

Adaptive Object-Oriented Software Development

The Demeter Method

9/30/99

AOOP / Demeter

1

Introduction

- Software engineering
- Programming languages
- Hands-on, practical, useful

9/30/99

AOOP / Demeter

2

Summary of Course

- How to design and implement flexible object-oriented software using the principles of adaptiveness in combination with UML and Java.
- How to design and implement flexible 100% pure Java software.

Who is in Attendance?

- Software developers who have some experience with object-oriented programming.
- Should know concepts of OOP, like classes, methods, and late binding.

Agenda for Adaptive Object-Oriented Software/Demeter

Demeter/Java	Adaptive Programming	
Java	Aspect-Oriented Progr.	strategy graphs
Java environment		class graphs
UML	principles	object graphs
XML	heuristics	state graphs
	patterns	
requirements	idioms	
domain analysis	theorems	use cases
design	algorithms	interfaces
implementation	Demeter Method	traversals
	iterative development	visitors
	spiral model	packages

9/30/99

AOOP / Demeter

5

Agenda

- UML class diagrams. Perspective: analysis, design, implementation
- Class graphs as special cases of UML class diagrams
- Textual representation of class graphs
- Default implementation of UML class diagrams as class graphs

9/30/99

AOOP / Demeter

6

Agenda: UML collaborations as use case realizations

- Participants
- Behavior/Structure separation
- Provided and required interface

Quote by Grady Booch, Aug. 30, 1999

- From the perspective of software architecture, we have found that collaborations are the soul of an architecture, meaning that they represent significant design decisions that shape the system's overall architecture.
- We have been using collaborations to capture the semantics ... of e-business ...

Important concepts

- Specification-level and instance-level collaborations
- Roles
- Connectors
- Two aspects of collaborations: structural and behavioral

High-level view of **collaborations**

- A **collaboration** defines a context in which a behavior can be specified in terms of interactions between the participants of the collaboration.
- A model describes a whole system; a **collaboration** is a slice or projection of that model.

Specification-level and instance-level collaborations

- Specification-level collaborations
 - great for imposing abstract patterns that you want to impose upon a system
- Instance-level collaborations
 - show actual imprint of pattern upon a real system

Instance-based collaborations

- A group of collaborating objects performing some task together.
- Not general; useful as examples.

Specification-based collaborations

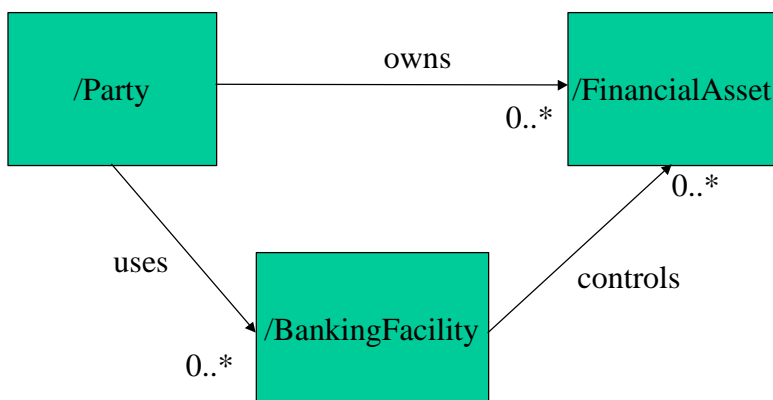
- A group of collaborating roles specifying some task.
- General; useful to specify behavior.
- Has structural and behavioral part.

9/30/99

AOOP / Demeter

13

Structural Part of a Bank (for bank-related collaborations)



9/30/99

AOOP / Demeter

14

What are AP&PC? Adaptive Plug&Play Component

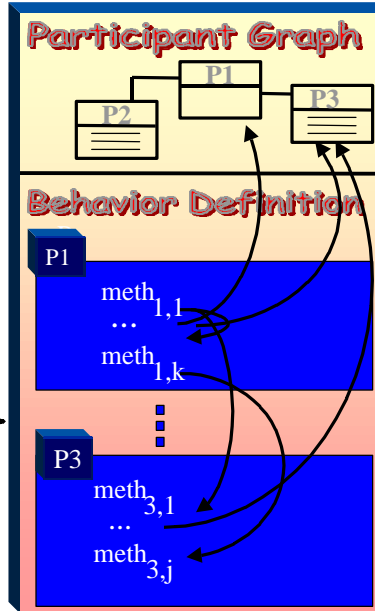
- AP&PC are a **language construct** that captures behaviour involving several roles (cross-cuts class boundaries)
- the programmer uses **classes** to implement the **primary data (object) structure**
- the programmer uses **AP&PC** to implement higher-level behavior cross-cutting the primary structure in a modular way

What are AP&PC ?

- AP&PC have **provided** and **expected interfaces**
- The expected interface consists of an **ideal role graph (Participant Graph, PG)** to enable defining one aspect of the system with **limited knowledge** about the object model and/or other aspects defined by other components
- AP&PC can be **deployed** into PGs or **concrete class graphs** and/or **composed/refined by 3rd parties** (reuse) by mapping interfaces via explicit connectors

AP&PC

minimal assumptions on application structure
+
expected interfaces



written to the PG
similar to an OO
program is written
to a concrete class
graph

add new functionality
+
(enhance the expected)

provided
=
everything declared
public

AP&PC

- A set of roles forming a graph called the participant graph (represented, e.g., by a UML class diagram). Participant
 - formal argument to be mapped
 - expects function members
 - (reimplementations)
 - local data and function members

AP&PC (continued)

- Local classes: visibility: AP&PC
- AP&PC-level data and function members.
There is a single copy of each global data member for each deployment

What is an AP&PC?

any identifiable **slice of functionality** that describes a **meaningful service**, involving, in general, **several roles**,

- with well-defined **expected** and **provided interfaces**,
- formulated for an **ideal ontology** - the expected interface
- subject to **deployment into** several **concrete ontologies** by 3rd parties
- subject to **composition** by 3rd parties
- subject to **refinement** by 3rd parties

An ontology is, in simple terms, a collection of roles with relations among them plus constraints on the relations.

Collaboration deployment/composition

- **Deployment** is mapping **idealized ontology to concrete ontology**
 - specified by **connectors** separately from components
 - without mentioning irrelevant details of concrete ontology in map to keep deployment flexible
 - non-intrusive, parallel, and dynamic deployment
- **Composition** is **mapping the provided interface** of one (lower-level) collaboration **to the expected interface** of another (higher-level) collaboration
- deployment is a special case of composition, where the lower level collaboration is a concrete ontology (no expected interface)

The goal

The goal is to separate concerns (each decision in a single place) and minimize dependencies between them (loose coupling):

- less tangled code, more natural code, smaller code
- concerns easier to reason about, debug and change
- a large class of modifications in the definition of one concern has a minimum impact on the others
- more reusable, can plug/unplug as needed

Deployment/Composition of AP&PCs

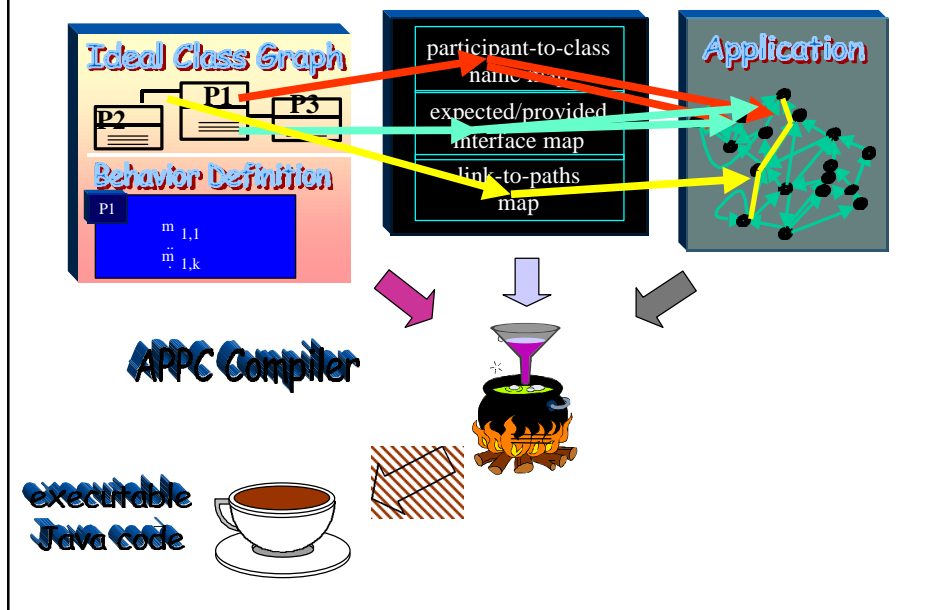
- Specified by **connectors** separately from AP&PC
- Connectors use
 - regular-expressions to express sets of method names and class names and interface names
 - standard code everywhere simple method name mapping is not enough
 - regular expression-like constructs for mapping graphs

9/30/99

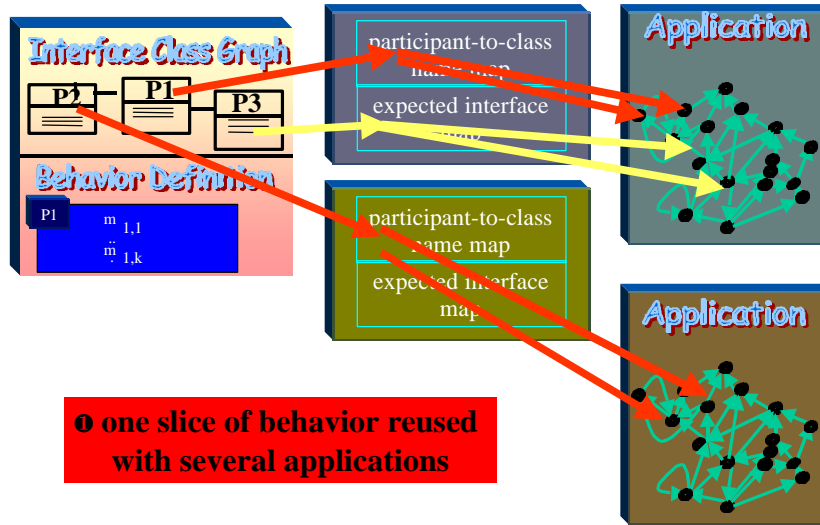
AOOP / Demeter

23

Deploying/Composing AP&PC



One AP&PC deployed into several applications

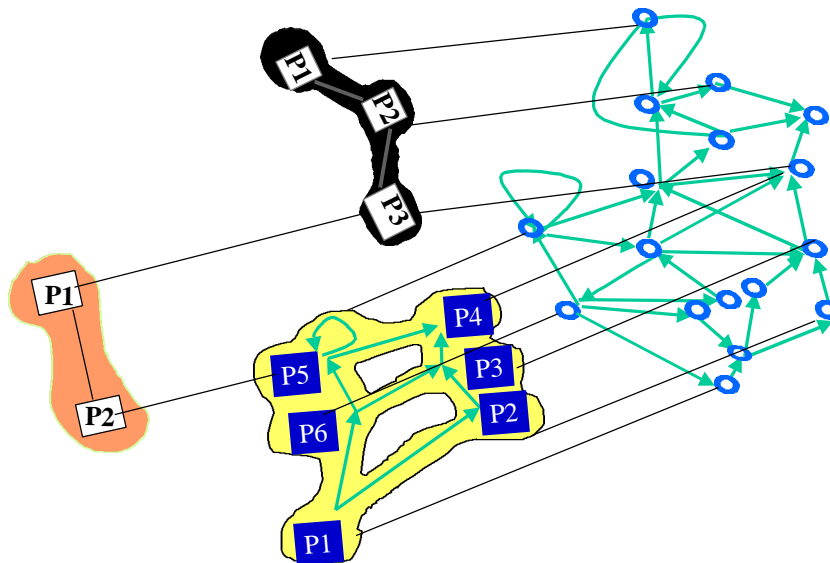


9/30/99

AOP / Demeter

25

Software Structure with AP&PCs



9/30/99

AOP / Demeter

26

Ideal Participant Graph Where Have We Seen That Before ?

Quote:

Avoid traversing multiple links or methods. **A method should have limited knowledge of an object model.** A method must be able to traverse links to obtain its neighbors and must be able to call operations on them, but it should not traverse a second link from the neighbor to a third class.

Rumbaugh and the Law of Demeter (LoD)

Collaborations in UML 1.3

- UML: A collaboration consists of a set of ClassifierRoles and AssociationRoles. A ClassifierRole specifies one participant of a collaboration. A collaboration may be presented at two levels: specification-level and instance-level.
- AP&PC: A collaboration consists of a set of participants and a participant graph. We use same terminology for specification and instance level.
- Correspondences: participant graph :: nodes: ClassifierRole, edges: AssociationRole, AssociationEndRole

Collaborations in UML 1.3

- UML: Each participant specifies the required set of features a conforming instance must have. The collaboration also states which associations must exist between participants.
- AP&PC: Each participant has a required interface. The participant graph is part of the required interface.
- Correspondences: Both separate behavior from structure. Both use the UML class diagram notation to specify the associations between participants. ClassifierRole names start with a /.

Collaborations in UML 1.3

- UML:
 - The term of classifier role and classifier is strange. Why not use participant role and participant?
 - The base classifier must have a subset of the features required by the classifier role. With AP&PC we are more flexible: we have a connector (or adapter) that allows to implement the required features. The base classifier must only provide enough “ingredients” to implement the required interface.

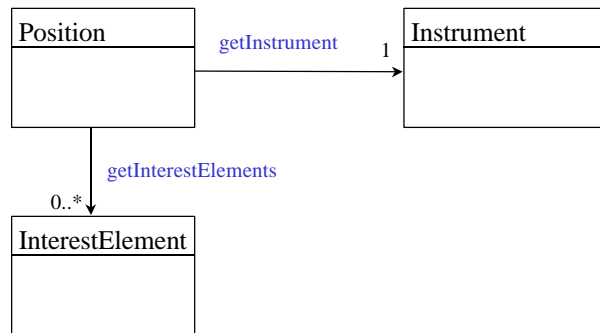
Collaborations in UML 1.3

- UML: A collaboration may also reference ... needed for expressing structural requirements such as generalizations between the classifiers ...
- AP&PC: All structural requirements are represented by the participant graph.

Collaborations in UML 1.3

- UML: An interaction specifies the communication between a set of interacting instances performing a specific task. Seems not sufficient to generate code.
- AP&PC: Interactions are specified using Java code or Interaction Schemata, an improved form of interaction diagrams (see TOOLS '99 paper by Sangal, Farrel, Lieberherr, Lorenz).

UML ClassifierRole Diagram for Interest Calculation Collaboration



9/30/99

AOOP / Demeter

33

Red: provided
Blue: required

Interest Calculation Collaboration

- Position
 - interest (from,to){ create interest elements for from-date to to-date using position and instrument information and add interest of all InterestElements },
getInstrument, getInterestElements
- Instrument
 - getDaysInYear(y), ...
- InterestElement **interest ()**, ...

9/30/99

AOOP / Demeter

34

Usage for Germany

connector InterestInGermany

AlterSparkkontoPosition is Position

SparInstrument is Instrument with {

int getDaysInYear(Year y)

{return anzTageProJahr(y.conv());}...}

ZinsElement is InterestElement ...

Examples of collaborations

- Traversal/visitor style of programming
 - participant graph: roles participating in traversal
 - behavior: traversal through roles plus visitors
 - connectors: mapping of role names to classes and role methods to class methods (editor)
 - required interface: participant graph plus methods called from visitors

Agenda

- UML interaction diagrams
 - sequence diagrams
 - collaboration diagrams
 - improving interaction diagrams to make them specification languages

Agenda

- Class graphs in textual form (construction, alternation)
- object graphs (graphical and textual form)
- object graphs defined by class graphs
- add repetition classes and optional parts
- translating class graphs to Java

Agenda

- annotating class graph with syntax: class dictionary
- printing objects and language defined by class dictionary
- grammars, parsing, ambiguous grammars, undecidable problem
- LL(1) grammars, robustness of sentences
- etc.

Overview

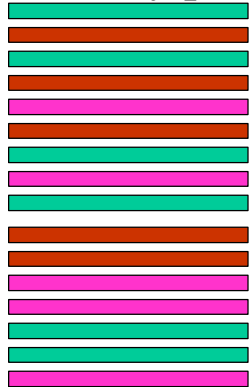
- good separation of concerns is the goal
- concerns should be cleanly localized
- programs should look like designs

Adaptive Programming

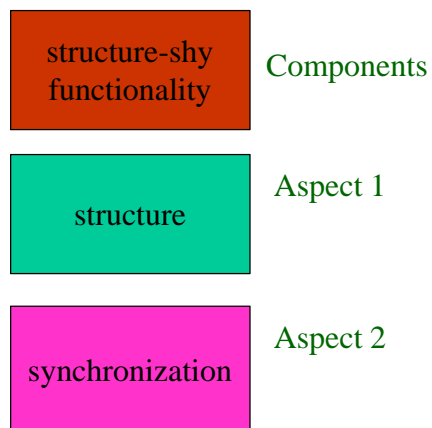
- Programs adapt to interesting context changes
- Structure-shy behavior
- Succinct representation of traversals
- Programming in terms of graph constraints

Cross-cutting of components and aspects

ordinary program

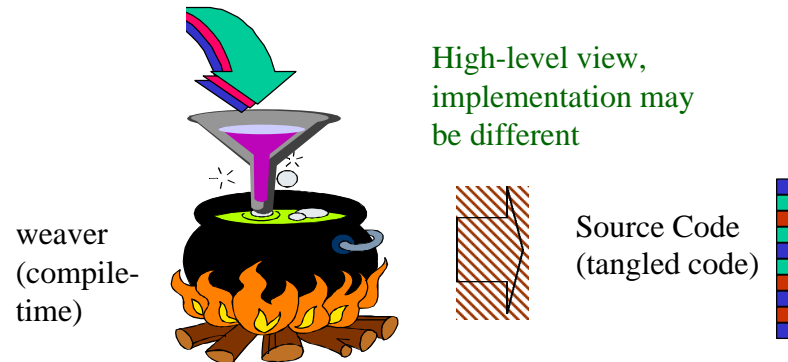


better program



Aspect-Oriented Programming

components and aspect descriptions



9/30/99

AOOP / Demeter

43

Examples of Aspects

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

9/30/99

AOOP / Demeter

44

Connections

- explain adaptive programming in terms of patterns
- Aspect-Oriented Programming (AOP) is a generalization of Adaptive Programming (AP)
- correspondence:
adaptive program : object-oriented program
= sentence : object graph

Vocabulary

- Graph, nodes, edges, labels
- Class graph, construction, alternation
- Object graph, satisfying class graph
- UML class diagram
- Grammar, printing, parsing

Vocabulary

- Traversals, visitors
- Strategy graphs, path set

Overview this lecture

- Basic UML class diagrams
- Traversals/Collaborating classes
- Traversal strategy graphs
- Adaptive programming
- Tools for adaptive programming
- Demeter/Java and AP/Studio

1: Basic UML class diagrams

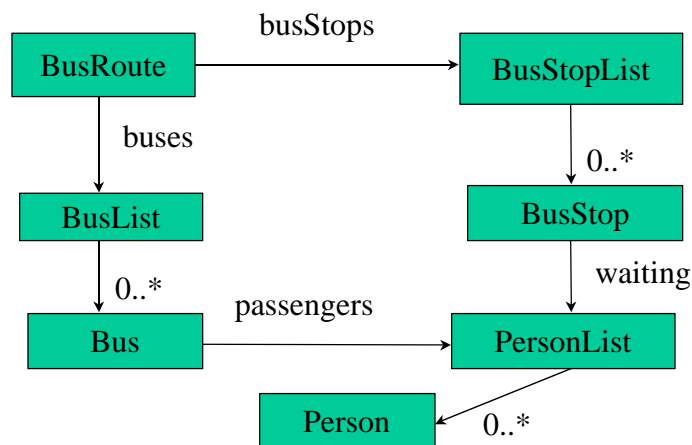
- Graph with nodes and directed edges and labels for nodes and edges
- Nodes: classes, edges: relationships
- labels: class kind, edge kind, cardinality

9/30/99

AOOP / Demeter

49

UML Class Diagram



9/30/99

AOOP / Demeter

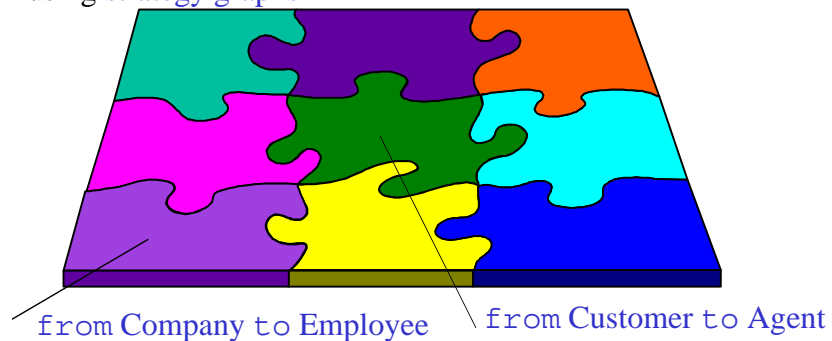
50

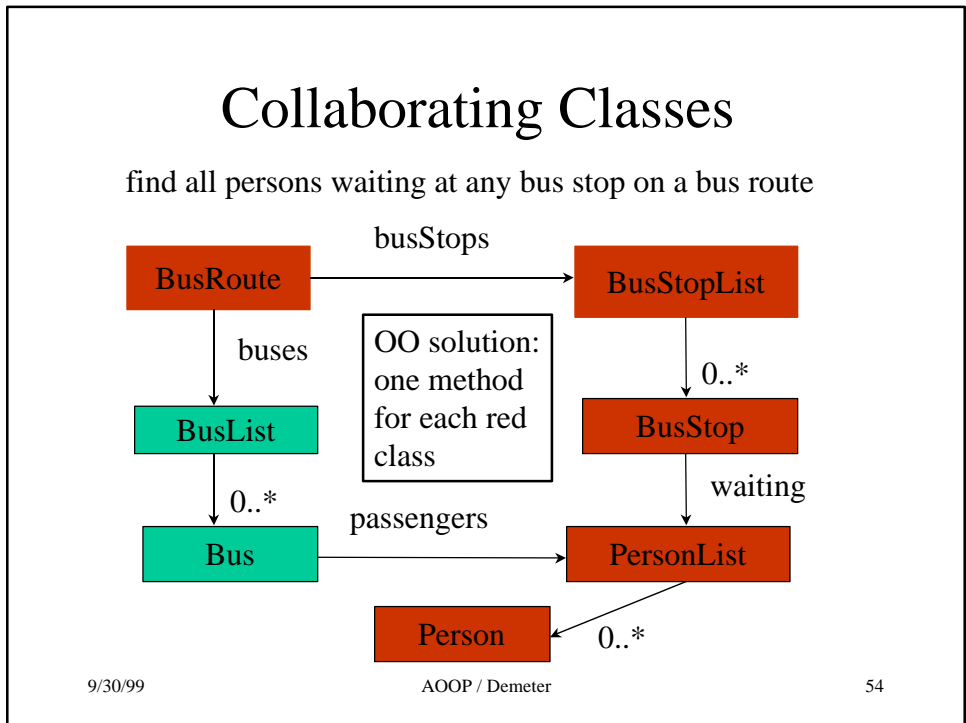
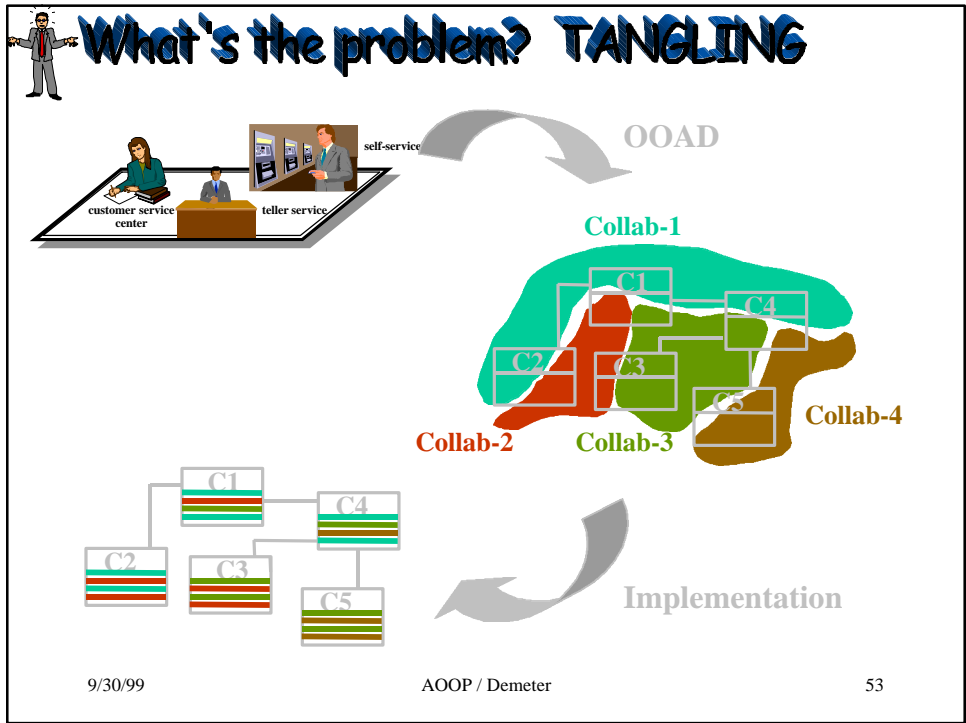
2: Traversals / Collaborating classes

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

Collaborating Classes

use connectivity in class graph to define them succinctly
using [strategy graphs](#)





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

9/30/99

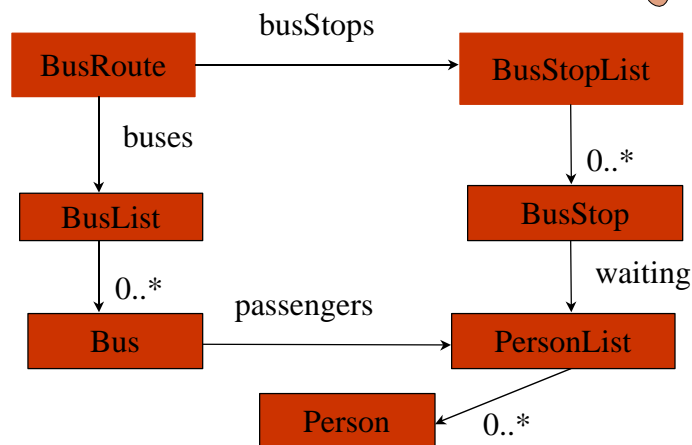
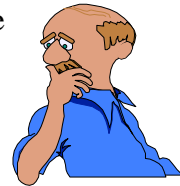
AOOP / Demeter

55

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

Traversal Strategy

first try: *from BusRoute to Person*



9/30/99

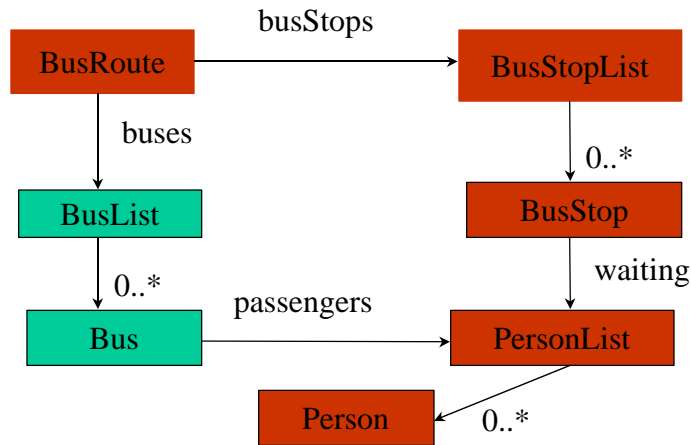
AOOP / Demeter

56

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

Traversal Strategy

from BusRoute through BusStop to Person



9/30/99

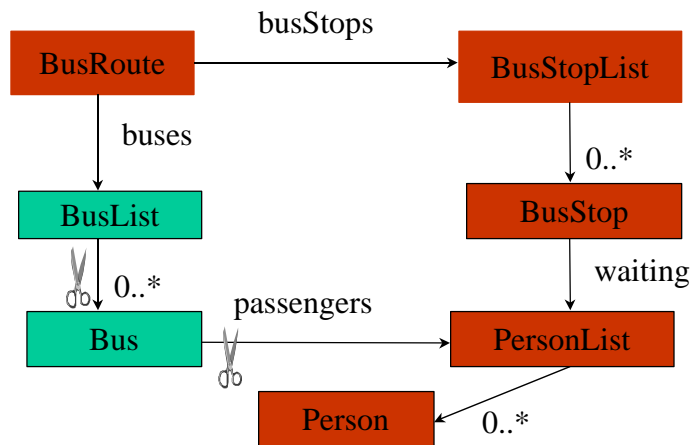
AOOP / Demeter

57

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

Traversal Strategy

Altern.: from BusRoute bypassing Bus to Person



9/30/99

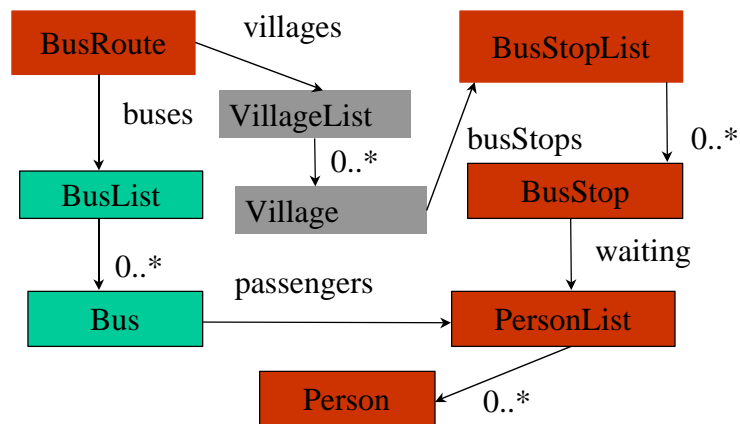
AOOP / Demeter

58

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

Robustness of Strategy

from BusRoute bypassing Bus to Person

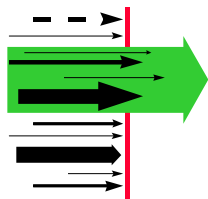


9/30/99

AOOP / Demeter

59

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

9/30/99

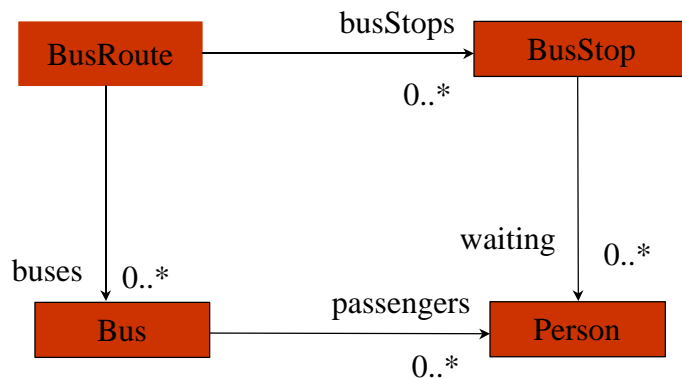
AOOP / Demeter

60

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

Even better: participant graph

from BusRoute through BusStop to Person



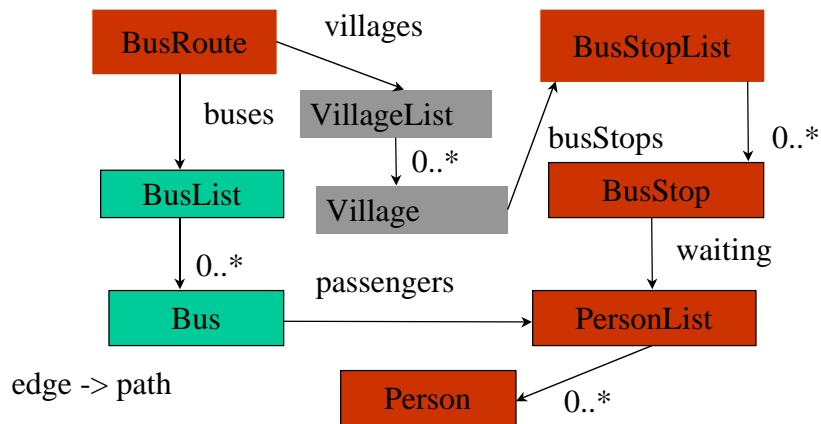
9/30/99

AOOP / Demeter

61

Map participant graph to application class graph

from BusRoute through BusStop to Person



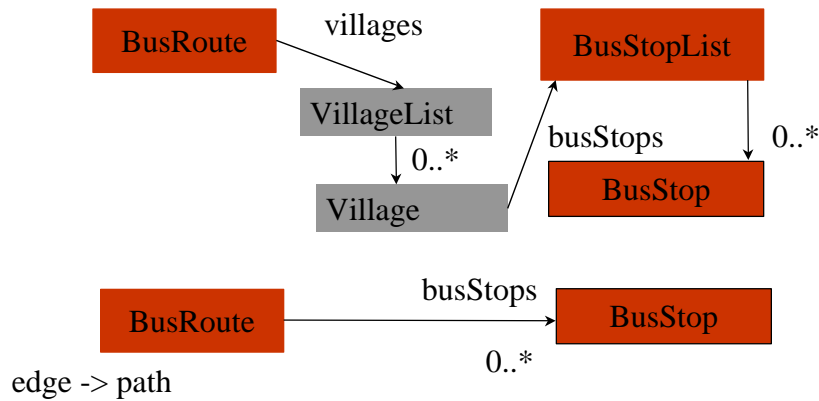
9/30/99

AOOP / Demeter

62

Map participant graph to application class graph

from BusRoute through BusStopList to Person



9/30/99

AOP / Demeter

63

Benefits of participant graph

- Shields program from details of application class graph
- Makes program more robust and simpler

9/30/99

AOP / Demeter

64

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

9/30/99

AOOP / Demeter

65

4: Adaptive Programming

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

9/30/99

AOOP / Demeter

66

Writing Adaptive Programs with Strategies (DJ=pure Java)

String WPStrategy="from BusRoute through BusStop to Person"

```
class BusRoute {
    TraversalGraph WP = new TraversalGraph(Womble.classGraph,
        new Strategy(WPStrategy));
    int printCountWaitingPersons() { // traversal/visitor weaving
        WP.traverse(this, new Visitor() { int r;
            public void before(Person host) { r++; ... }
            public void start() { r = 0; }
            ...
        }
    }
}
// ClassGraph classGraph = new ClassGraph();
```

9/30/99

AOOP / Demeter

67

Writing Adaptive Programs with Strategies (Demeter/Java)

strategy: from BusRoute through BusStop to Person

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

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

9/30/99

AOOP / Demeter

68

Taxi driver analogy

- Streets and intersections correspond to class graph
- Traversal strategy determines how the taxi will navigate through the streets
- You can take pictures before and after intersections
- You can veto sub traversals

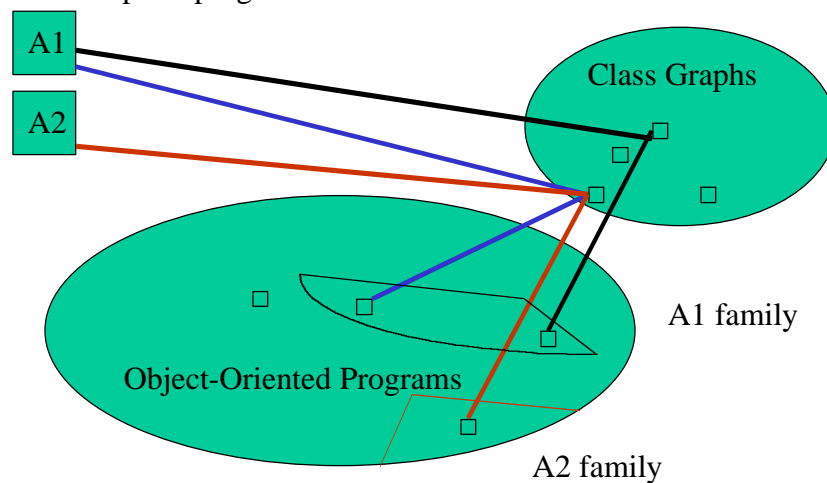
9/30/99

AOOP / Demeter

69

Programming in Large Families

Two adaptive programs



9/30/99

AOOP / Demeter

70

Adaptive Programming

Strategy Diagrams



are use-case based
abstractions of

Class Diagrams



define family of

Object Diagrams

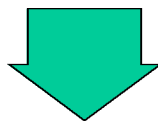
9/30/99

AOOP / Demeter

71

Adaptive Programming

Strategy Diagrams



define traversals
of

Object Diagrams

9/30/99

AOOP / Demeter

72

Adaptive Programming

Strategy Diagrams



guide and
inform

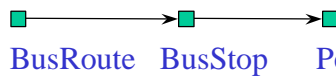
Visitors

9/30/99

AOOP / Demeter

73

Strategy Diagrams



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

from BusRoute

through BusStop

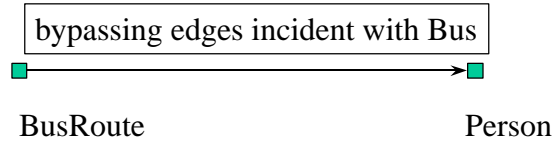
to Person

9/30/99

AOOP / Demeter

74

Strategy Diagrams



Edges: negative information:
Delete edges from class diagram.

from BusRoute bypassing Bus to Person

5: Tools for Adaptive Programming

- free and commercial tools available

Free Tools on WWW

- DJ and AP Library
 - Demeter/C++
 - Demeter/Java ← ●
 - Demeter/StKlos
 - Dem/Perl5
 - Dem/C++
 - Dem/CLOS
 - Demeter/Object Pascal
- last four developed outside our group



9/30/99

AOOP / Demeter

77

Commercial Tools available on WWW

StructureBuilder from Tendril Software Inc.

has support for DJ style actions

www.tendril.com



9/30/99

AOOP / Demeter

78

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



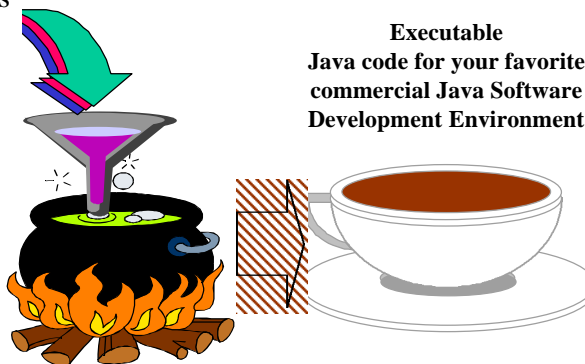
Applicable to design and documentation
of your current systems.

Demeter/Java

www.ccs.neu.edu/research/demeter

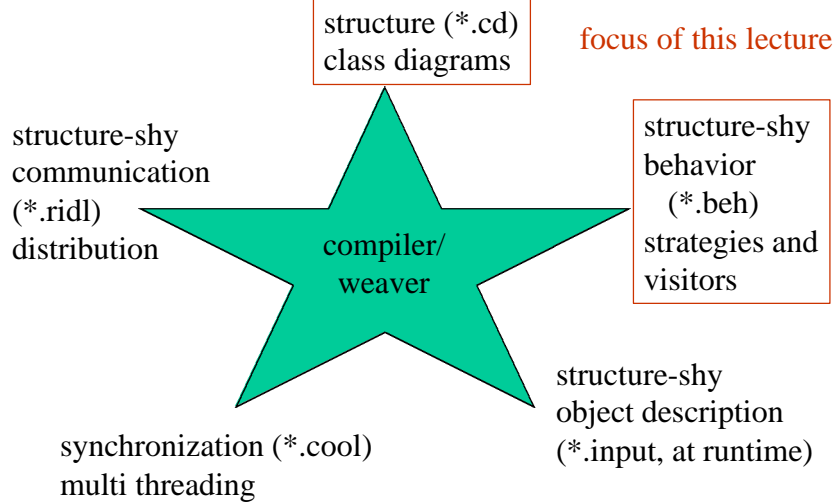
- class diagrams
- functionality
 - strategies
 - visitors
- etc.

weaver



**Executable
Java code for your favorite
commercial Java Software
Development Environment**

Demeter/Java in Demeter/Java



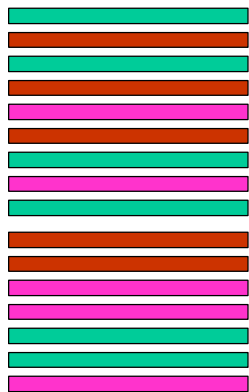
9/30/99

AOOP / Demeter

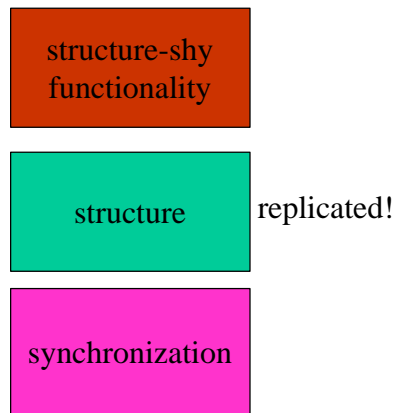
81

Cross-cutting in Demeter/Java

generated Java program



Demeter/Java program



9/30/99

AOOP / Demeter

82

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 (8 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)



Real Life

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

Summary

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

Where to get more information

- Adaptive Programming book
- UML Distilled
- Demeter/Java home page
- Course home page:

[www.ccs.neu.edu/research/demeter/
course/f98](http://www.ccs.neu.edu/research/demeter/course/f98)

Feedback

- Request feedback of training session