(B) which we cannot express in Java. Instead we have: A = Vector(Object). Object : A | B. Let's assume we also have a class X=B.

Lack of parameterized classes in Java makes DJ harder to use

- We have: A = Vector(Object). Object : A | B | X. X = B.
- If the vector contains an X object it will be traversed!!!



No X-object is allowed to be in vector



Moral of the story

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

Moral of the story

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

Size of traversal graph

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

Technical Details

- Using DJ and DemeterJ

Combining DJ and DemeterJ

- DJ is a 100% Java solution for adaptive programming.
- DemeterJ has
  - XML style data binding facilities: code generation from schema (class dictionary).
  - Its own adaptive programming language.
- We attempt an optimal integration giving us the strong advantages of both and only few small disadvantages.

Optimal DJ and DemeterJ Integration

- Take all of DJ
- Take all of DemeterJ class dictionary notation
- Take a very tiny bit of DemeterJ adaptive programming language (basically only part that allows us to weave methods).

Combining DJ and DemeterJ

- Pros (advantages)
  - Java class generation from class dictionary (getters, setters, constructors).
  - Parser generation.
  - Better packaging of Java code into different files.
  - MUCH MORE POWERFUL.
- Cons (disadvantages)
  - No longer pure Java solution.
  - need to learn Demeter notation for class dictionaries (similar to XML DTD notation).
  - need to learn how to call DemeterJ and how to use project files.

Combining DJ and DemeterJ

- What do we have to learn about DemeterJ?
  - Class dictionaries: `*.cd` files
  - Behavior files: `*.beh` files. Very SIMPLE!
    - `A { {{ ... }} }` defines methods ... of class A
  - Project files: `*.prj`
    - list behavior files `*.beh` that you are using
  - Commands:
    - `demeterj new, demeterj test, demeterj clean`

Combining DJ and DemeterJ

- What you might forget in class dictionaries that are used with DJ:
  - `import edu.neu.ccs.demeter.dj.*;`
  - visitors need to inherit from `Visitor`
  - parts of visitors need to be defined in class dictionary

Combining DJ and DemeterJ

- Structuring your files
  - put reusable visitors into separate behavior files.
  - put each new behavior into a separate behavior file. Visitors that are not planned for reuse should also go into same behavior file. Update the class dictionary with required structural information for behavior to work.
  - List all `*.beh` files in `.prj` file.

Plans for DJ

- Continued use and refinement: 50+ students this quarter
- Write compile-time checker and partial evaluator for adaptive part.

Example : Count Aspect Pattern

```
collaboration Counting {
   participant Source {
       expect TraversalGraph getT();
     public int count (){ // traversal/visitor weaving
          getT().traverse(this, new Visitor(){ int r;
            public void before(Target host){ r++; }
            public void start() { r = 0;} …) } }
   participant Target {}
}
```

Base:
Meta variable: bold
Keywords: underscore

Adapter 1

classGraph1 is fixed and therefore the traversal is fixed

```
adapter CountingForBusRoute1 {
   BusRoute is Counting.Source
     with {
       TraversalGraph getT() {
         ClassGraph classGraph1 = new ClassGraph();
         return
          new TraversalGraph(classGraph1,
           new Strategy("from BusRoute via BusStop to Person"));}
     }
   Person is Counting.Target { }
}
```