Smaller, More Evolvable Software

Karl J. Lieberherr
Northeastern University
College of Computer Science
lieber@ccs.neu.edu/www.ccs.neu.edu/home/lieber

Smaller, More Evolvable Software

- Use standard Java and OMG technology (UML) with better design and implementation techniques
- Make smaller by eliminating redundancies
- Make more evolvable by bringing code closer to design and by avoiding tangling in code

Thanks to Industrial Collaborators/Sponsors

- Citibank, SAIC: Adaptive Programming
- IBM: Theory of contracts, Adaptive Programming
- Mettler Toledo: OO Evolution
- Xerox PARC: Aspect-Oriented Programming (Gregor Kiczales et al.)

- supported by DARPA (EDCS) and NSF

Many Contributors

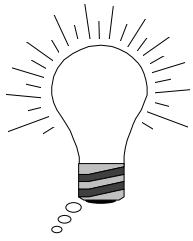
- The Demeter/C++ team (Cun Xiao, Walter Huersch, Ignacio Silva-Lepe, Linda Seiter, …)
- The Demeter/Java team (Doug Orleans, Johan Ovlinger, Crista Lopes, Kedar Patankar, Joshua Marshall, Binoy Samuel, Geoff Hulten, Linda Seiter, ...)
- Faculty: Mira Mezini, Jens Palsberg, Boaz Patt-Shamir, Mitchell Wand

Plan for talk

- AOP and AP (avoid code tangling)
- Bus simulation example (untangle structure and behavior)
- Law of Demeter dilemma and AP
- Tools for AOP and AP
- Synchronization aspect
- History, Technology Transfer

Theme, Idea

- good separation of concerns is the goal
- concerns should be cleanly localized
- programs should look like designs
- avoid *code tangling*



Some sources of code tangling

- Code for a requirement is spread through many classes. In each class, code for different requirements is tangled together.
- Synchronization code is tangled with sequential code.
- Data structure information is tangled with behavior.

Drawbacks of object-oriented software development

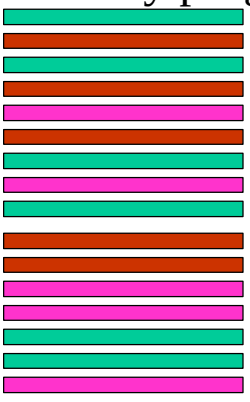
- programs are tangled and redundant
  - data-structure tangling, data structure encoded repeatedly
  - synchronization tangling
  - distribution tangling
  - behavior tangling, pattern tangling
- programs are hard to maintain and too long
  - because of tangling and redundancy

Eliminating drawbacks with aspect-oriented programming (AOP)

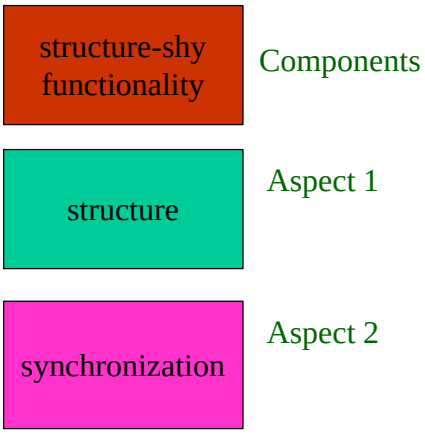
- Solution: Split software into cooperating, *loosely* coupled components and aspect-descriptions.
- Untangles and eliminates redundancy.
- Aspect description examples: marshalling, synchronization, exceptions etc.

Cross-cutting of components and aspects

ordinary program



better program



Aspect-Oriented Programming

components and aspect descriptions

weaver (compile-time)

High-level view, implementation may be different

Source Code (tangled code)

Examples of Aspects

- Data Structure
- Synchronization of methods across classes
- Remote invocation (e.g., using Java RMI)
- Quality of Service (QoS)
- Failure handling
- External use (e.g., being a Java bean)
- Replication

What is adaptive programming (AP)? A special case of AOP

- One of the aspects or the components use graphs which are referred to by traversal strategies.
- A traversal strategy defines traversals of graphs without referring to the details of the graphs.
- Adaptive programming is aspect-oriented programming with traversal strategies.

AOP is useful with and without objects

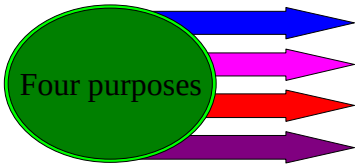
- AOP not tied to OO
- Also AP not tied to OO
- From now on focus on OO AP
- Remember: OO AP is a special kind of OO AOP.

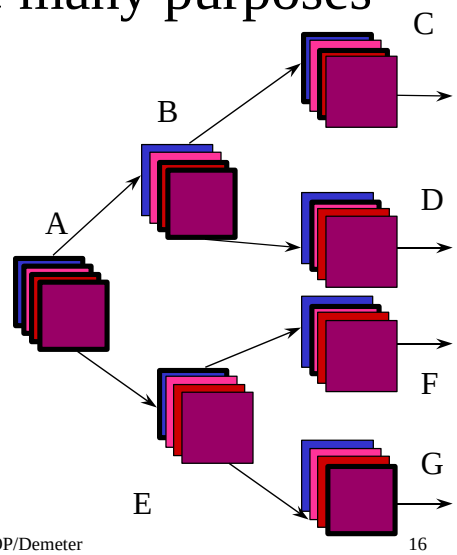
Objects serve many purposes

- Software objects need to serve many purposes.
- For each purpose, some of the object structure is noise.
- Want to filter out that noise and not talk about it. Focus only on what is relevant. Specify object structure in one place.

Objects serve many purposes

shorter A(E+C)...
A(D+F)...
ABC…
A(B+G)...





Benefits of OO AP

- robustness to changes
- shorter programs
- design matches program
  more understandable code
- partially automated evolution
- keep all benefits of OO technology
- improved productivity

Applicable to design and documentation of your current systems.

Five Patterns

- Structure-shy Traversal
- Selective Visitor
- Structure-shy Object
- Class Graph
- Growth Plan

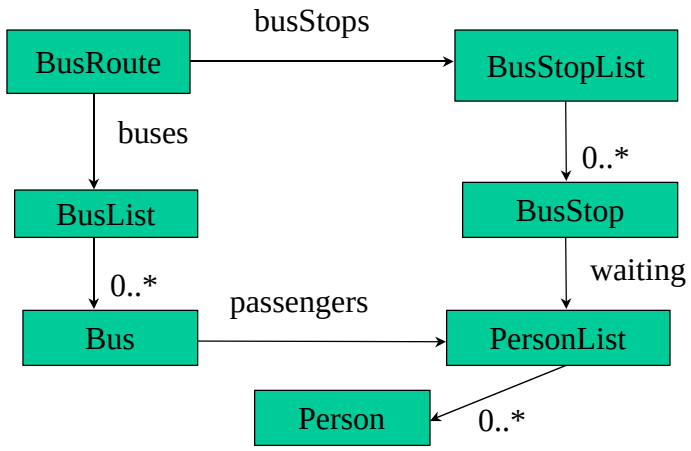
On-line information

- $D = www.ccs.neu.edu/research/demeter
- $D is Demeter Home Page
- $AOO =$D/course/f97/
- Lectures are in: $AOO/lectures
- Patterns in powerpoint/PLAP.ppt and powerpoint/PLAP-v4.ppt

1: Basic UML class diagrams

- Graph with nodes and directed edges and labels for nodes and edges
- Nodes: classes, edges: relationships
- labels: class kind, edge kind, cardinality

UML Class Diagram

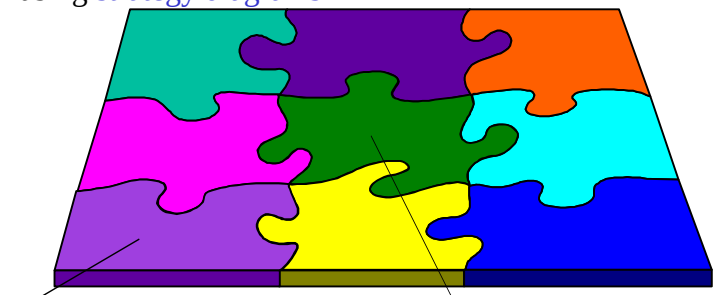


2: Traversals / Collaborating classes

- To process objects we need to traverse them
- Traversal can be specified by a group of collaborating classes

Collaborating Classes

use connectivity in class diagram to define them succinctly using strategy diagrams

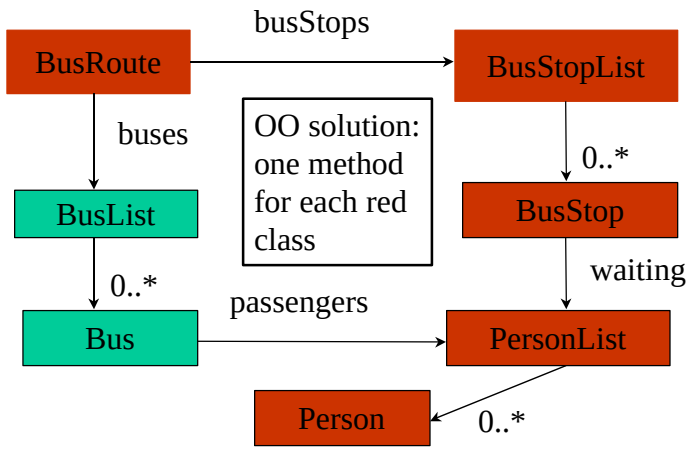


`from` Company `to` Employee

`from` Customer `to` Agent

Collaborating Classes

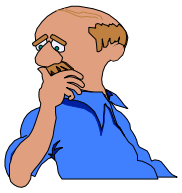
find all persons waiting at any bus stop on a bus route



3: Traversal Strategy Graphs

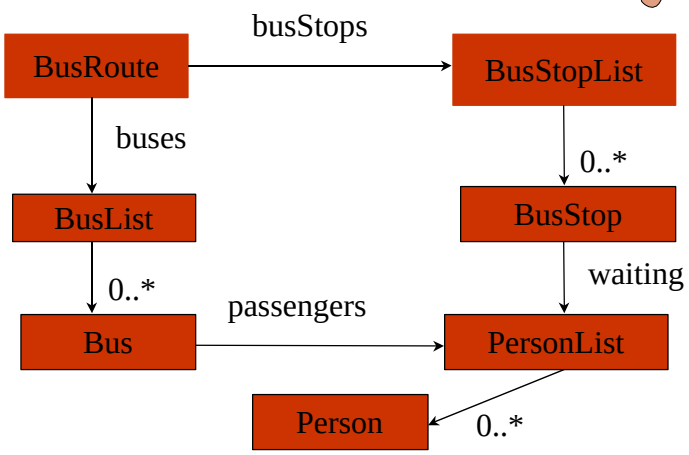
- Want to define traversals succinctly
- Use graph to express abstraction of class diagram
- Express traversal intent: useful for documentation of object-oriented programs

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



Traversal Strategy

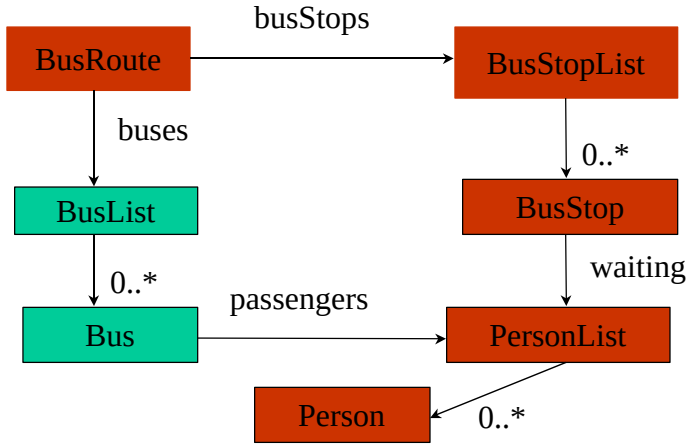
first try: `from` BusRoute `to` Person



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

Traversal Strategy

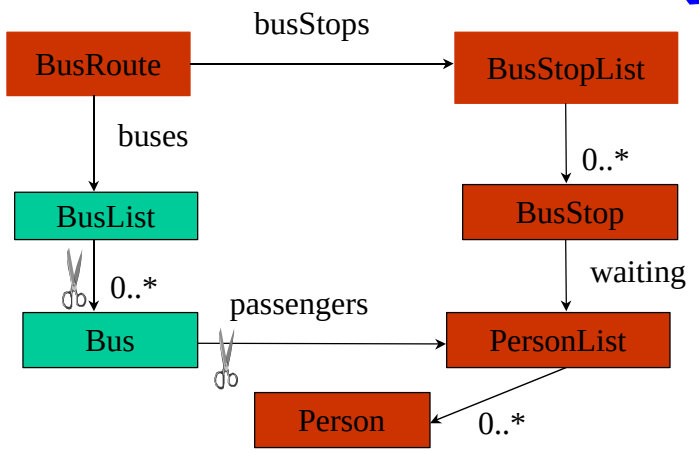
`from` BusRoute `through` BusStop `to` Person



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

Traversal Strategy

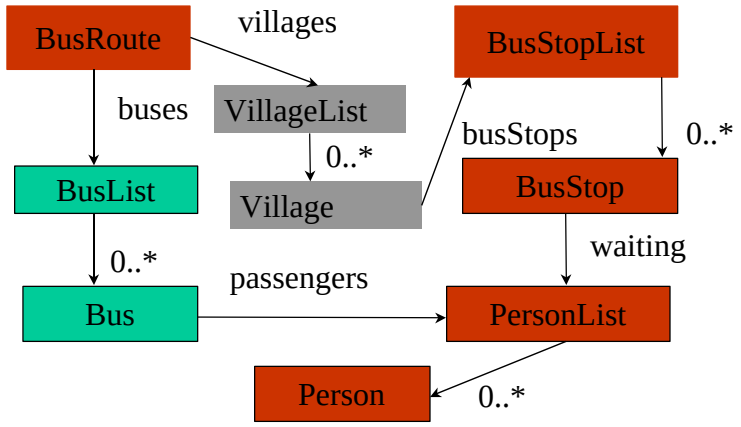
Altern.: `from` BusRoute `bypassing` Bus `to` Person



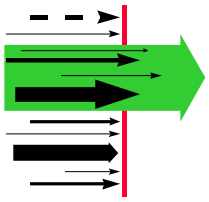
find all persons waiting at any bus stop on a bus route

Robustness of Strategy

`from` BusRoute `bypassing` Bus `to` Person



Filter out noise in class diagram



- only three out of seven classes are mentioned in traversal strategy!

`from` BusRoute `through` BusStop `to` Person

replaces traversal methods for the classes
BusRoute VillageList Village BusStopList BusStop
PersonList Person

Nature Analogy

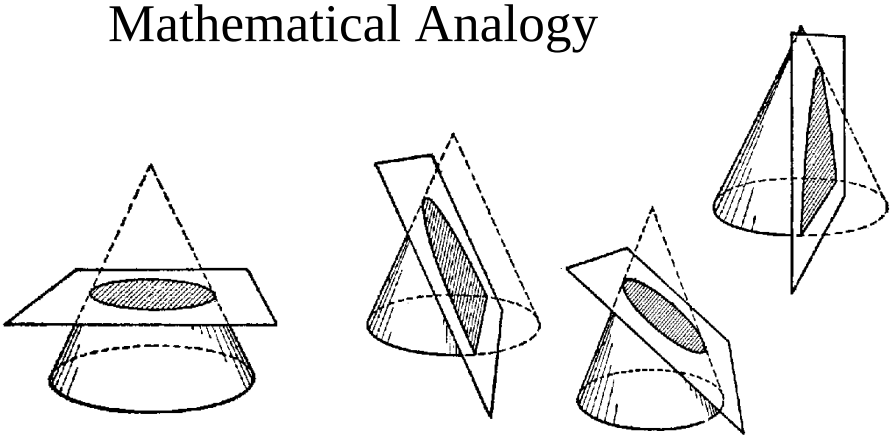
same strategy in different class graphs: similar traversals
same seeds in different climates: similar trees

warm climate

cold climate

same cone different planes define different point sets
same strategy different class graphs define different path sets

Mathematical Analogy



Why Traversal Strategies?

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

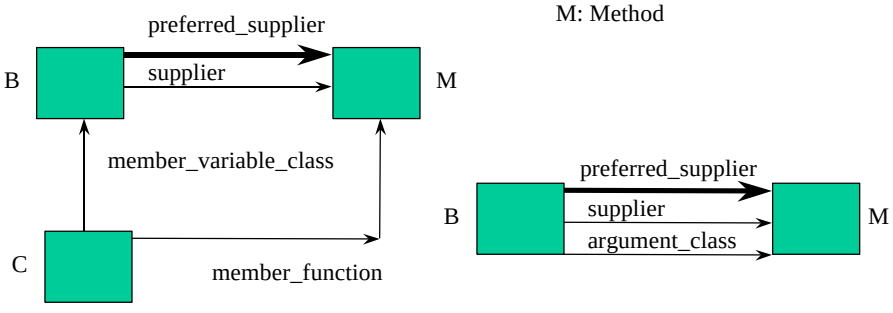
- Dilemma:
  - Small method problem of OO (if followed) or
  - Unmaintainable code (if not followed)

- Traversal strategies are the solution to this dilemma

Law of Demeter (simplified)

All suppliers should be preferred suppliers

B, C: Class
M: Method



OR

4: Adaptive Programming

- How can we use strategies to program?
- Need to do useful work besides traversing: visitors
- Incremental behavior composition using visitors

Writing Adaptive Programs with Strategies

strategy: `from` BusRoute `through` BusStop `to` Person

```
BusRoute {
   traversal waitingPersons(PersonVisitor) {
    through BusStop to Person; } // from is implicit
   int printWaitingPersons() // traversal/visitor weaving instr.
     = waitingPersons(PrintPersonVisitor);
PrintPersonVisitor {
   before Person (@ … @) … }
PersonVisitor {init (@ r = 0; @) … }
```

Extension of Java: keywords: `traversal init through bypassing to before after` etc.

Adaptive Programming

Strategy Diagrams

are use-case based abstractions of

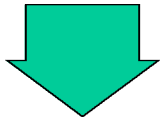
Class Diagrams

define family of

Object Diagrams

Adaptive Programming

Strategy Diagrams



define traversals of

Object Diagrams

Adaptive Programming

Strategy Diagrams

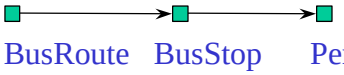
guide and inform

Visitors

AP

- An application of automata theory.
- Apply idea of regular expressions and finite automata to data navigation.

Strategy Diagrams



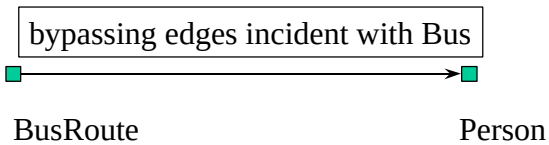
Nodes: positive information: Mark corner stones in class diagram: Overall topology of collaborating classes. 3 nodes:

`from` BusRoute

`through` BusStop

`to` Person

Strategy Diagrams



Edges: negative information:
Delete edges from class diagram.

`from` BusRoute `bypassing` Bus `to` Person

5: Tools for Aspect-Oriented and Adaptive Programming

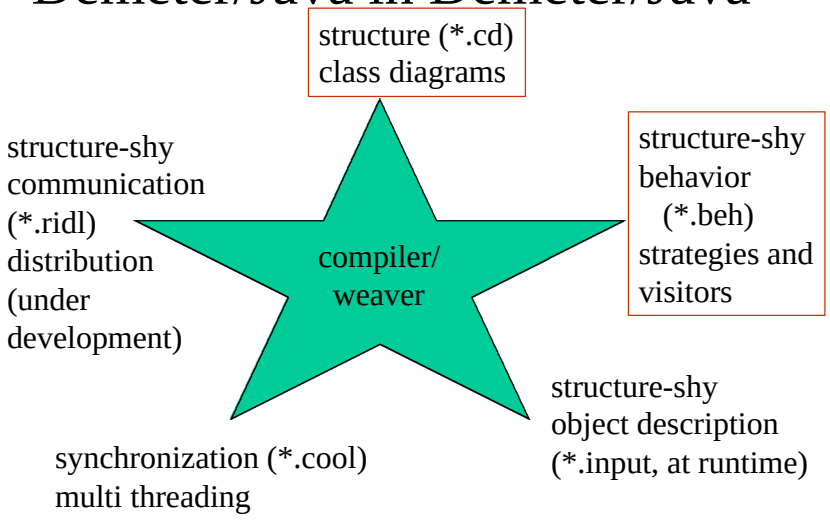
- many free tools available, including Aspect/J from Xerox PARC
- one commercial tool which uses a point and-click interface to define traversals (StructureBuilder from Tendril)

What is Demeter

- A high-level interface to object-oriented programming and specification systems
- Demeter System/OPL =

  Demeter Method + Demeter Tools/OPL
- So far: OPL = {Java, C++, Perl, Borland Pascal, Flavors}
- Demeter Tools/OPL = Demeter/OPL

Demeter/Java in Demeter/Java



Goal of Demeter/Java

- Avoid code tangling
  - traversal strategies and visitors untangle structure and behavior
  - visitors untangle code for distinct behaviors
  - COOL untangles synchronization issues and behavior
  - RIDL untangles remote invocation issues and behavior

Free Tools on WWW

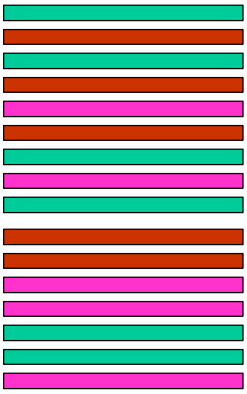
- Demeter/C++
- Demeter/Java
- Demeter/StKlos
- Dem/Perl5
- Dem/C++
- Dem/CLOS
- Demeter/Object Pascal

last five developed outside our group

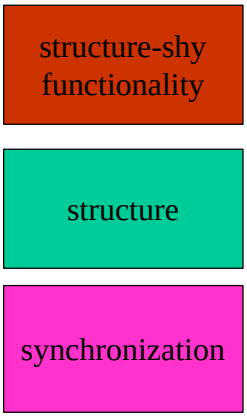
Aspect/J from Xerox

Cross-cutting in Demeter/Java

generated Java program



Demeter/Java program



replicated!

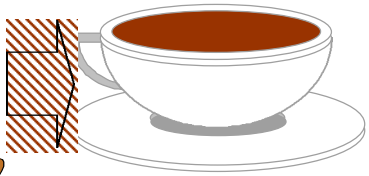
Demeter/Java

www.ccs.neu.edu/research/demeter

- class diagrams
- functionality
  - strategies
  - visitors
- etc.

weaver

Executable
Java code for your favorite commercial Java Software Development Environment



AP Studio

- visual development of traversal strategies relative to class diagram
- visual feedback about collaborating classes
- visual development of annotated UML class diagrams

Strengths of Demeter/Java

- Theory
  - Novel algorithms for strategies
  - Formal semantics
  - correctness theorems
- Practice
  - Extensive feedback (7 years)
  - Reflective implementation

Meeting the Needs

- Demeter/Java
  - Easier evolution of class diagrams (with strategy diagrams)
  - Easier evolution of behavior (with visitors)
  - Easier evolution of objects (with sentences)

Commercial Tools available on WWW

StructureBuilder from Tendril Software Inc.

Has support for traversals

www.tendril.com

Tendril Software, Inc.

- Researched in Fall/Winter of 1995
- Founded in 1996
- Sells a new generation of Java development tools using some of the Demeter ideas: point and click traversals and object transportation

Generated Methods

- Sequence of actions which contain enough detail to actually generate code
- Contains a palette of data structures
- New data structures can be added by writing new templates

Synchronization Aspect

- Developed by Crista Lopes
- Separate synchronization and behavior

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();
}
```

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

Using Demeter/Java, *.beh file

With coordinator: basics

```
BoundedBuffer {
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; @)
```

Using Demeter/COOL, put into *.cool file

Coordinator

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

exclusion sets

coordinator variables

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
- `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

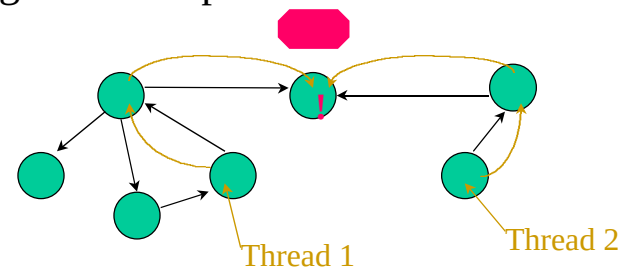
- The smallest unit of synchronization is the method.
- The provider of a service defines the synchronization (monitor approach).
- Coordination is contained within one coordinator.
- Association from object to coordinator is static.

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.

COOL

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



coordinator

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

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();
  }
}
```

COOL Shape

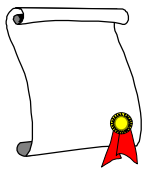
```
coordinator Shape {
  selfex {adjustLocation,
          adjustDimensions}
  mutex {adjustLocation,x}
  mutex {adjustLocation,y}
  mutex {adjustDimensions,
          width}
  mutex {adjustDimensions,
          height}
}
```

Risks of adaptive OO

- Advantages: Simpler programs for next project (compensates for learning curve). Programs are easier to evolve and maintain.
- Disadvantages: Additional training costs. New concepts, debugging techniques and tools to learn.

Experience regarding training costs

- GTE project which took approximately four man months by non-adaptive techniques, took only 7 days to complete with adaptive techniques (using Demeter/Java).
- Our experience with Demeter/C++ is that the first project also has a shorter development time and maintenance is much simpler.



Real Life

- Used in several commercial projects
- Implemented by several independent developers
- Used in several courses, both academic and commercial

Scenarios

- Best: Use Demeter/Java for future projects. Build library of adaptive components. Reduce software development and maintenance costs significantly.
- Worst: Use Demeter/Java only to generate Java code, but then you maintain Java code manually. You still win since a lot of useful Java code is produced.

History

- 1982 Hades (HArdware DEScription language by Niklaus Wirth at ETH Zurich)
- 82-85 Zeus (a brother of Hades, a silicon compilation language developed at Princeton University/MIT, implemented at GTE Labs; predecessor of VHDL)
- 1985- Demeter (a sister of Zeus, used to implement Zeus, started at GTE Labs)

History

- 1990 First traversal specifications
- 1995 Separation of Concerns paper by Huersch and Lopes started “untangling” movement TR NU-CCS-95-03. Collaboration with Xerox PARC started (initiated in 1994).
- 1996 Gregor Kiczales and his group name and further develop AOP.

Technology Evolution

Object-Oriented Programming

Traversal/
Visitor style

Law of Demeter dilemma
Tangled structure/behavior

Adaptive Programming

Tangled code

Aspect-Oriented Programming

Benefits of Demeter

- robustness to changes
- shorter programs
- design matches program,
  more understandable code
- partially automated evolution
- keep all benefits of OO technology
- improved productivity

Applicable to design and documentation of your current systems.

Related Work

- Gregor Kiczales and his group: Open Implementation, Aspect-Oriented Programming
- Polytypic programming and shape-polymorphism
- Alberto Mendelzon: GraphLog query language, query languages for semi-structured data

Where to get more information

- Adaptive Programming book
- Demeter/Java page
- Demeter home page:

www.ccs.neu.edu/research/demeter/

Summary

- What has been learned: Concepts of AOP, simple UML class diagrams, strategies and adaptive programs
- How can you apply:
  - Demeter/Java takes adaptive programs as input
  - Document object-oriented programs with aspect-descriptions and strategies
  - Design in terms of traversals and visitors and aspects.