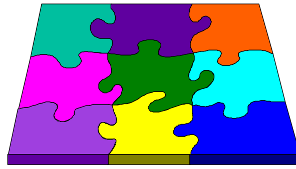


Smaller, More Evolvable Software

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

Outline

- Tangled software: 11-12.30
- Pattern Language for AP: 12.30-1.20
- UML with OCL and AP: 1.30-2.30
- Traversal strategies: 3.00-3.50
- Coordination aspect, Demeter/Java : 4-5

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

- IBM: Theory of contracts, Adaptive Programming
- Citibank, SAIC: 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

References

- Adaptive Object-Oriented Software Development: The Demeter Method, Karl Lieberherr, PWS Publishing Company, 1996.
- Also available in searchable form (PDF format) at:
  - www.ccs.neu.edu/research/demeter/biblio/dem-book.html

References

- Karl Lieberherr and Boaz Patt-Shamir, Traversals of Object-Structures: Specification and Efficient Implementation, TR NU-CCS-97, College of Computer Science, Northeastern University, Sep. 97. ftp://ftp.ccs.neu.edu/pub/people/lieber/strategies.ps

References

- Karl Lieberherr and Doug Orleans, Preventive Program Maintenance in Demeter/Java, International Conference on Software Engineering, 1997, Boston, MA, pages 604-605, ACM Press.

References

- Linda M. Seiter, Design Patterns for Managing Evolution, Ph.D. Thesis, Northeastern University, 1996.
- Cristina Lopes, D, Ph.D. Thesis, Northeastern University, 1996.

References

- Mira Mezini and Karl Lieberherr, Adaptive Plug-and-Play Components for Evolutionary Software Development, Technical Report, 1998, College of Computer Science, Northeastern University.

Plan for first part

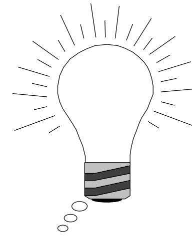
- AOP and AP (avoid code tangling)
- Bus simulation example (untangle structure and behavior)
- Law of Demeter dilemma and AP
- Tools for AOP and AP
- Synchronization aspect
- History, Technology Transfer

Many evolution problems come from tangled designs/programs

- Code for a requirement is spread through many artifacts. In each artifact, code for different requirements is tangled together.
- For example:
  - Information structure is tangled with behavior. We want structure-shyness.
  - Synchronization code is tangled with sequential code.

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

Something wrong with OO

Soren Lauesen,
Real-Life Object-Oriented Systems,
IEEE Software, 1998, March/April,
PAGES 76-83.

http://www.cbs.dk/~slauesen/OOcaseStudies/

It was like "reading a road map through a soda straw".

OO problems

It was like "reading a road map through a soda straw".

He means that the soda straw is pointing to one class at a time and you have to figure out what the collaborations are.

Another argument for separating high-level operations from the objects derives from GTE's experience with large OO systems.

GTE did not succeed until it put *control flow* and *business processes* -- high-level operations -- outside class behavior. If built into the classes involved, it was impossible to get an overview of the control flow.

It was like "reading a road map through a soda straw".

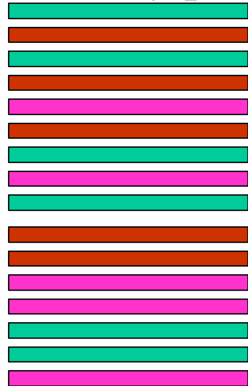
Eliminating drawbacks with aspect-oriented programming (AOP)

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

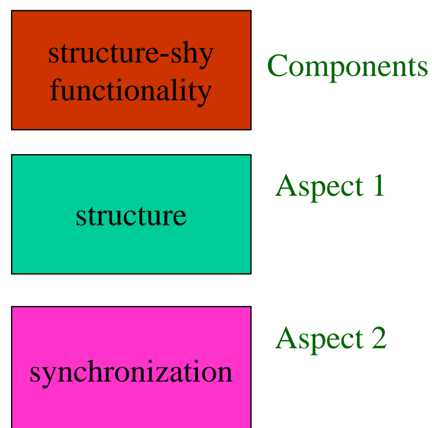
4/20/98 AOP/Demeter 19

Cross-cutting of components and aspects

ordinary program



better program



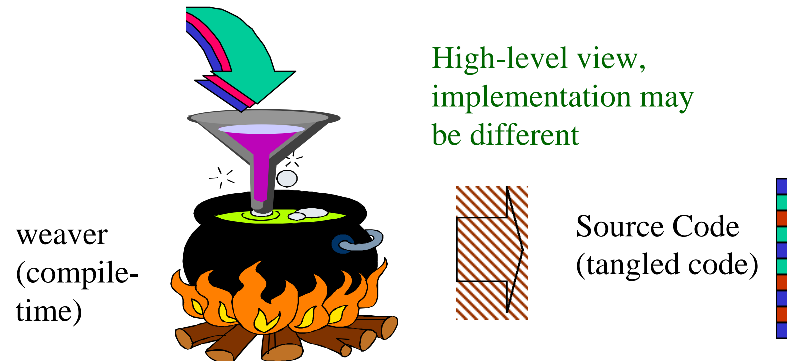
Components

Aspect 1

Aspect 2

4/20/98 AOP/Demeter 20

Aspect-Oriented Programming



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.

Two types of concerns: avoid tangling of any two concerns

- Generic concerns
  - structure
  - coordination
  - remote invocation
  - exception handling
  - efficiency
  - interoperability
- Specific concerns (behaviors)
  - sum
  - print
  - translate
  - pricing
  - DFT traversal
  - cycle checking

Aspects

Adaptive Plug and Play Components


```
interface ShapeI extends Remote {
  double get_x() throws RemoteException ;
  void   set_x(int x) throws RemoteException ;
  double get_y() throws RemoteException ;
  void   set_y(int y) throws RemoteException ;
  double get_width() throws RemoteException ;
  void   set_width(int w) throws RemoteException ;
  double get_height() throws RemoteException ;
  void   set_height(int h) throws RemoteException ;
  void   adjustLocation() throws RemoteException ;
  void   adjustDimensions() throws RemoteException ;
}
```

```
public class Shape
          implements ShapeI {
  protected AdjustableLocation loc;
  protected AdjustableDimension dim;
  public Shape() {
    loc = new AdjustableLocation(0, 0);
    dim = new AdjustableDimension(0, 0);
  }
  double get_x() throws RemoteException {
    return loc.x(); }
  void   set_x(int x) throws RemoteException {
    loc.set_x(); }
  double get_y() throws RemoteException {
    return loc.y(); }
  void   set_y(int y) throws RemoteException {
    loc.set_y(); }
  double get_width() throws RemoteException {
    return dim.width(); }
  void   set_width(int w) throws RemoteException {
    dim.set_w(); }
  double get_height() throws RemoteException {
    return dim.height(); }
  void   set_height(int h) throws RemoteException {
     dim.set_h(); }
  void   adjustLocation() throws RemoteException {
    loc.adjust();
  }
  void   adjustDimensions() throws RemoteException {
    dim.adjust();
  }
}
```

```
class AdjustableLocation {
  protected double x_,  y_;
  public AdjustableLocation(double x, double y) {
    x_ = x; y_ = y;
  }
  synchronized double get_x() { return x_; }
  synchronized void   set_x(int x) {x_ = x;}
  synchronized double get_y() { return y_; }
  synchronized void   set_y(int y) {y_ = y;}
  synchronized void adjust() {
    x_ = longCalculation1();
    y_ = longCalculation2();
  }
}
```

```
class AdjustableDimension {
  protected double width_=0.0, height_=0.0;
  public AdjustableDimension(double h, double w) {
    height_ = h; width_ = w;
  }
  synchronized double get_width() { return width_; }
  synchronized void   set_w(int w) {width_ = w;}
  synchronized double get_height() { return height_; }
  synchronized void   set_h(int h) {height_ = h;}
  synchronized void adjust() {
    width_ = longCalculation3();
    height_ = longCalculation4();
  }
}
```

thread synchronization

remote interaction

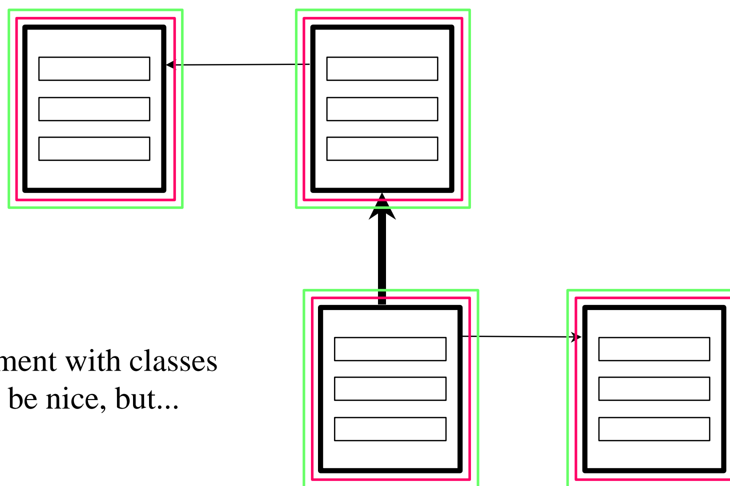
```
public class Shape
          implements ShapeI {
  protected AdjustableLocation loc;
  protected AdjustableDimension dim;
  public Shape() {
    loc = new AdjustableLocation(0, 0);
    dim = new AdjustableDimension(0, 0);
  }
  double get_x() throws RemoteException {
    return loc.x(); }
  void   set_x(int x) throws RemoteException {
    loc.set_x(); }
  double get_y() throws RemoteException {
    return loc.y(); }
  void   set_y(int y) throws RemoteException {
    loc.set_y(); }
  double get_width() throws RemoteException {
    return dim.width(); }
  void   set_width(int w) throws RemoteException {
    dim.set_w(); }
  double get_height() throws RemoteException {
    return dim.height(); }
  void   set_height(int h) throws RemoteException {
     dim.set_h(); }
  void   adjustLocation() throws RemoteException {
    loc.adjust();
  }
  void   adjustDimensions() throws RemoteException {
    dim.adjust();
  }
}
```

```
interface ShapeI extends Remote {
  double get_x() throws RemoteException ;
  void   set_x(int x) throws RemoteException ;
  double get_y() throws RemoteException ;
  void   set_y(int y) throws RemoteException ;
  double get_width() throws RemoteException ;
  void   set_width(int w) throws RemoteException ;
  double get_height() throws RemoteException ;
  void   set_height(int h) throws RemoteException ;
  void   adjustLocation() throws RemoteException ;
  void   adjustDimensions() throws RemoteException ;
}
```

```
class AdjustableLocation {
  protected double x_,  y_;
  public AdjustableLocation(double x, double y) {
    x_ = x; y_ = y;
  }
  synchronized double get_x() { return x_; }
  synchronized void   set_x(int x) {x_ = x;}
  synchronized double get_y() { return y_; }
  synchronized void   set_y(int y) {y_ = y;}
  synchronized void adjust() {
    x_ = longCalculation1();
    y_ = longCalculation2();
  }
}
```

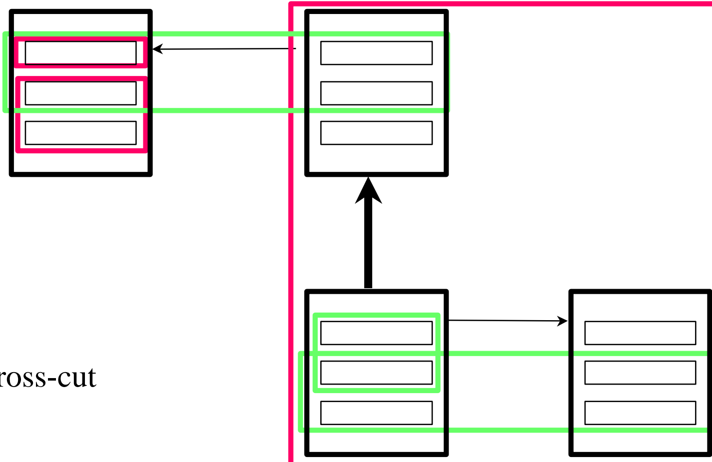
```
class AdjustableDimension {
  protected double width_=0.0, height_=0.0;
  public AdjustableDimension(double h, double w) {
    height_ = h; width_ = w;
  }
  synchronized double get_width() { return width_; }
  synchronized void   set_w(int w) {width_ = w;}
  synchronized double get_height() { return height_; }
  synchronized void   set_h(int h) {height_ = h;}
  synchronized void adjust() {
    width_ = longCalculation3();
    height_ = longCalculation4();
  }
}
```


The source of tangling



Alignment with classes would be nice, but...

The source of tangling



...issues cross-cut classes


```
interface ShapeI extends Remote {
  double get_x() throws RemoteException ;
  void   set_x(int x) throws RemoteException ;
  double get_y() throws RemoteException ;
  void   set_y(int y) throws RemoteException ;
  double get_width() throws RemoteException ;
  void   set_width(int w) throws RemoteException ;
  double get_height() throws RemoteException ;
  void   set_height(int h) throws RemoteException ;
  void   adjustLocation() throws RemoteException ;
  void   adjustDimensions() throws RemoteException ;
}
public class Shape
       implements ShapeI {
  protected AdjustableLocation loc;
  protected AdjustableDimension dim;
  public Shape() {
    loc = new AdjustableLocation(0, 0);
    dim = new AdjustableDimension(0, 0);
  }
  double get_x() throws RemoteException {
    return loc.x(); }
  void   set_x(int x) throws RemoteException {
    loc.set_x(); }
  double get_y() throws RemoteException {
    return loc.y(); }
  void   set_y(int y) throws RemoteException {
    loc.set_y(); }
  double get_width() throws RemoteException {
    return dim.width(); }
  void   set_width(int w) throws RemoteException {
    dim.set_w(); }
  double get_height() throws RemoteException {
    return dim.height(); }
  void   set_height(int h) throws RemoteException {
     dim.set_h(); }
  void   adjustLocation() throws RemoteException {
    loc.adjust();
  }
  void   adjustDimensions() throws RemoteException {
    dim.adjust();
  }
}
class AdjustableLocation {
  protected double x_,  y_;
  public AdjustableLocation(double x, double y) {
    x_ = x; y_ = y;
  }
  synchronized double get_x() { return x_; }
  synchronized void   set_x(int x) {x_ = x;}
  synchronized double get_y() { return y_; }
  synchronized void   set_y(int y) {y_ = y;}
  synchronized void adjust() {
    x_ = longCalculation1();
    y_ = longCalculation2();
  }
}
class AdjustableDimension {
  protected double width_=0.0, height_=0.0;
  public AdjustableDimension(double h, double w) {
    height_ = h; width_ = w;
  }
  synchronized double get_width() { return width_; }
  synchronized void   set_w(int w) {width_ = w;}
  synchronized double get_height() { return height_; }
  synchronized void   set_h(int h) {height_ = h;}
  synchronized void adjust() {
    width_ = longCalculation3();
    height_ = longCalculation4();
  }
}
```

Instead of writing this

D

Write this

```
public class Shape {
  protected double x_= 0.0,  y_= 0.0;
  protected double width_=0.0, height_=0.0;

  double get_x() { return x_(); }
  void   set_x(int x) { x_ = x; }
  double get_y() { return y_(); }
  void   set_y(int y) { y_ = y; }
  double get_width(){ return width_(); }
  void   set_width(int w) { width_ = w; }
  double get_height(){ return height_(); }
  void   set_height(int h) { height_ = h; }
  void   adjustLocation() {
    x_ = longCalculation1();
    y_ = longCalculation2();
  }
  void   adjustDimensions() {
    width_ = longCalculation3();
    height_ = longCalculation4();
  }
}

coordinator Shape {
  selfex adjustLocation, adjustDimensions;
  mutex {adjustLocation, get_x, set_x,
                         get_y, set_y};
  mutex {adjustDimensions, get_width, get_height,
                           set_width, set_height};
}

portal Shape {
  double get_x() {} ;
  void   set_x(int x) {};
  double get_y() {};
  void   set_y(int y) {};
  double get_width() {};
  void   set_width(int w) {};
  double get_height() {};
  void   set_height(int h) {};
  void   adjustLocation() {};
  void   adjustDimensions() {};
}
```

Code tangling is bad

- Harms program structure
- Distracts from main functionality

- Hard to program, error-prone
- Code difficult to understand, maintain

Ways to decrease the tangling

- Style guidelines
- Coding rules
- Design patterns
- **Better design and programming languages**
  - AOP
  - Adaptive Plug and Play Components

Tangling of traversal/visitor calls

class graph

A = B.
B = C.
C = D.
D = .

traversal + visitor calls

```
class A
void t(V v){
  v.before_b(this);
  b.t(v);
  v.after_b(this);}
class B
void t(V v){
  v.before_c(this);
  c.t(v);
  v.after_c(this); }
```

```
visitor class V
before_b(A host) {
  … }
after_b(A host) {
  …}
...
```


Tangling of behaviors V and W

class graph

A = B.
B = C.
C = D.
D = .

visitor class V
visitor class W

traversal + visitor calls

```
class A:
void f() {t(new V(),
  new W());}
void t(V v,W w){
  v.before_b(this);
  w.before_b(this);
  b.t(v);
  w.after_b(this);
  v.after_b(this);}
```

```
class B:
void t(V v,W w){
  v.before_c(this);
  w.before_c(this);
  c.t(v);
  w.after_c(this);
  v.after_c(this); }
```

At design level: class A:void f() to D (V,W)

Ways to decrease redundancy

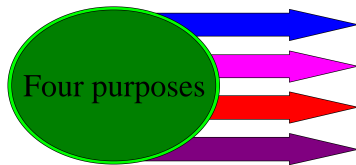
- Factor out repeated information
- Refer to it sparingly so that the factored-out information can be easily changed

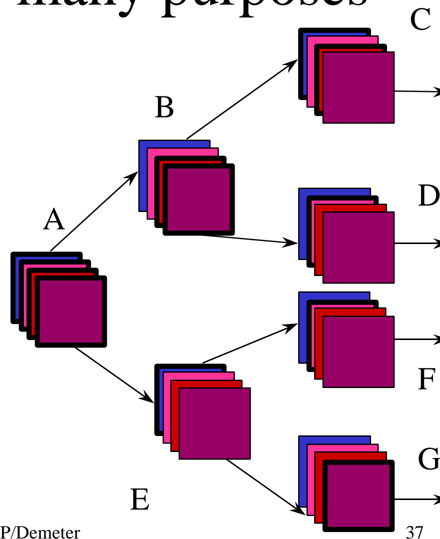
Objects serve many purposes

- 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)...

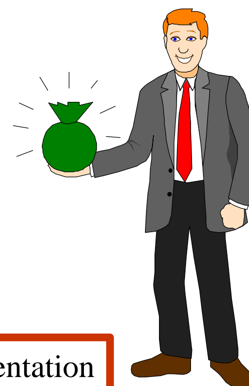




4/20/98 AOP/Demeter 37

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.

4/20/98 AOP/Demeter 38

Five Patterns

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

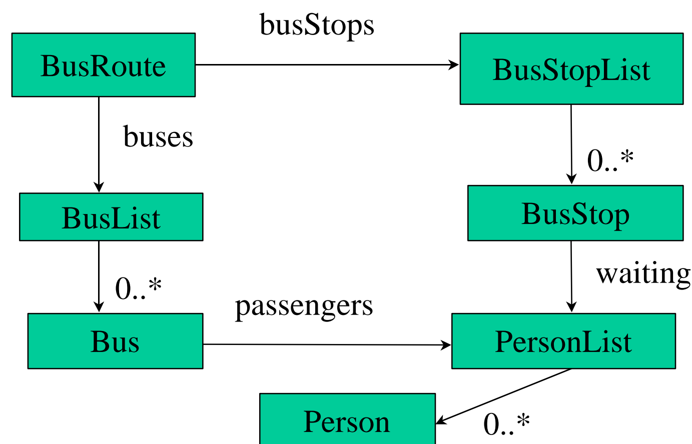
On-line information

- $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

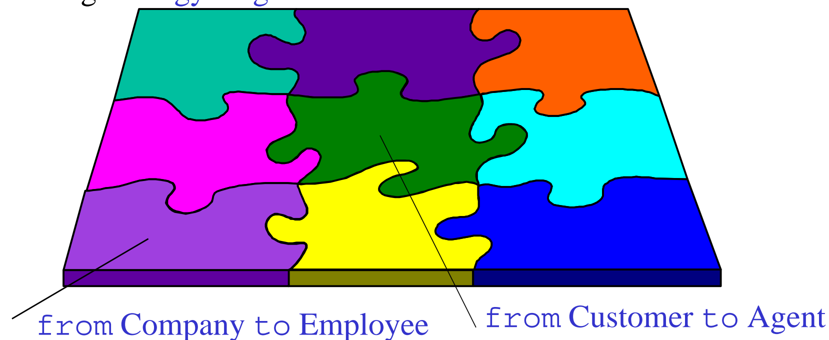


2: Traversals / Collaborating classes

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

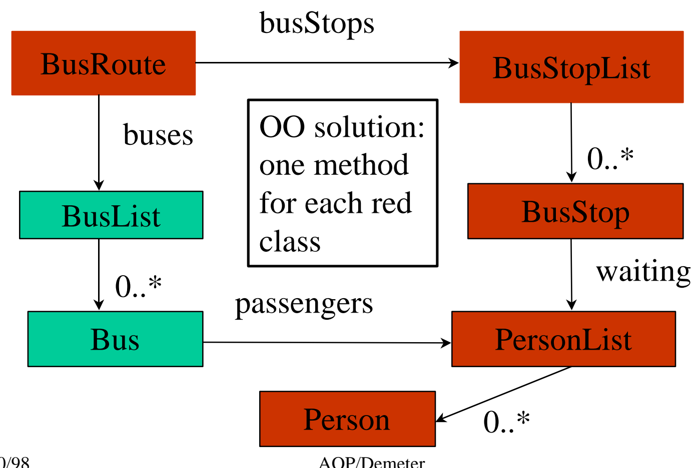
Collaborating Classes

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



Collaborating Classes

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

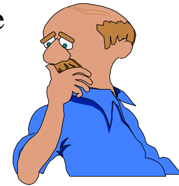


3: Traversal Strategy Graphs

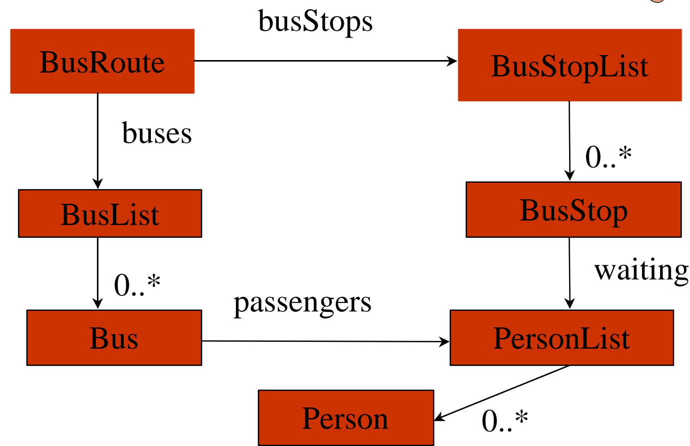
- Want to define traversals succinctly
- Use graph to express abstraction of class diagram
- Express traversal intent: useful for documentation of object-oriented programs

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

Traversal Strategy



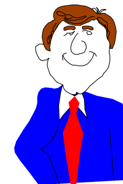
first try: from BusRoute to Person



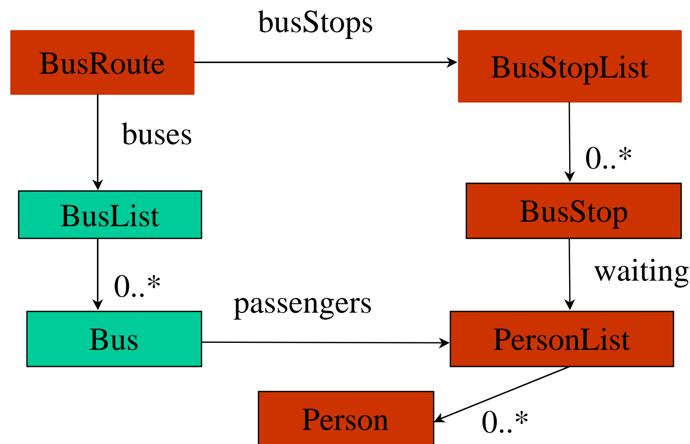
4/20/98 AOP/Demeter 47

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

Traversal Strategy



from BusRoute through BusStop to Person

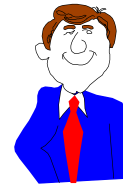


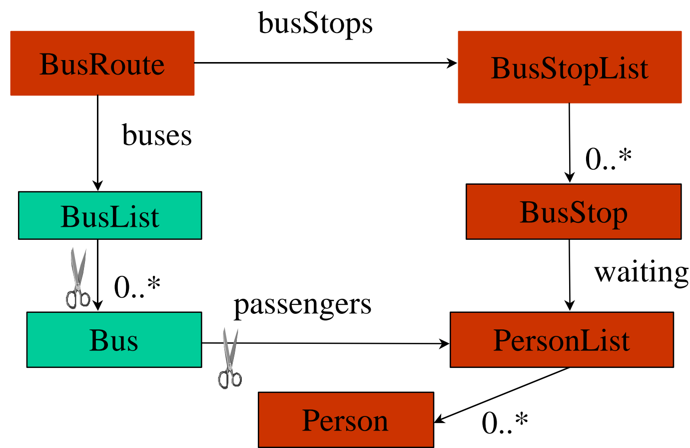
4/20/98 AOP/Demeter 48

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

Traversal Strategy

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



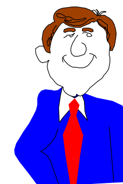


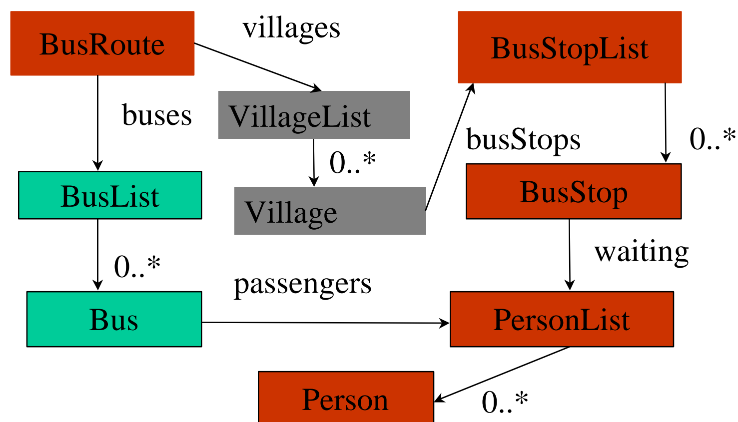
4/20/98 AOP/Demeter 49

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

Robustness of Strategy

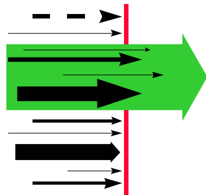
`from` BusRoute `bypassing` Bus `to` Person





4/20/98 AOP/Demeter 50

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

Families: Nature Analogy for AP

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

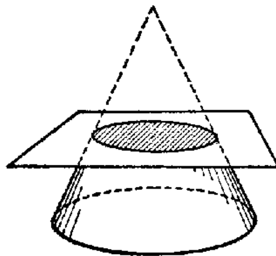
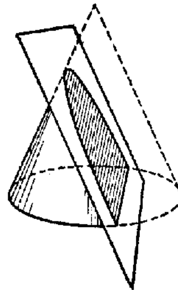
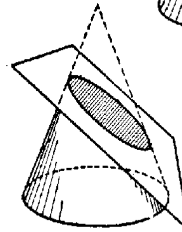
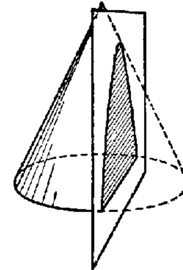
warm climate

cold climate

Families of systems

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

Mathematical Analogy for AP

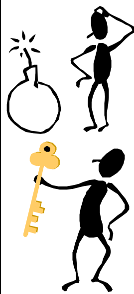





Why Traversal Strategies?

10 year anniversary

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

Widely used, for example, at JPL for the Mars exploration software.

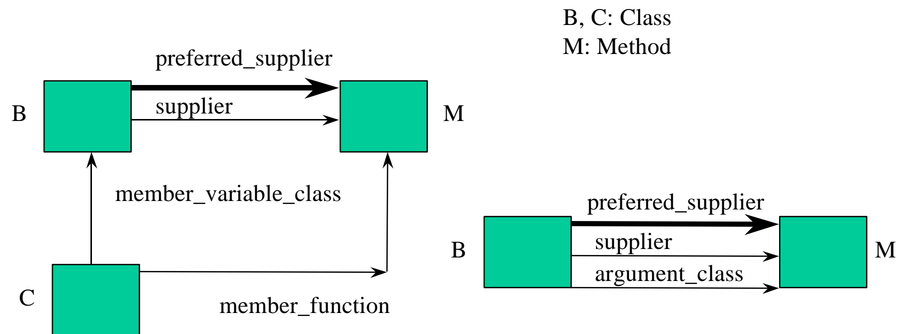


- Dilemma:
  - If followed: Small method problem of OO
  - If not followed: Unmaintainable code
- Traversal strategies are the solution to this dilemma

Graph patterns: for implementing patterns

Law of Demeter (simplified)

All suppliers should be preferred suppliers



OR

Software Architecture View

- Software architectures where connections necessary for a specific behavior are specified approximately, yet precisely, using regular expression-like constructs.
- Improves on conventional architectures by supporting structure-shyness allowing both simpler and more flexible architectures.

Software Architecture View

Behavioral architecture	Hierarchies of strategy graphs Strategic traversal automata Adaptive Plug and Play Components (APPCs)
Structural architecture	Class graphs
Control	Object graphs

Implementation of traversal strategies

- Based on novel applications and variations of standard techniques:
  - Intersection of non-deterministic finite automata
  - Simulation of non-deterministic finite automata

How to do the interesting work?

- Traversal strategies only navigate.
- APPCs specify what is done in addition to navigation: Components.
- Visitors: a special kind of APPC
  - change group of classes during a traversal
  - plug-and-play organization based on ports
  - parameterized by traversal strategies
  - tree organization

How to do the interesting work?

- APPCs (Adaptive Plug and Play Components)
  - Improvement to visitors
  - Use strategy types as an abstract class graph
  - Extend or modify methods of a group of classes
  - Can be composed easily
- In the following we focus on visitors
  - Are close to oo languages
  - Positive experience with visitors in Demeter/Java

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

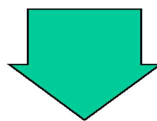
Class Diagrams

define family of

Object Diagrams

Adaptive Programming

Strategy Diagrams

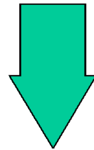


define traversals of

Object Diagrams

Adaptive Programming

Strategy Diagrams



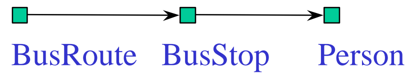
guide and inform

Visitors

AP

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

Strategy Diagrams



BusRoute BusStop Person

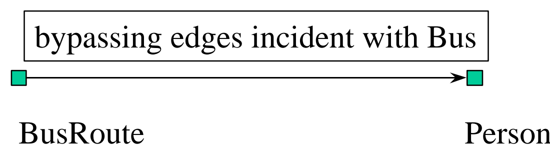
Nodes: positive information: Mark corner stones in class diagram: Overall topology of collaborating classes. 3 nodes:

`from` BusRoute

`through` BusStop

`to` Person

Strategy Diagrams



BusRoute Person

Edges: negative information:
Delete edges from class diagram.

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

5: Tools for Aspect-Oriented and Adaptive Programming

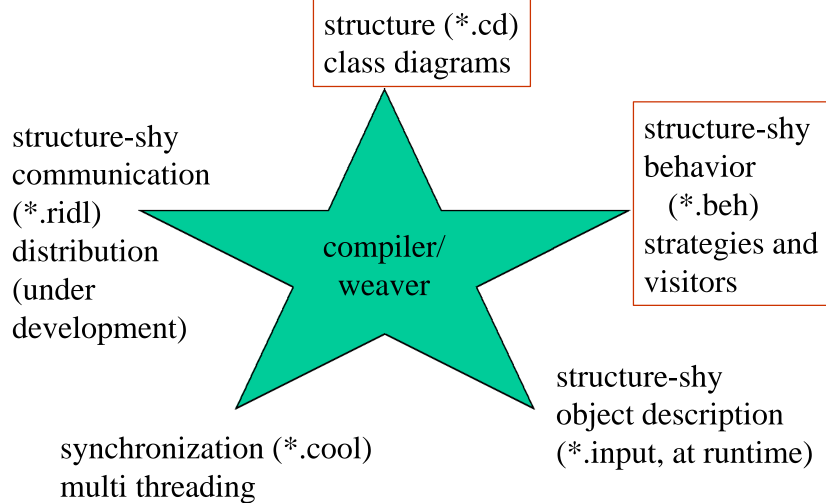
- many free tools available, including Aspect/J from Xerox PARC
- one commercial tool which uses a point and-click interface to define traversals (StructureBuilder from Tendril)

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



4/20/98 AOP/Demeter 71

Tools using Demeter ideas: Demeter/C++, Dem/CLOS (BBN), Demeter/Perl5 (MIT), AspectJ (Xerox PARC)

Goal of Demeter/Java

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

4/20/98 AOP/Demeter 72

Free Tools on WWW

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

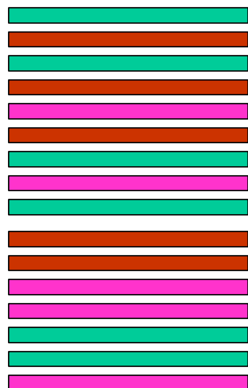
last five developed outside our group

Aspect/J from Xerox

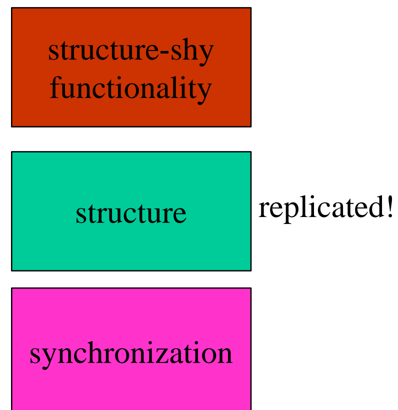
4/20/98 AOP/Demeter 73

Cross-cutting in Demeter/Java

generated Java program



Demeter/Java program



replicated!

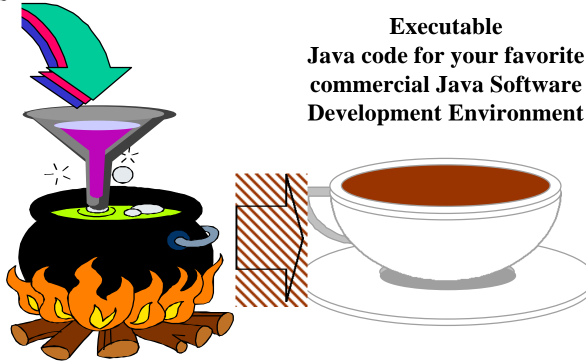
4/20/98 AOP/Demeter 74

Demeter/Java

www.ccs.neu.edu/research/demeter

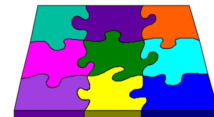
- class diagrams
- functionality
  - strategies
  - visitors
- etc.

weaver



Executable Java code for your favorite commercial Java Software Development Environment

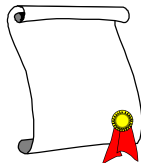
AP Studio



- 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



Success indicators

- Used in several commercial projects (HP (printer family installation), GTE (compiler), Motorola (pattern generator), Novell (schema comparator)
- AspectJ from Xerox PARC (EDCS project) based on Cristina Lopes Ph.D. thesis (1998) at Northeastern University supported by Xerox PARC.

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; @)
```

4/20/98 AOP/Demeter 87

Using Demeter/COOL, put into *.cool file

Coordinator

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


4/20/98 AOP/Demeter 88

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

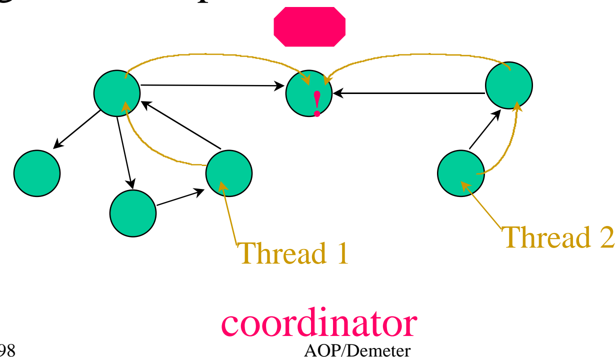
- The smallest unit of synchronization is the method.
- The provider of a service defines the synchronization (monitor approach).
- Coordination is contained within one coordinator.
- Association from object to coordinator is static.

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

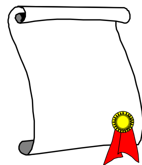
```
coordinator Shape {
  selfex {adjustLocation,
          adjustDimensions}
  mutex {adjustLocation,x}
  mutex {adjustLocation,y}
  mutex {adjustDimensions,
          width}
  mutex {adjustDimensions,
          height}
}
```


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

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

History

- First traversal specifications

1990

- Separation of Concerns paper by Huersch and Lopes started "untangling" movement TR NU-CCS-95-03. Collaboration with Xerox PARC started (initiated in 1994).

1995

- Gregor Kiczales and his group name and further develop AOP.

1996

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.

Pattern Language for Adaptive Programming (AP)

Karl Lieberherr
Northeastern University

Introduction

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

Summary

- Present ideas of AP at a high-level of abstraction.
- Explain concepts independent of tools and languages.

Vocabulary

- Pattern: Reusable solution to a problem in a context.
- Class graph = Class diagram: Graph where nodes are classes and edges are relationships between the classes.
- Design pattern book: Gamma, Helm, Johnson, Vlissides: 23 design patterns

Vocabulary

- Visitor pattern: Define behavior for classes without modifying classes.
- Parser: Takes a sequence of tokens and creates a syntax tree or object based on a grammar.
- Grammar: a class graph annotated with concrete syntax.

Overview

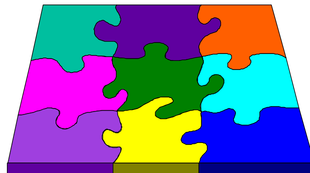
- Patterns a useful way to write down experience.
- Use a standard format: Intent, Motivation, Applicability, Solution, Consequences, etc.
- Patterns are connected and refer to each other.
- Extended version at: $D/adaptive-patterns/pattern-lang-conv

Connections

- There are several connections between the AP patterns and other design patterns.
- Class Graph is the basis for Structure-shy Traversal, Selective Visitor and Structure-shy Object.

Structure-shy Traversal

- Intent
  - Succinctly represent a traversal to be performed on objects
  - Commit only to navigation strategy and specify navigation details later



Solve Law of Demeter Dilemma

Small Method Goat

Big Method Goat

Structure-shy Traversal

- Could also be called:
  - Adaptive Traversal
  - Structure-shy Walker
  - Adaptive Visitor (significantly improves the Visitor pattern)

Structure-shy Traversal

- Motivation
  - Noise in objects for specific task
  - Focus on long-term intent
  - Don't want to attach every method to a specific class explicitly. Leads to brittle programs.
  - Small methods problem (example: 80% of methods are two lines long or shorter)

Structure-shy Traversal

- Applicability
  - Need collaboration of at least two classes.
  - In the extreme case, each data member access is done through a succinct traversal specification.
  - Some subgraphs don't have a succinct representation, for example a path in a complete graph. More generally: avoid well connected, dense graphs.

Structure-shy Traversal

- Solution
  - Use succinct subgraph specifications
  - Use succinct path set specifications

Structure-shy Traversal: Solution

- Traversal Strategy Graphs (Strategies)
  - *First stage*: A strategy is a graph with nodes and edges. Nodes are labeled with nodes of a class graph. Edges mean: all paths.

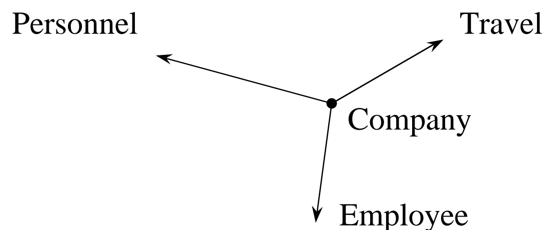
  - *Second stage*: label edges with constraints excluding edges and nodes in class graph
  - *Third stage*: Encapsulated strategies. Use symbolic elements and map to class graph.

Structure-shy Traversal: Solution

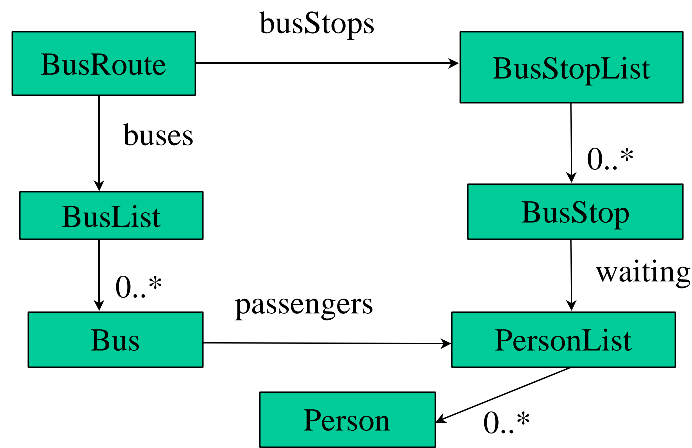
- Traversal Strategy Graphs (Strategies)
  - Simplest useful strategy: One Edge. Possible syntax:
    - from Company to Salary or
    - {Company -> Salary}
  - Line graph. Several edges in a line. Possible syntax:
    - From Company via Employee to Salary
    - {Company -> Employee, Employee -> Salary}

Structure-shy Traversal: Solution

- Traversal Strategy Graphs (Strategies)
  - Star graph
    - From Company to {Personnel, Travel, Employee}

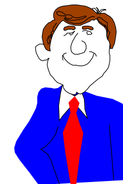


UML Class Diagram

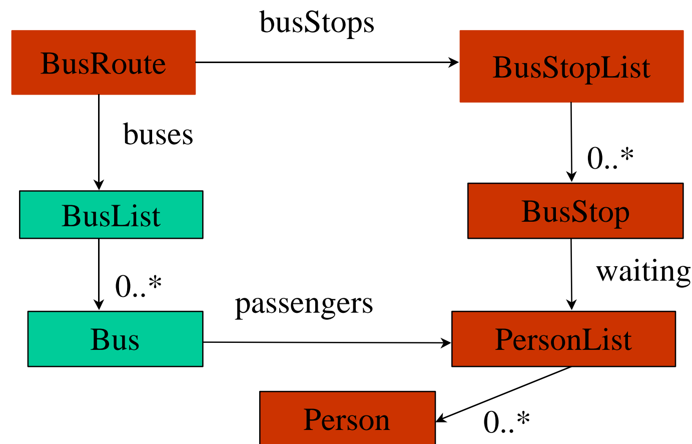


find all persons waiting at any bus stop on a bus route

Traversal Strategy

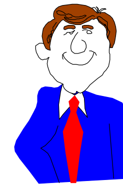


`from` BusRoute `through` BusStop `to` Person

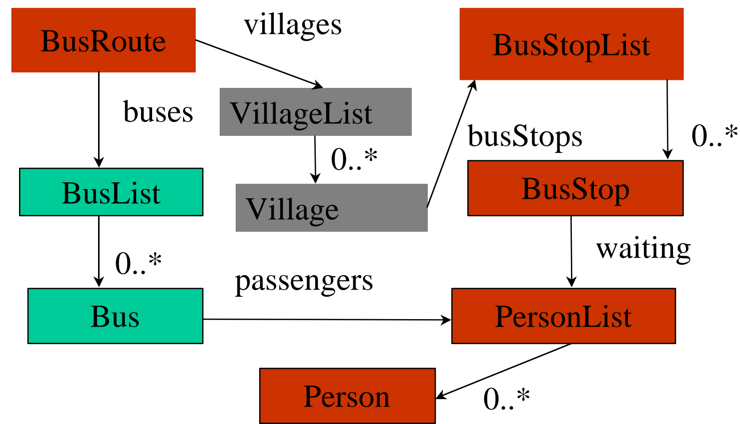


find all persons waiting at any bus stop on a bus route

Robustness of Strategy



from BusRoute through BusStop to Person



Structure-shy Traversal

- Consequences
  - Programs become shorter and more powerful. A paradox. With less work we achieve more. Polya's inventor paradox.
  - Program will adapt to many changes in class structure.

Structure-shy Traversal

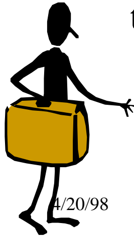
- Implementation
  - Many different models for succinct traversal specifications.
  - Best one: Strategies
  - Correct implementation of strategies is tricky. See paper by Lieberherr/Patt-Shamir strategies.ps in my FTP directory.

Structure-shy Traversal

- Known Uses
  - *Adaptive Programming*: Demeter/C++, Demeter/Java, Dem/Perl, Dem/CLOS etc.
  - *Databases* (limited use): Structure-shy queries: See Cole Harrison's Master's Thesis (Demeter Home Page)
  - *Artificial Intelligence* (limited use): Minimal ontological commitment

Selective Visitor

- Intent
  - Loosely couple behavior modification to behavior and structure.
  - Would like to write an editing script to modify traversal code instead of modifying the traversal code manually.



Selective Visitor

- Could also be called:
  - Structure-shy Behavior Modification
  - Event-based Coupling

Selective Visitor

- Motivation:
  - Avoid tangling of code for one behavior with code for other behaviors.
  - Localize code belonging to one behavior.
  - Compose behaviors.
  - Modify the behavior of a traversal call (traversals only traverse).

Selective Visitor

- Applicability:
  - Need to add behavior to a traversal.

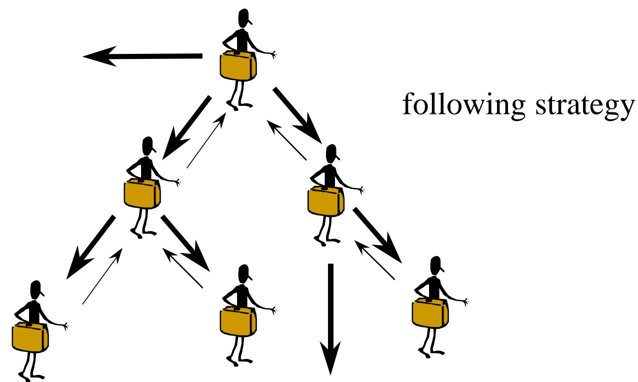
Selective Visitor

- Solution:
  - Use visitor classes and objects.
  - Pass visitor objects as arguments to traversals.
  - Either use naming conventions for visitor methods (e.g., before_A()) or extend object-oriented language (e.g. before A, before is a new key word).

Selective Visitor

- Solution:
  - before, after methods for nodes and edges in the class graph
  - Activated during traversal as follows:
    - Execute before methods
    - Traverse
    - Execute after methods

Visitor visits objects



Visitor collects information in suitcase (variables)

Selective Visitor

- Solution: Focus on what is important.

```
SummingVisitor {
 (@ int total; @)
 init (@ total = 0; @)
 before Salary (@ total = total + host.get_v(); @)
 return (@ total @)
}
```

host is object visited

Code between (@ and @) is Java code

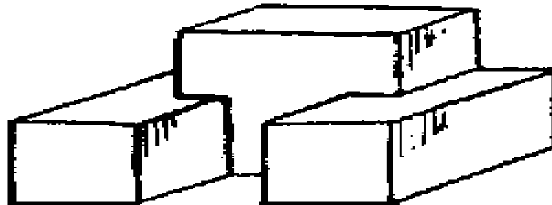
Selective Visitor

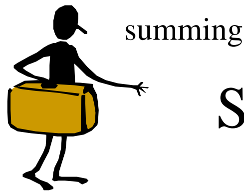
- Solution: Use of visitor

```
Company {
 traversal allSalaries(UniversalVisitor) {do S;}
 (@ int sumSalaries() {
  SummingVisitor s = new SummingVisitor();
  this.allSalaries(s);
  return s.get_return_val();
 }@)
}
```


Selective Visitor

- Consequences
  - Easy behavior adjustments: Add visitor
  - Reuse of visitors





Selective Visitor

- Consequences: Easy behavior enhancement

```
Company { // enhancements in red
 traversal allSalaries(UniversalVisitor, UniversalVisitor)
  {do S;}
 (@ float averageSalaries() {
  SummingVisitor s = new SummingVisitor();
  CountingVisitor c = new CountingVisitor();
  this.allSalaries(s, c);
  return s.get_return_val() / c.get_return_val();
 }@)
}
```


Writing Programs with Strategies

Example of Adaptive Program

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.

Selective Visitor

- Consequences:
  - Can reuse SummingVisitor and CountingVisitor in other applications.

Selective Visitor

- Implementation
  - Translate to object-oriented language.
  - See Demeter/Java, for example.

Selective Visitor

- Known uses
  - Propagation patterns use inlined visitor objects (see AP book).
  - Demeter/Java.
  - The Visitor Design Pattern from the design pattern book uses a primitive form of Selective Visitor.

Differences to Visitor pattern

- Focus selectively on important classes. Don’t need a method for each traversed class.
- Finer control: not only one accept method but before and after methods for both nodes and edges.

Structure-shy Object

- Intent
  - Make object descriptions for tree objects robust to changes of class structure.
  - Make object descriptions for tree objects independent of class names.

Structure-shy Object

- Could also be called:
  - Object Parsing
  - Grammar
  - Abstract=Concrete Syntax

Structure-shy Object

- Motivation
  - Data maintenance a major problem when class structure changes
  - Tedious updating of constructor calls
  - The creational patterns in the design pattern book also recognize need
  - Concrete syntax is more abstract than abstract syntax!

Structure-shy Object

- Applicability
  - Useful in object-oriented designs of any kind.
  - Especially useful for reading and printing objects in user-friendly notations. Ideal if you control notation.
  - If you see many constructor calls: think of Structure-shy Object.

Structure-shy Object

- Solution
  - Extend the class structure definitions to define the syntax of objects.
  - Each class will define a parse function for reading objects and a print visitor for printing all or parts of an object.

Structure-shy Object

- Solution
  - Start with familiar grammar formalism and change it to make it also a class definition formalism. In Demeter we use Wirth's EBNF formalism.
  - Use a parser generator (like YACC or JavaCC) or a generic parser.

Structure-shy Object

Parsers weave sentences into objects

Problem in OO programs: Constructor calls for compound objects are brittle with respect to structure changes.

Solution: Replace constructor calls by calls to a parser. Annotate class diagram to make it a grammar.

Benefit: reduce size of code to define objects, object descriptions are more robust

Correspondence: Sentence defines a family of objects. Adaptive program defines family of object-oriented programs. In both cases, family member is selected by (annotated) class diagram.

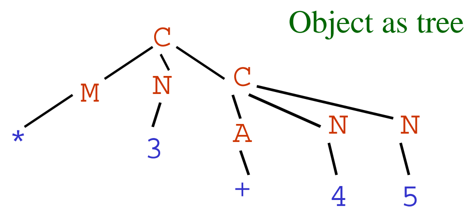
Structure-shy Object

Run-time weaving: Description

Sentence
* 3 + 4 5

Grammar
Compound ...
Simple ...
Number ...
Multiply ...
Add ...
etc.

Object as tree



Object in linear form (Constructor calls)

C M * N 3 C A + N 4 N 5

SENTENCE IS MORE ROBUST THAN OBJECT

Grammar defined by annotating UML class diagram

Parsing

- class graphs
- class dictionaries = class graphs + syntax
- legal objects
- printing function
- defines language defined by class dictionary
- ambiguous class graph: two distinct objects are mapped to same sentence

Class graphs

- construction, alternation nodes and edges
- textual and graphical
- common parts, flat class graphs
- optional parts and repetition
- parameterized classes

Parsing

- Is a class dictionary ambiguous?
- If not ambiguous, can define inverse of printing function: called parsing
- Goal: Want a fast parser. Want to quickly recognize an ambiguous class dictionary.

Parsing

- Inherent problem: There is no algorithm to check for ambiguity. Shown by reduction: Could solve other undecidable problems.
- Invent condition which is stronger than non-ambiguity and which can be checked efficiently. Ok. to exclude some non-ambiguous grammars.
- LL(1) conditions.

Parsing

- LL(1) condition 1: Alternatives of abstract class must have unique first sets.
- LL(1) condition 2: Deals with alternatives which may be empty. Requires follow sets.
- LL(1) conditions can be checked efficiently.
- Printing is a bijection between tree objects and sentences.

Commercial parsing tools

- Yacc and Lex
- JavaCC (Java compiler compiler)
  LL(k) support
- JJTree: define classes using a grammar and provides a universal visitor
- many more

Structure-shy Object

- Consequences
  - more robust and shorter object descriptions
  - Need to deal with *unique readability with respect to an efficient parsing algorithm*
  - Can guarantee unique readability by adding more syntax
  - debug class structures by reading objects

Structure-shy Object

- Related patterns
  - Creational patterns in design pattern book.
  - Interpreter pattern uses similar idea but fails to propose it for general object-oriented design.
  - Structure-shy Object useful in conjunction with Prototype pattern.

Structure-shy Object

- Known uses
  - Demeter Tools since 1986, T-gen, applications of YACC, programming language Beta and many more.

Structure-shy Object

- References
  - Chapters 11 and 16 of AP book describe details.
- Exercise
  - Use your favorite grammar notation and modify it to also make it a class graph notation.

Class Graph

- Intent
  - Write class relationships once and reuse them many times.
  - Generate a visitor library from class graph for copying, displaying, printing, checking, comparing and tracing of objects.

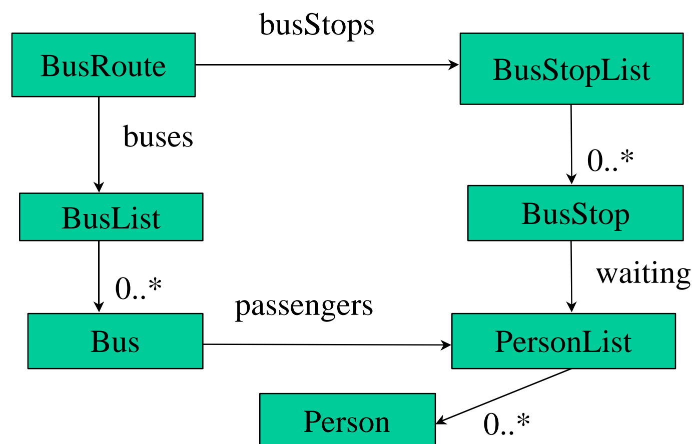
Class Graph

- Could also be called:
  - Class diagram
  - Class dictionary

Class Graph

- Applicability
  - For every application having more than one class.
- Implementation
  - Preferred: Use UML class graph model and notation (becoming a standard)
  - Use tool to generate visitor library (see Demeter/Java).

UML Class Diagram



Class Graph

- Known uses:
  - Almost all object-oriented design methods use some form of class diagram. Only Demeter/Java generates visitor library and allows strategies to refer to the class graph.
- References
  - UML class graphs, see UML books
  - Demeter class graphs, see chapter 6 of AP book

Growth plan pattern

- Intent
  - Build your adaptive programs incrementally. Use structural and behavioral simplifications which allow, ideally, for growth by addition and refinement.
- Could also be called:
  - Evolutionary development

Growth plan

- Motivation: It is useful to have at all times a simplified version of the program running.
  - good for your self-confidence
  - good for your customers: feedback

  Build applications in growth phases, where a phase *next* can do more than a previous phase *previous*.

Growth plan

- Motivation (continued): We want to build *next* as much as possible out of *previous* by adding or refining, not by modifying *previous*. *next* ideally reuses the test inputs of *previous.*
- Application: Use this pattern when you build applications involving more than a small number of classes (say 5).

Growth plan

- Solution: A good strategy is to build a relative big chunk of the class dictionary of the application and to test it with several input sentences. Then build a structural shrinking plan (whose inverse is a structural growth plan) consisting of a decreasing (in size) sequence of class dictionaries.

Growth plan

- Solution (continued): For the smallest class dictionary you should be able to implement some interesting behavior. For each phase in the structural growth plan you implement increasingly more complex behavior. The structural growth steps should fall into one or both of the following categories:

cd = class dictionary

Growth plan

- Solution (continued):
  - weakly object extending cd transformations
    - The next class dictionary defines more objects but does not invalidate any existing objects. What runs now should run later. Reuse of test objects.
  - language extending cd transformations
    - The next cd defines a super language of the language of the current cd.

Object-extending transformations

- relations on class graphs, associated with transformations, fundamental for reuse
  - object-equivalence
    - preserves the set of objects
  - weak extension
    - enlarges the set of objects
  - extension
    - enlarges and augments the set of objects

Part clusters

- What can be put into parts?
- PartClusters of a class *v* is a list of pairs, one for each induced part of *v*. Each pair consists of the part name and the set of construction classes whose instances can be assigned to the part
- $PartClusters_{Furnace}$(TempSensor) = {temp {Kelvin, Celsius}, trigger {Integer}}

Object-equivalence

- Let G_1 and G_2 be two class graphs. G_1 is object-equivalent to G_2 if for the concrete classes VC_1 of G_1 and the concrete classes VC_2 of G_2:
  - $VC_1 = VC_2$
  - and for all *v* in VC_1:

$$PartClusters_{G1}(v) = PartClusters_{G2}(v).$$

Covered

- Let PC_1 and PC_2 be two part clusters. PC_1 is covered by PC_2 if for each pair (l,T_1) in PC_1 there exists a pair (l,T_2) in PC_2 such that $T_1 \subseteq T_2$.
- Tightly covered means: covered and $|PC_1| = |PC_2|$.

Weak extension

- Let G_1 and G_2 be two class graphs. G_1 is a weak extension of G_2 if for the concrete classes VC_1 of G_1 and the concrete classes VC_2 of G_2:
  - $VC_1 \subseteq VC_2$ and for all v in VC_1:
  - $PartClusters_{G1}(v)$ is tightly covered by $PartClusters_{G2}(v)$.

Extension

- Let G_1 and G_2 be two class graphs. G_1 is an extension of G_2 if for the concrete classes VC_1 of G_1 and the concrete classes VC_2 of G_2:
  - $VC_1 \subseteq VC_2$ and
  - for all v in VC_1: $PartClusters_{G1}(v)$ is covered by $PartClusters_{G2}(v)$.

Properties

- The three class graph relations have the following inclusion properties:
  - object-equivalence $\subseteq$
  - weak-extension $\subseteq$
  - extension

Primitive Transformations

- Addition of Abstract Class (AddA)
- Deletion of Abstract Class (DelA)
- Abstraction of Common Reference (AbsR)
- Distribution of Common Reference (DisR)
- Replacement of Reference (RepR)

object-equivalence = DelA* AbsR* RepR* DisR* AddA*.

Primitive Transformations

- Addition of Abstract Class (AddA)
  - adds an abstract class *u* and subclass edges outgoing from *u*. *u* must not have any outgoing construction edges.
- Deletion of Abstract Class (DelA)
  - inverse of AddA. Deletes an abstract class *u* and all its subclass edges. *u* must not have any incoming construction or subclass edges nor any outgoing construction edges.

Primitive Transformations

- Abstraction of Common Reference (AbsR)
  - moves a construction edge common to a set of sibling classes up to their direct superclass.
- Distribution of Common Reference (DisR)
  - moves a construction edge to the direct subclasses.

Primitive Transformations

- Replacement of Reference (RepR)
  - reroutes a construction edge (v,l,u_1) to a new target (v,l,u_2) where u_1 and u_2 have the same set of concrete subclasses.

Primitive Transformations

- Addition of Concrete Class (AddC)
- Generalization of Reference (GenR)
- Addition of Reference (AddR)
- Equiv = object equivalence

weak-extension =
(Equiv((GenR)Equiv)*AddC*

extension =
(Equiv((AddR|GenR)Equiv)*AddC*

Primitive Transformations

- Addition of Concrete Class (AddC)
  - adds an “empty” concrete class
- Generalization of Reference (GenR)
  - reroutes a construction edge (v,l,u_1) to a new target (v,l,u_2), where u_2 is a direct superclass of u_1.

Primitive Transformations

- Addition of Reference (AddR)
  - adds a new construction edge between existing vertices of the class graph.

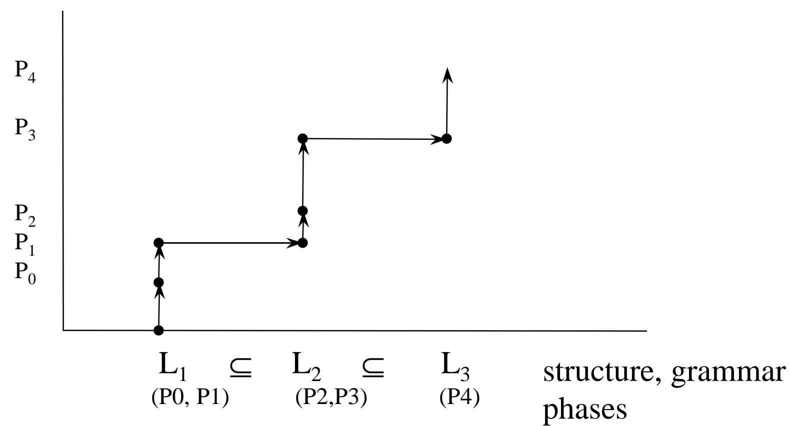
Connections

- weak extension implies language extension
  - If two cds are in a weakly object-extending relationship they can be brought to a language extending relationship with appropriate syntax.
- object-equivalent extension implies language-equivalent extension
  - similar

Growth plan

- behavior transformations should ideally extend the program by addition instead of modifying it:
  - use inheritance between visitor classes
  - add methods, traversals, visitors
  - refine methods and visitors
  - refine strategies (add more constraints)

Growth plan



Growth plan

- Consequences: Following Growth plan has a number of benefits:
  - Gradual building of confidence in your software development skills.
  - Show prototypes to your customers.
  - Simplified testing. Find earliest phase where a bug shows up.
  - faster compilation and generation

Growth plan

- Implementation: Create separate directories for each growth phase. Document changes.

  See chapter 13 in AP book for study of class graph extension. Also follow the pages listed under index entry *growth plan*.

Example

Same adaptive program for Terminal Buffer Rule checking works for both phases. Faster for phase 1.

phase 1

```
Cd_graph = <first> Adj.
Adj = <vertex> Vertex <ns> Construct ".".
Construct = "=" <l1> Labeled_vertex <l2> Labeled_vertex.
Labeled_vertex = "<" <label_name> Ident ">"
  <class_name> Vertex.
Vertex = <name> Ident.
```

phase 2

```
Cd_graph = <first> Adj <rest> Adj_list.
Adj = <vertex> Vertex <ns> Neighbors ".".
Neighbors : Construct | Alternat.
Construct = "=" <c_ns> Any_vertex_list.
Labeled_vertex = "<" <label_name> Ident ">"
  <class_name> Vertex.
Vertex = <name> Ident.
Any_vertex_list = <first> Any_vertex <rest> Any_vertex_list.
Any-Vertex : Labeled_vertex | Syntax_vertex.
...
```


Further information

- Paul Bergstein's OOPSLA 91 paper
- Walter Huersch's Ph.D. thesis
- Linda Seiter's Ph.D. thesis

Summary

- State what has been learned: Principles of AP in high-level form.
- How to apply: Do homework one and recognize those patterns in the thousands of lines Java code. See $D/course/f97/hw/1

Feedback

- Please see me after class or send me email if you have improvements to those patterns.
- lieberherr@ccs.neu.edu