Overview

- Structure-shy Object pattern
- Better separation of concerns using Structure-shy Traversal and Selective Visitor
- Cover now more details

Structure-shy object pattern

- Parsing, class graphs etc.
- separation of concerns:
  - sentences : robust representation
  - objects : brittle representations
  - grammars: rules for constructing brittle rep. from robust rep.
- XML, JSR 31

Parsing

- class graphs
- class dictionaries = class graphs + syntax
- legal objects
- printing function
- defines language defined by class dictionary
- ambiguous class dictionary: 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

Exercise

- Derive a class graph for explaining concrete class structure and syntax.

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 unambiguous 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 pair-wise disjoint 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

Context switch: Separation of concerns

- An introduction to aspect-oriented programming

Better separation of concerns

•An old topic: Dijkstra 1976
•Minimize tangling

Two types of concerns

- Generic concerns
  - structure
  - coordination
  - remote invocation
  - exception handling
  - efficiency
  - interoperability
  - migration
- Behaviors
  - sum
  - check
  - print
  - translate
  - Pricing
  - FrequentCustomerPricing

```
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();
  }
}
```


thread synchronization
remote interaction

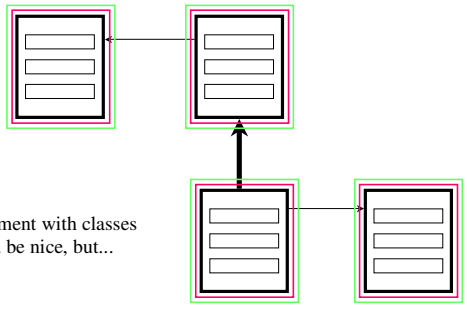
```
public class Shape
        implements ShapeI {
  protected AdjustableLocation loc;
  protected AdjustableDimension dim;
  public Shape() {
    loc = new AdjustableLocation(0, 0);
    dim = new AdjustableDimension(0, 0);
  }
  double get_x() throws RemoteException {
    return loc.x(); }
  void   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();
  }
}
```

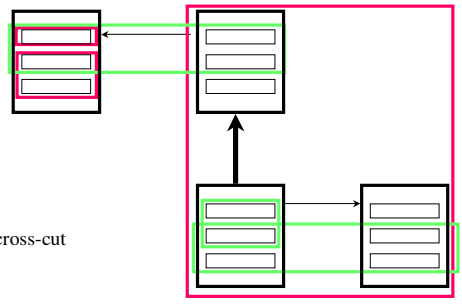
```
class AdjustableDimension {
  protected double width_=0.0, height_=0.0;
  public AdjustableDimension(double h, double w) {
    height_ = h; width_ = w;
  }
  synchronized double get_width() { return width_; }
  synchronized void   set_w(int w) {width_ = w;}
  synchronized double get_height() { return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


The source of tangling

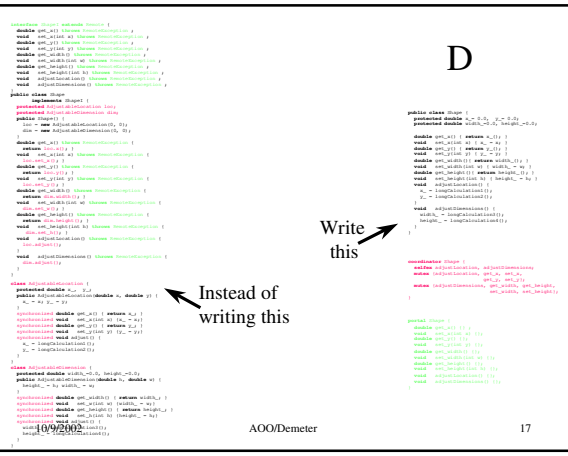


Alignment with classes would be nice, but...

The source of tangling



...issues cross-cut classes



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
- Design patterns
- Better programming languages

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);
  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);
  w.after_c(this);
  v.after_c(this); }
```

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

AOP => control of tangling

- DemeterJ controls the following tangling problems
  - structure/behavior,
  - navigation/behavior,
  - behavior/behavior,
  - object construction/structure,
  - synchronization/behavior,
  - remote invocation/behavior.

Context switch

Developing an adaptive program

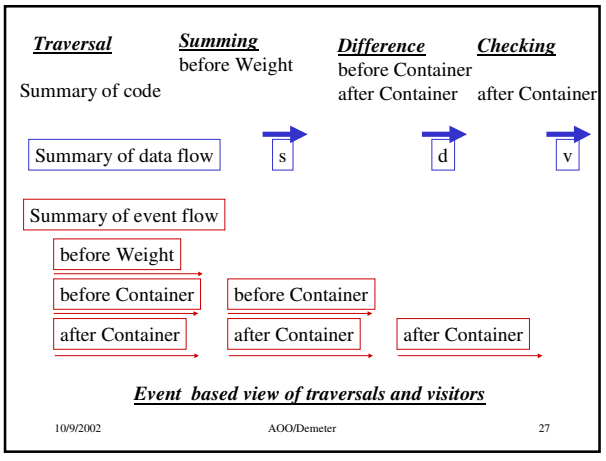
- each simple behavior is translated into a visitor; localizes concerns; visitors are self-contained
- compose visitors, order visitors
- use top-down approach for writing program: first write method using visitors before visitors are implemented

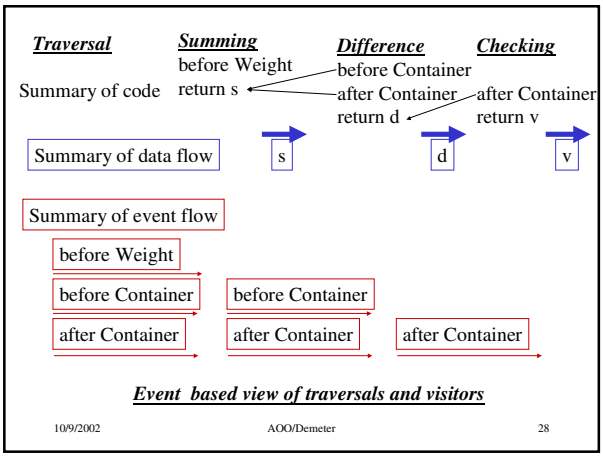
Container capacity problem

- recursive containers
- each contains items and has capacity
- each item has a weight
- find all containers above capacity
- leave details of containers vague

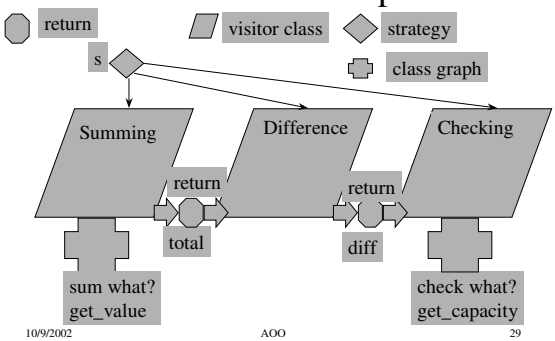
Container capacity problem

- need to sum: use SummingVisitor
- need to check: use CheckingVisitor
- want to do it in one traversal: need to store sum when container is entered to compute difference after the container: DifferenceVisitor



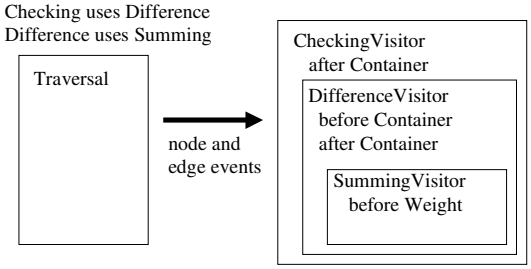


Visitor classes as components



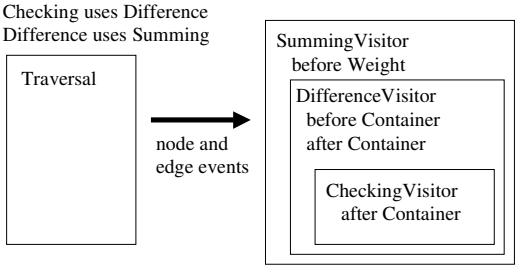
Which Conceptual View? Most basic innermost

Checking uses Difference
Difference uses Summing



Which Conceptual View? Most basic outermost: filter

Checking uses Difference
Difference uses Summing



Traversals are iterative

- from Container to Weight:
  total = total + host.get_i();
- some recursive computations need extra attention using difference visitors

Container capacity problem

- SummingVisitor
  - data member total, initialize to 0
  - add at Weight

Container capacity problem

- CheckingVisitor
  - data member: violations
  - after each container, check whether difference of sum at the end and the beginning of a container (supplied by DifferenceVisitor) is larger than capacity
  - uses DifferenceVisitor
  - return violations

Container capacity problem

- DifferenceVisitor
  - store sum at beginning (SummingVisitor) of container
  - return difference between sum at beginning of container and current sum upon request
  - uses SummingVisitor twice per container

Container capacity problem

- Preliminary visitor class graph!

  CheckingVisitor = <dV> DifferenceVisitor.
  DifferenceVisitor = <sV> SummingVisitor.
- Visitor order: after container, both DifferenceVisitor and CheckingVisitor are active.
- DifferenceVisitor must be after CheckingVisitor (because after methods are in reverse order)

DemeterJ

Container capacity problem

```
traversal allWeights(CheckingVisitor cV, DifferenceVisitor dV,
      SummingVisitor sV) {to Weight;}
int checkCapacity() {{
  CheckingVisitor cV = new CheckingVisitor();
  DifferenceVisitor dV = new DifferenceVisitor();
  SummingVisitor sV = new SummingVisitor();
  cV.set_dV(dV); dV.set_sV(sV); //link behaviors
  this.allWeights(cV, dV, sV); // call traversal with visitors
  return (cV.get_return_val());  }}
```

cannot avoid all tangling, but can localize it. No longer spread over several classes.

DJ

Container capacity problem

```
int checkCapacity(TraversalGraph whereToGo) {
  CheckingVisitor cV = new CheckingVisitor();
  DifferenceVisitor dV = new DifferenceVisitor();
  SummingVisitor sV = new SummingVisitor();
  cV.set_dV(dV); dV.set_sV(sV); //link behaviors
  Integer res = (Integer) whereToGo.traverse(this,
    new Visitor[]{cV, dV,sV});
  return res.intValue(); }
```

cannot avoid all tangling, but can localize it. No longer spread over several classes.

Container capacity problem

- Refined visitor class graph

```
CheckingVisitor =
  <dV> DifferenceVisitor <violations> int.
DifferenceVisitor =
  <sV> SummingVisitor
  <difference> int
  <stack> Stack. // from java.util.*;
SummingVisitor = <total> int.
```


DemeterJ

CheckingVisitor

```
init {{ violations = 0; }}
after Container {{ diff = // use dV
   cap = host.get_capacity()
   if (diff > cap) violations = violations + 1; }}
return int {{ violations }}
```

need to give return type since DemeterJ does not analyze code between {{ and }}

CheckingVisitor

```
void start() {violations = 0; }
void after(Container host) { diff = // use dV
   cap = host.get_capacity()
   if (diff > cap) violations = violations + 1; }
Object getReturnValue() {return
   new Integer(violations);}
```


DifferenceVisitor

```
init {{ stack = new Stack(); }}
before Container {{ stack.push(… sV.get_total() …); }}
after Container {{ difference =
   sV.get_return_val() - … (Integer) stack.pop() …; }}
return int {{ difference }}
```


DifferenceVisitor

```
void start() { stack = new Stack(); }
void before(Container host){ stack.push(
  sV.getReturnValue()); }
void after(Container host) { diff = ((Integer)
   sV.getReturnValue()).intValue()  -
   ((Integer) stack.pop()).intValue(); }
Object getReturnValue() {return new Integer(diff);}
```


SummingVisitor

```
init {{ total = 0; }}
before Weight
   {{ total = total + host.get_i(); }}
return int {{ total }}
```

DJ

SummingVisitor

```
SummingVisitor {
{{ void start() { total = 0; }
    void before(Weight host)
      { total = total + host.get_i(); }
    Object getReturnValue()
      {return new Integer(total);}
}}  }
```


Adaptiveness

- program we developed can be used with many different class graphs which satisfy a small set of constraints:
  - Container must contain Weight
  - Weight must have data member i of type int
  - Container must have data member capacity
- 2 examples

Class graphs

```
Container = <contents> List(Item) <capacity> Capacity.
Item : Container | Simple.
List(S) ~ {S}.
Simple = <name> Ident <w> Weight.
Capacity = <i> int.
Weight = <i> int.

 alternative: move <w> Weight to Item
```


Class graphs

```
Container : Container1 | Container2 common
  <contents> List(Item) <capacity> Capacity.
Container1 = <packaging1> Simple.
Container2 = <packaging2> Simple.
Item : Container | Simple common <w> Weight.
List(S) ~ {S}.
Simple = <name> Ident.
Capacity = <i> int.
Weight = <i> int.
```