Better Software Variability Using A Functional, Traversal-Based Implicit Invocation Architecture

by Karl Lieberherr
Northeastern University, CCIS/PRL
Joint work with Bryan Chadwick, Ahmed Abdelmeged and Therapon Skotiniotis

Vision

- If you want to take advantage of multi-core computers, it is beneficial to write programs in a style that is amenable to parallel execution.
  - Today I will present such a style: AP-F and its implementation in Java and C#, called DemeterF.
  - AP-F has other benefits: easier software variability

Modularization of crosscutting concerns

```
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

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() {};
}
```

What we want: get rid of boilerplate code

```
// area function class
class area extends ID{
  double up ( circle c, int r){ return Math .PI*r*r; }
  double up ( square s, int d){ return d*d; }
  double up ( rect r, int w, int h){ return w*h; }
  double up ( pair p, double l, double r){ return l+r; }
  static double area ( shape s){
    return new Traversal ( new area ()).
      < Double > traverse (s);}
}
```

Get back what we lost in 1967 (Jim Coplien).
But in a better way. Where is the recursion?

One of infinitely many structures belonging to the "form"

shape : circle | square | rect | x | y.

circle = <radius> int.

square = <side> int.

rect = <width> int <height> int.

interface pair = .

x = <f> circle <s> rect implements pair.

y = <f> square <s> shape implements pair.

Outline

- DemeterF: a traversal abstraction tool that supports structure-shy software
  - Intro to AP-F
  - Generalized data abstraction
  - Intro to DemeterF
    - 4 important classes
    - 3 important methods
    - Examples
  - Composition in DemeterF
- Conclusions

Builds on

- **Adaptive Programming (AP):** Graph-based traversal control based on expansions, separation of concerns into ClassGraph, WhereToGo, WhatToDo. Mostly the separation of functionality (WhatToDo) and traversal (WhereToGo). Functions don't need to traverse, and traversals don't mention any specific function.
- **Functional Programming (FP):** Side-effect-free programming, WhatToDo without local state.
- **Generic functions (GF):** dynamic dispatch for down and up methods. Predicate Dispatch. Socrates. PolyD.
- **Scrap Your Boilerplate (SYB):** separation into type preserving and type unifying computations.
- **Datatype-Generic Programming (DGP):** Polytypic Programming.

What I want

- Make you think differently about how you write your methods or functions.
- Applies to both object-oriented and functional programming.

Common task: method for an object (type-unifying implementation)

```
class U  {V v; W w;
  int t( E e ){ return up(v.t( e ), w.t( e )); }
  int up(int t1, int t2) {return t1 + t2;} }
class V { X x; Y y; Z z;
  int t( E e ){ return up(x.t( e ), e ); }
  int up(int t1, E e) {return t1 + f(e);} }
```

new U(…).t(e);

```
class U  {V v; W w;
   int t( E e ){ return up(v.t( e ), w.t( e )); }
   int up(int t1, int t2) {return t1 + t2;} }
class V { X x; Y y; Z z;
   int t( E e ){ return up(x.t( e ), e ); }
   int up(int t1, E e) {return t1 + f(e);} }
         new U(…).t(e);
```

Structure
WhereToGo
WhatToDo (DOWN/UP)

```
new Traversal(ID(DOWN/UP),
          new (WhereToGo ({U: v,w; V: x})).traverse(new U(…), e);
class U  {V v; W w; }    class V { X x; Y y; Z z; }
UP = {
int up(U u, int t1, int t2) {return t1 + t2;}
int up(V v, int t1, E e) {return t1 + f(e);} }
```

DOWN = {}

Other code we write often (type-preserving implementation)

```
class U  { V v; W w;
  U t(E e){ return up(U(v.t(e), w.t(e)));}
  U up(U u) {return u;}
class V { X x; Y y; Z z;
  V t(E e){ return up(V(x.t(e),y.t(e),z.t(e)));}
  V up(V v) {return v;} }
```

Structure
WhereToGo
WhatToDo (DOWN/UP)

new U(…).t(e);

What are the concerns?

```
class U  { V v; W w;
  U t(E e){ return up(U(v.t(e), w.t(e)));}
  U up(U u) {return u;}
class V { X x; Y y; Z z;
  V t(E e){ return up(V(x.t(e),y.t(e),z.t(e)));}
  V up(V v) {return v;} }
```

Structure
WhereToGo
WhatToDo (DOWN/UP)

new U(…).t(e);

new Traversal(Bc(DOWN/UP),
new (WhereToGo (everywhere)).traverse(new U(…), e);

class U {V v; W w; }   class V { X x; Y y; Z z; }

DOWN = {}   UP = {}

A Safer Form of Aspects: Adaptive Programming (AP)

• General Aspects (96)	• AP (92)
• Base: any program	• Base: any traversal
• join points	• before / after events
• pointcuts	• method signatures
• advice	• method bodies

AP is a special case of AOP.
AP-F is a special case of AP.
DemeterF is an implementation of AP-F (Java, C#)

Theme: noise elimination

- leads to better separation of concerns and better correlations.
  - requirements: problem reductions eliminate noise
    - irrelevant information is eliminated.
  - programming: traversal abstractions eliminate noise
    - irrelevant classes are not mentioned. Correlations: WhereToGo, WhatToDo.

WhatToDo: Type-Unifying

Return 1 for each Person-object.

Return 0 for other leafs.

Sum for lists: int up(ConsI c, int f, int r) {return f+r;}

Pass for one part: int up(Object o, int p) {return p;}

Processing works for many class graphs.
Static type-checking protects against some bad changes to the class graph.

AP-F/DemeterF: a useful tool to get rid of boiler plate code

- OOP and Functional Adaptive Programming
- Traversal abstraction library
  - similar to a library supporting the visitor design pattern
  - new: better traversal abstraction through multi-methods and traversal control

Recent related activity

- **Functional Visitor Pattern (VP):** OOPSLA 2008 (Oxford paper). Bruno Oliveira, Jeremy Gibbons.

Important Software Topic To Which We Contribute with AP-F

- Generalizing the Data Abstraction Problem
  - Through Better Abstractions for Traversals

Generalizing Data Abstraction

well-accepted, since 70s

- Data Abstraction Principle: the implementation of objects can be changed without affecting clients provided the interface holds.
- Adaptive Programming (AP) Principle: the interface of objects can be changed without affecting clients provided the generalized interface holds.

since 90s

Motto of AP-F

- Law of Demeter: Talk only to your friends.
- Law of AP-F: Listen only to your friends and selectively receive information from your superiors.
- You might not need all the information you get from your friends. You only take what you need and what the communication protocol requires.

What happens if you don't use traversal abstraction?

- You write a lot of boiler plate code! Related work: Scrap Your Boilerplate (SYB) started by Ralf Laemmel and Simon Peyton-Jones.
- You tightly couple structure (changes frequently) and computation (more stable). That is against common sense.
- You always deal with the worst-case scenario of AP-F.

A Quick Intro to AP-F

- An AP-F program is defined by two sets of functions which adapt behavior of a predefined traversal with a multiple dispatch function. The two sets of functions:
  - down
  - up

Modularization of crosscutting concerns

In AP-F:

concern 1: class graph

concern 2: WhereToGo

concern 3: WhatToDo

Instead of writing this

```
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();
  }
}
```

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() {};
}
```

Motto of AP-F

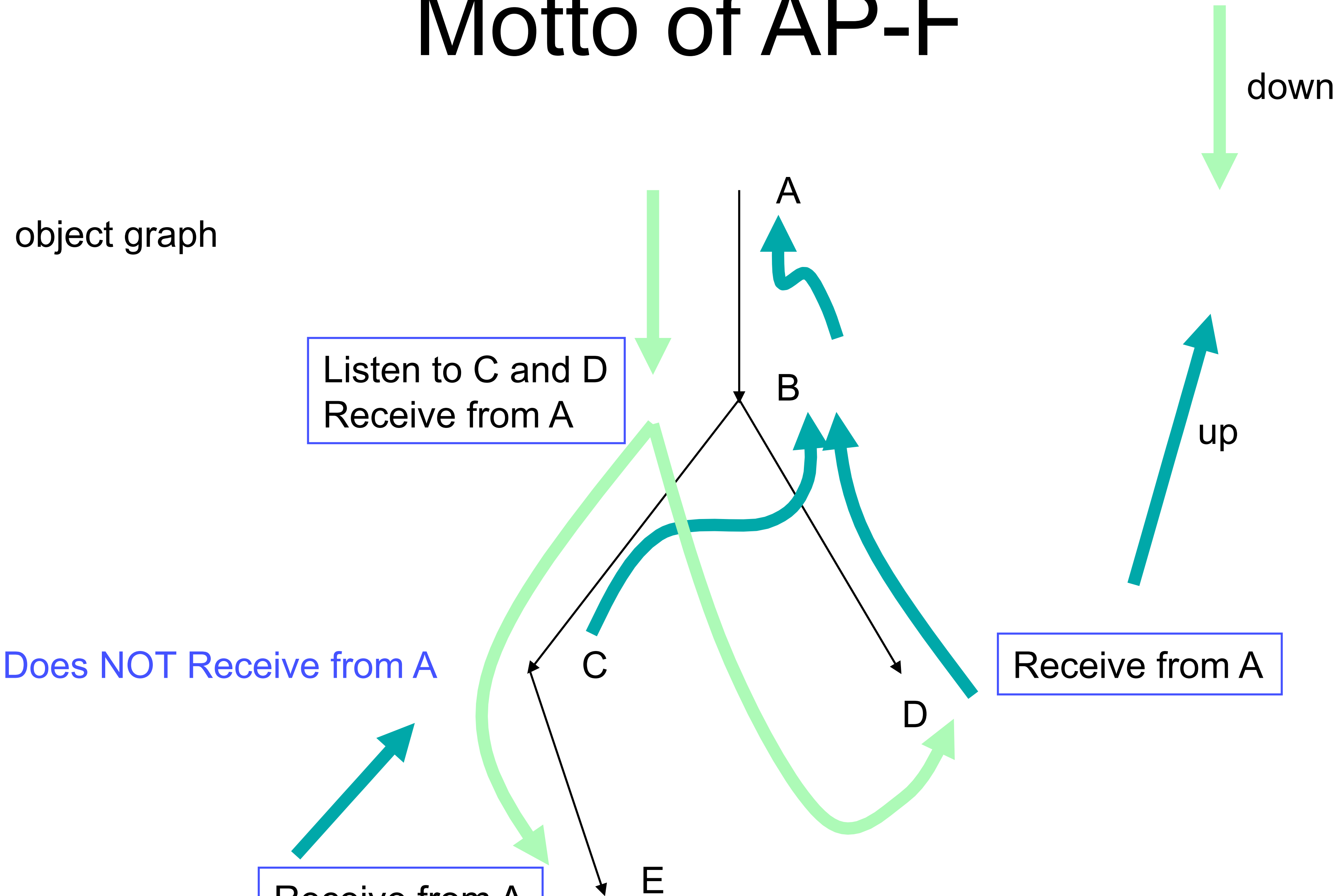
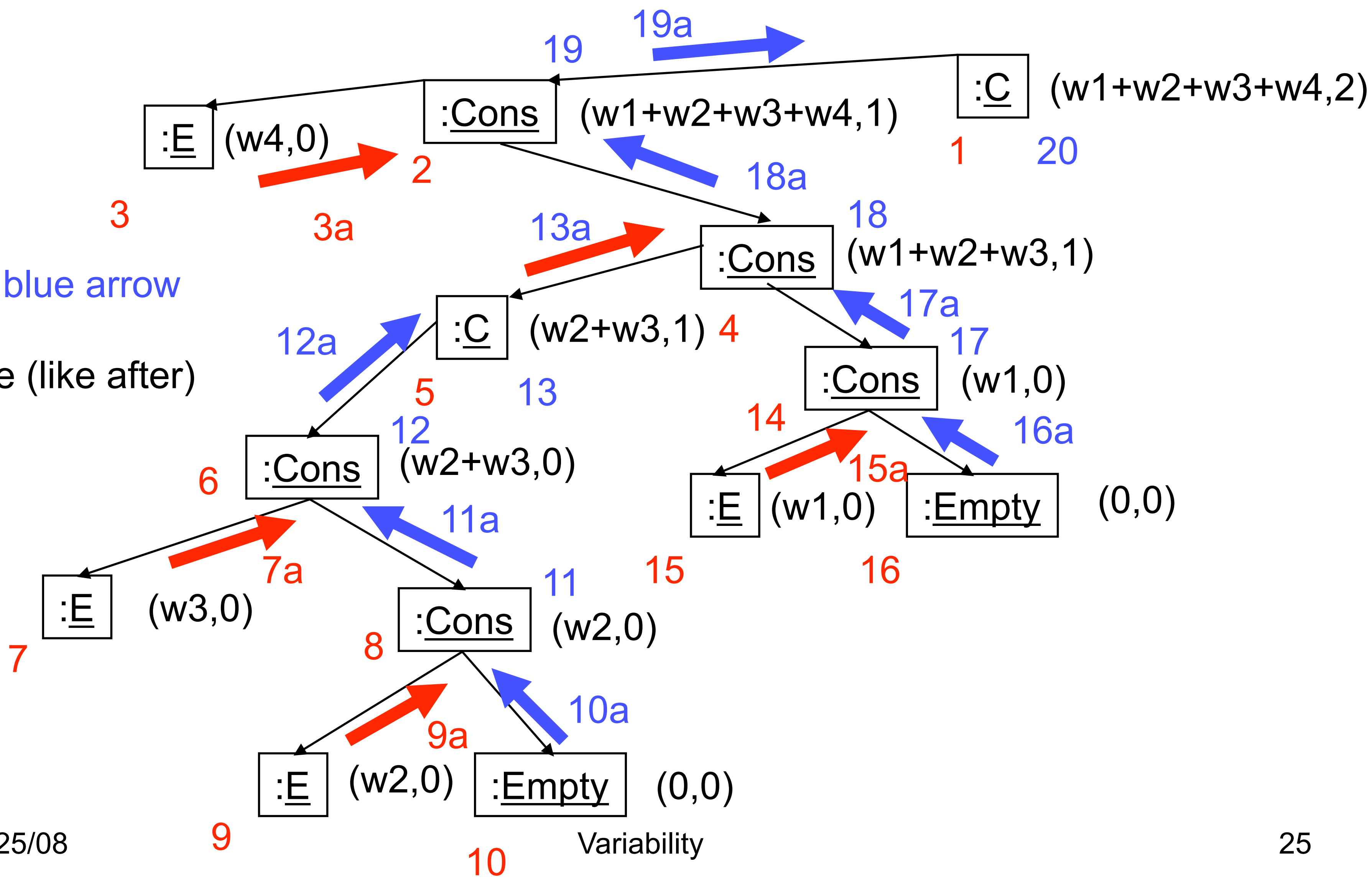

down
A
object graph
Listen to C and D
Receive from A
B
up
Does NOT Receive from A
C
Receive from A
D
Receive from A
E

Illustration of up



after blue arrow
up is
active (like after)

DemeterF

- A significant class of applications can be coded modularly without tangled code and without having to write boiler plate code.
- DemeterF has an optional code generation tool to generate Java classes from grammars (data binding) and traversal strategies (navigation binding).

Family of Programs

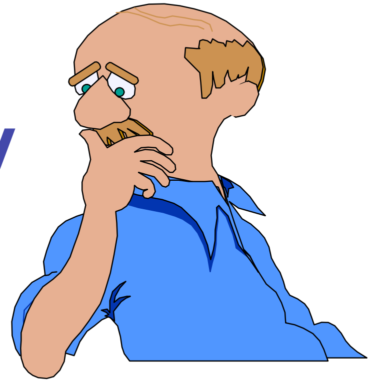
- Writing programs for a family of grammars.
- Grammar also defines data types.

Example of a Grammar

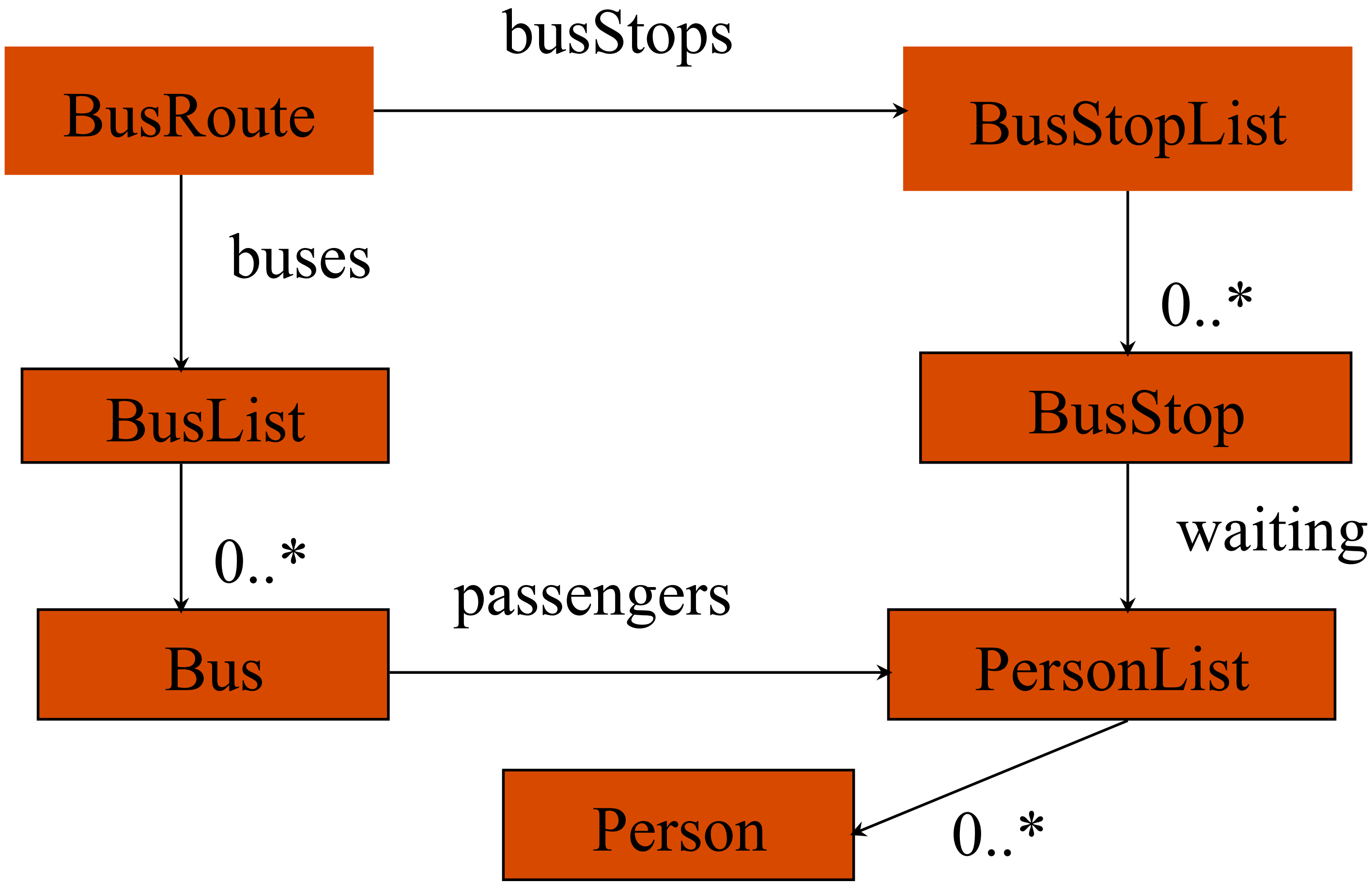
```
// Parameters of Maximization Problem
Type = "Typ"  /*...*/.
Outcome = "O" /*...*/.
Quality = "Q" /*...*/.
RawMaterial = "R" <type> Type /*...*/ .

Derivative = "D" <type> Type <price> double.
FinishedProd = "F" <r> RawMaterial <o> Outcome.


// Helper classes
Pair(A,B) = <a> A <b> B.
PlayerID = <id> int.


// Main Game structures...
SDG = "SDG"
     <players> List(Player)
     <account> List(Pair(PlayerID,Double))
     <store> List(Pair(PlayerID,PlayerStore))
     <config> Config.

Player = "player" <id> PlayerID <name> String.
PlayerStore = "pstore" <forSale> List(Derivative)
                  <bought> List(BoughtDeriv).
BoughtDeriv = "bought" <d> Derivative
                  <seller> PlayerID
                  <r> Option(RawMaterial)
                  <f> Option(FinishedProd).
```

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

WhereToGo: Strategy

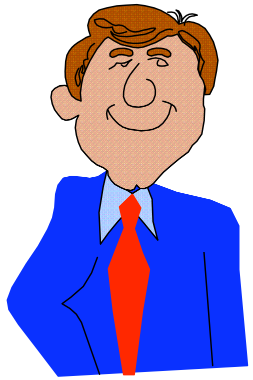


first try: `from` BusRoute `to` Person

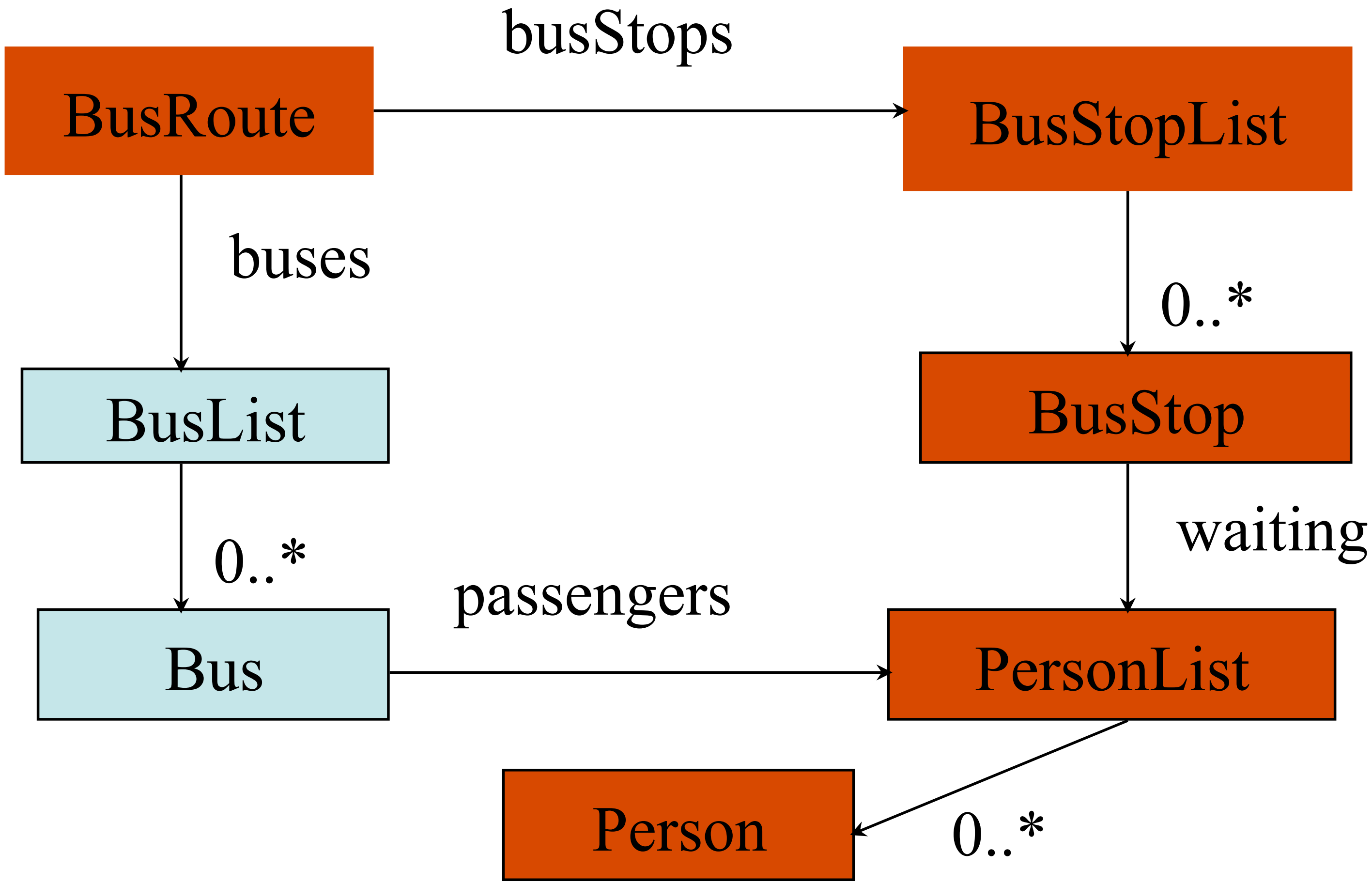


find all persons waiting at any bus stop on a bus route

WhereToGo: Strategy



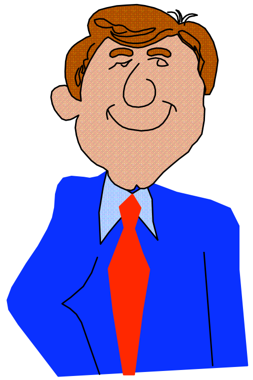
from BusRoute via BusStop to Person

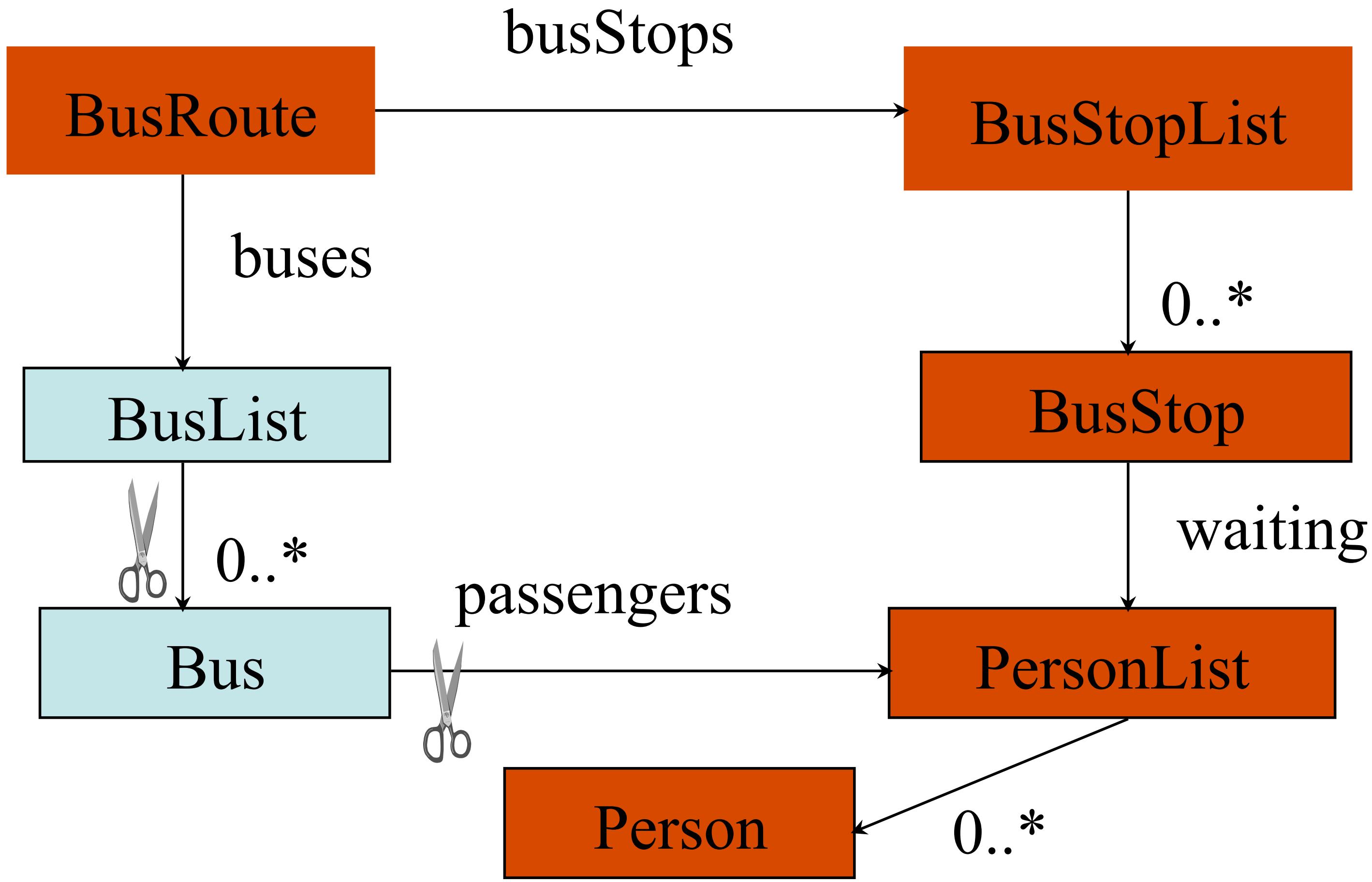


find all persons waiting at any bus stop on a bus route

WhereToGo: Strategy

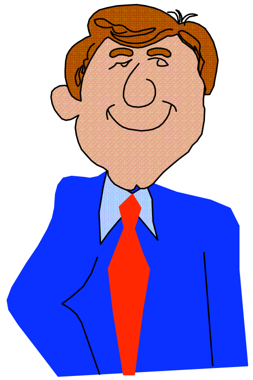
Altern.: `from` BusRoute `bypassing` Bus `to` Person



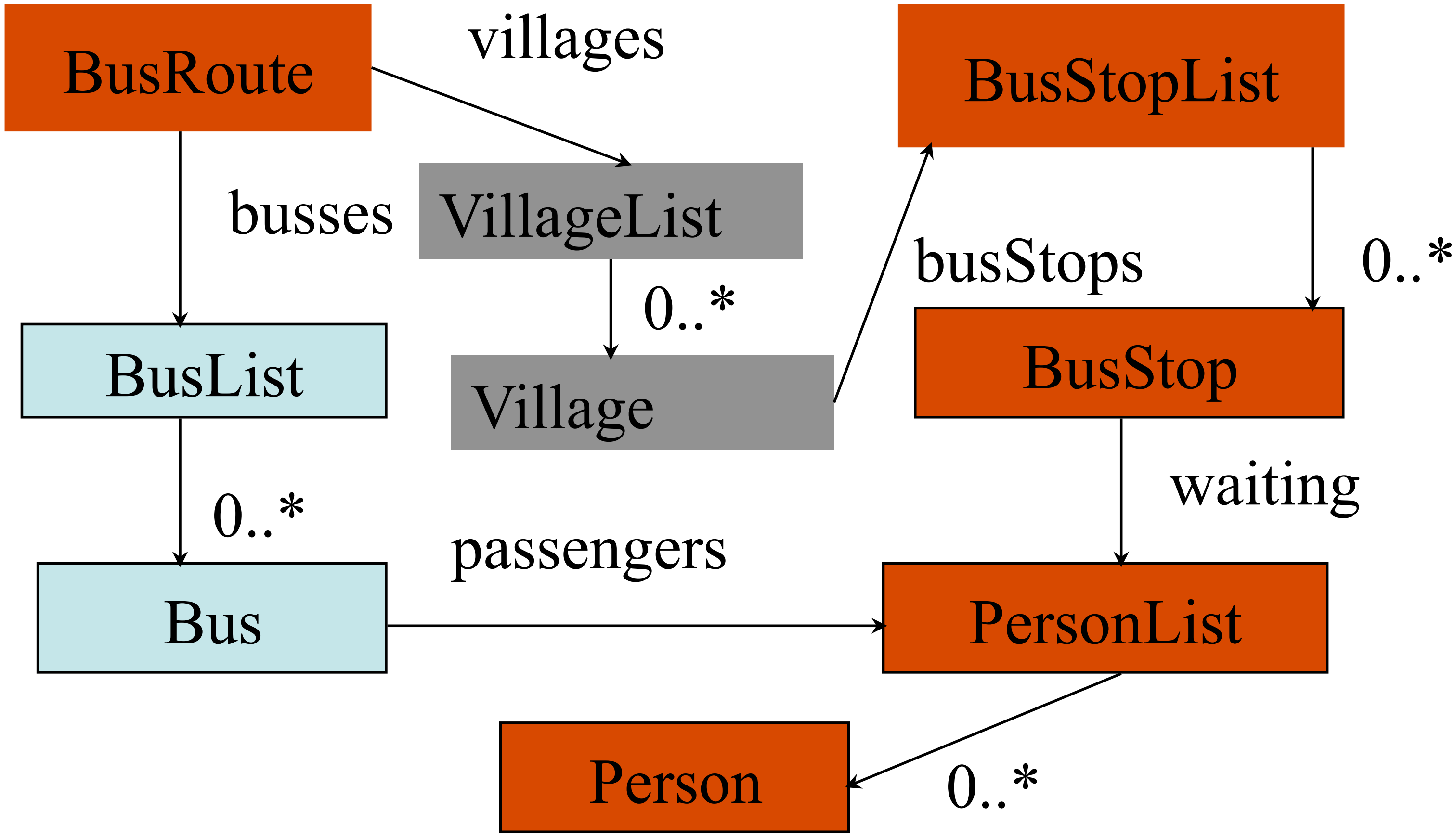


find all persons waiting at any bus stop on a bus route

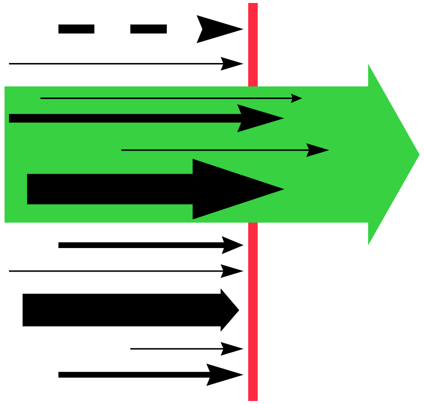
Robustness of Strategy



`from` BusRoute `via` BusStop `to` Person



Filter out noise in class diagram



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

`from` BusRoute `via` BusStop `to` Person

replaces traversal methods for the classes

BusRoute VillageList Village BusStopList BusStop PersonList Person

WhatToDo: Type-Unifying

Return 1 for each Person-object.

Return 0 for other leafs.

Sum for lists: int up(ConsI c, int f, int r) {return f+r;}

Pass for one part: int up(Object o, int p) {return p;}

Processing works for many class graphs.
Static type-checking protects against some bad changes to the class graph.

Architectural View of DemeterF: Traversal-based Implicit Invocation (II)

- POS
  - Data format abstracted away
  - Variability: Computation separate from data representation
    - Either can be changed independently of other
- NEG
  - Efficiency, gained back through parallel execution on multi-core.

DemeterF Function Objects

- up methods
  - several parameters (from your friends)
  - interesting default: reconstruct using (new) parameters
- down methods
  - three parameters (where, which field/path, what came from above)
  - default: identity (return third parameter)

What is new with DemeterF

- easier and safer generalization of functionality thanks to multimethods: exploit similarities between up methods at different nodes. Type-checking captures missing methods.
- easy parallelization

What Hinders Creation of Flexible Software

- Manually following rules like: Follow the structure, follow the grammar.
- Actively call traversal methods (explicit traversal problem).
- Also leads to manual passing of arguments (explicit argument problem).

explicit traversal problem

```
class U  {V v; W w;
   int t( E e ){ return up(v.t( e ), w.t( e )); }
   int up(int t1, int t2) {return t1 + t2;} }
class V { X x; Y y; Z z;
   int t( E e ){ return up(x.t( e ), e ); }
   int up(int t1, E e) {return t1 + f(e);} }
         new U(…).t(e);
```

Structure
WhereToGo
WhatToDo (DOWN/UP)

explicit argument problem

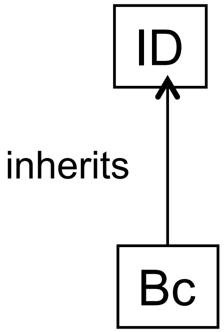
```
new Traversal(ID(DOWN/UP),
              new (WhereToGo ({U: v,w; V: x})).traverse(new U(…), e);
class U  {V v; W w; }    class V { X x; Y y; Z z; }
UP = {
int up(U u, int t1, int t2) {return t1 + t2;}
int up(V v, int t1, E e) {return t1 + f(e);} }
```

DOWN = {}

Still follow the grammar?

- Might have to write customized methods for every node. Extreme case.
- Encode local information about structure in customized methods.

The 4 most important classes in DemeterF



ID == id-for-trav-arguments and
id-for-prims (int,String, etc.)

Bc == ID + rebuild

Use ID for type-unifying processing.
Use Bc for type-preserving processing.

Traversal(ID, EdgeControl)

EdgeControl

The 3 most important methods in DemeterF

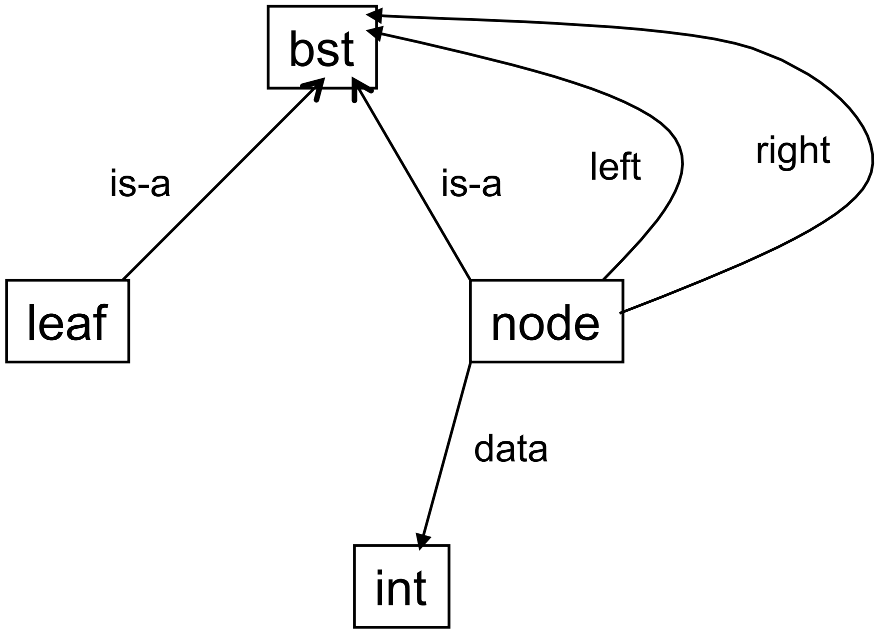
- for initiating a traversal: traverse
- for modifying a traversal (WhatToDo):
  - down:
    - method is activated when the down method of the parent is complete.
  - up:
    - method is activated when all subtraversals have returned.

Type-unifying: ID, Traversal up, traverse

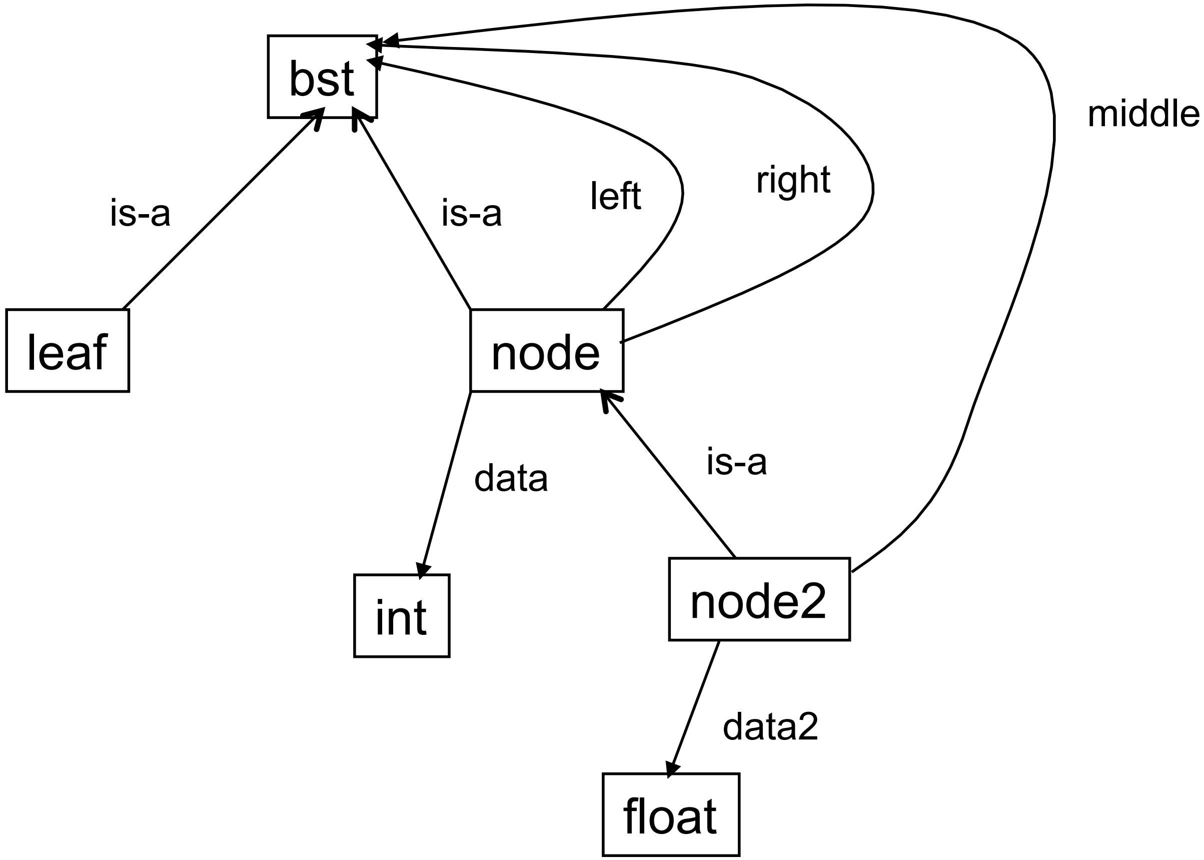
```
// Produces a string representation of the tree
  static class Str extends ID {
     String up(leaf l) { return ""; }
     String up(node n, int d, String l, String r) {
        return "("+d+" "+l+" "+r+")";
     }
  public static String doStr(bst t){
        return new Traversal(new Str()).<String>traverse(t);
     }
  }
```

how generic? positions 2 and 3 are important

Example: Class Graph bst



Example: Class Graph bst



Type-preserving: Bc, Traversal up, traverse

```
// Increments each int element and rebuilds the
// resulting tree
  static class Incr extends Bc {
   int up(int i) { return i+1; }
  public static String doIncr(bst t){
        return new Traversal(new Incr()).
          <bst>traverse(t);
     }

}
```

how generic? very general

ID, Traversal, EdgeControl up, traverse

```
// Find the minimum data element in the BST... keep going left
  static class Min extends ID {
      int up(node n, int d, leaf l) { return d; }
      int up(node n, int d, int l) { return l;  }

      public static int min(bst t){
          EdgeControl ec = EdgeControl.create(
            new Edge(node.class,"right"));
          ec.addBuiltIn(leaf.class);
          return new Traversal(new Min(), ec).<Integer>traverse(t);
      }
  }
```

ID, Traversal
down, up, traverse

```
// Produces a list of strings representing the paths to each
//   leaf element in the tree

static class Paths extends ID {
      String down(node n, node.left f, String path)
        { return path+".left"; }
      String down(node n, node.right f, String path)
        { return path+".right"; }
      List<String> up(leaf l)
        { return new Empty<String>(); }
      List<String> up(node n, int d, List<String> l, List<String> r, String p)
        { return l.append(r).push(" "+d+" : "+p); }
      public static String doPaths(bst t){
          return new Traversal(new Paths()).
             < List<String>>traverse(t, "root"); }

   }
```

Type-unifying

- Tree Height, bottom up:
- Unifying type: (int height)
- fold: max + 1

```
class Height extends ID{
 int up(node t, int d,
   int l, int r){ return
    Math.max(l,r)+1; }
 int up(leaf l){return 0;}
}
```

Dispatch function of DemeterF

- chooses most specific method
  - more information means more specific
  - only list arguments up to last one that is needed

Traversal Constructors

- **Traversal**(ID ba)

  Takes an ID function object,
  traverses every where
- **Traversal**(ID ba, EdgeControl eb)

  In addition takes a traversal control object.
- Edge control may be specified using domain-specific language.

Unifying Type

- The unifying type which may contain many more elements than only what we need to compute.
- We want a type `T = (U1, U2, ...)` so that `T` carries just enough information to implement up: `T up (X x, T t1, T t2, ...)` at all nodes in the structure.

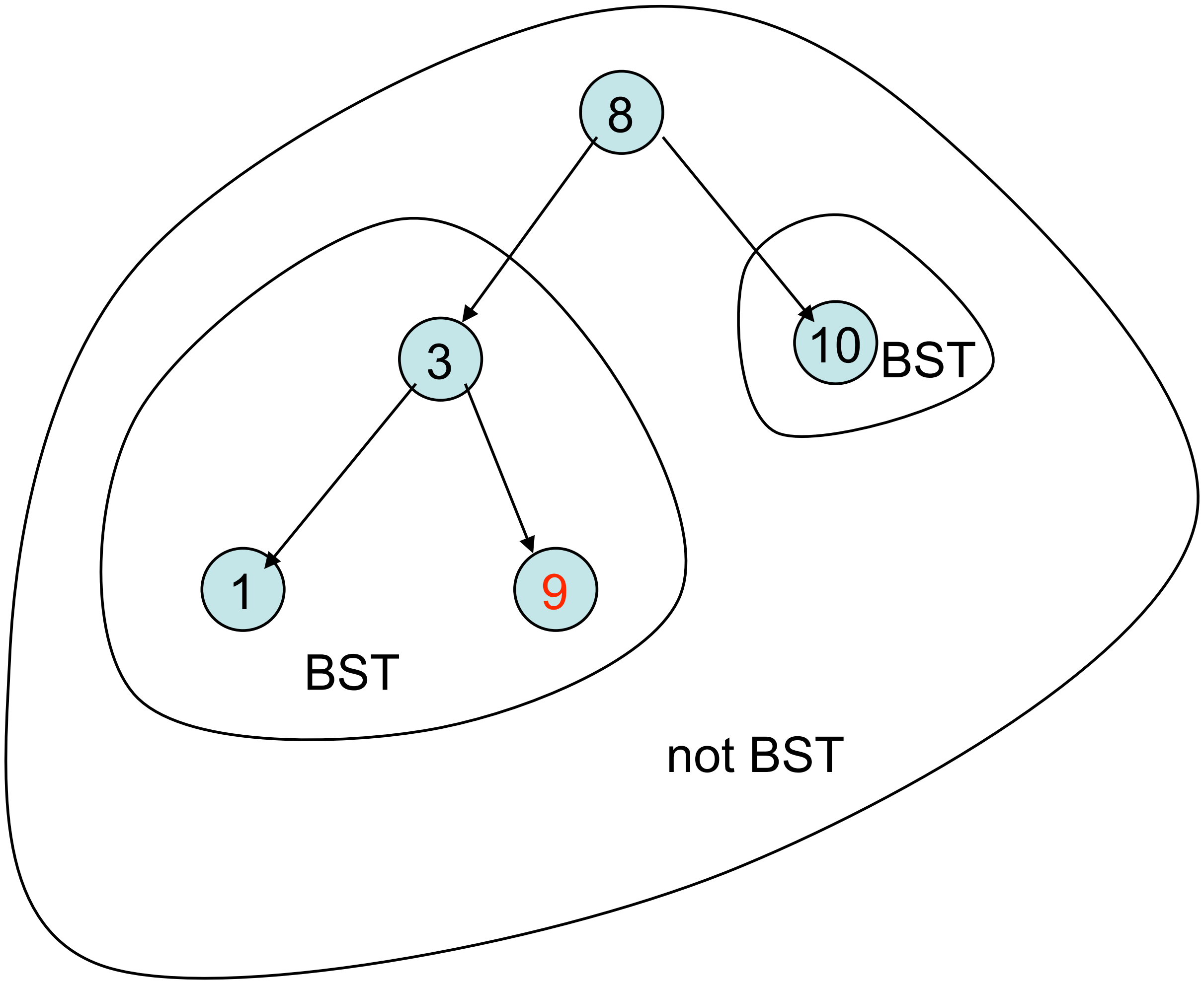
Type-unifying Examples

- Binary Search Tree Property:
- Given a binary tree of ints, are all nodes in the left tree smaller than the root and all nodes in the right subtree greater to or equal than the root.
- Consider the maximum max of the left subtree and the minimum min of the right subtree to do the max < d <= min check.
- Unifying type: (boolean BSTproperty, int min, int max)
- Base case: Empty tree: (true, +oo, -oo)
- fold: operation on two triples.

Check BST Property

- An integer tree T, with integer n at its root, is a binary search tree if (all must hold)
  - the left and right tree of T are a BST
  - the maximum of all nodes in the left tree of T is smaller than n
  - the minimum of all nodes in the right tree of T is greater than or equal to n

Why do we need the last 2?



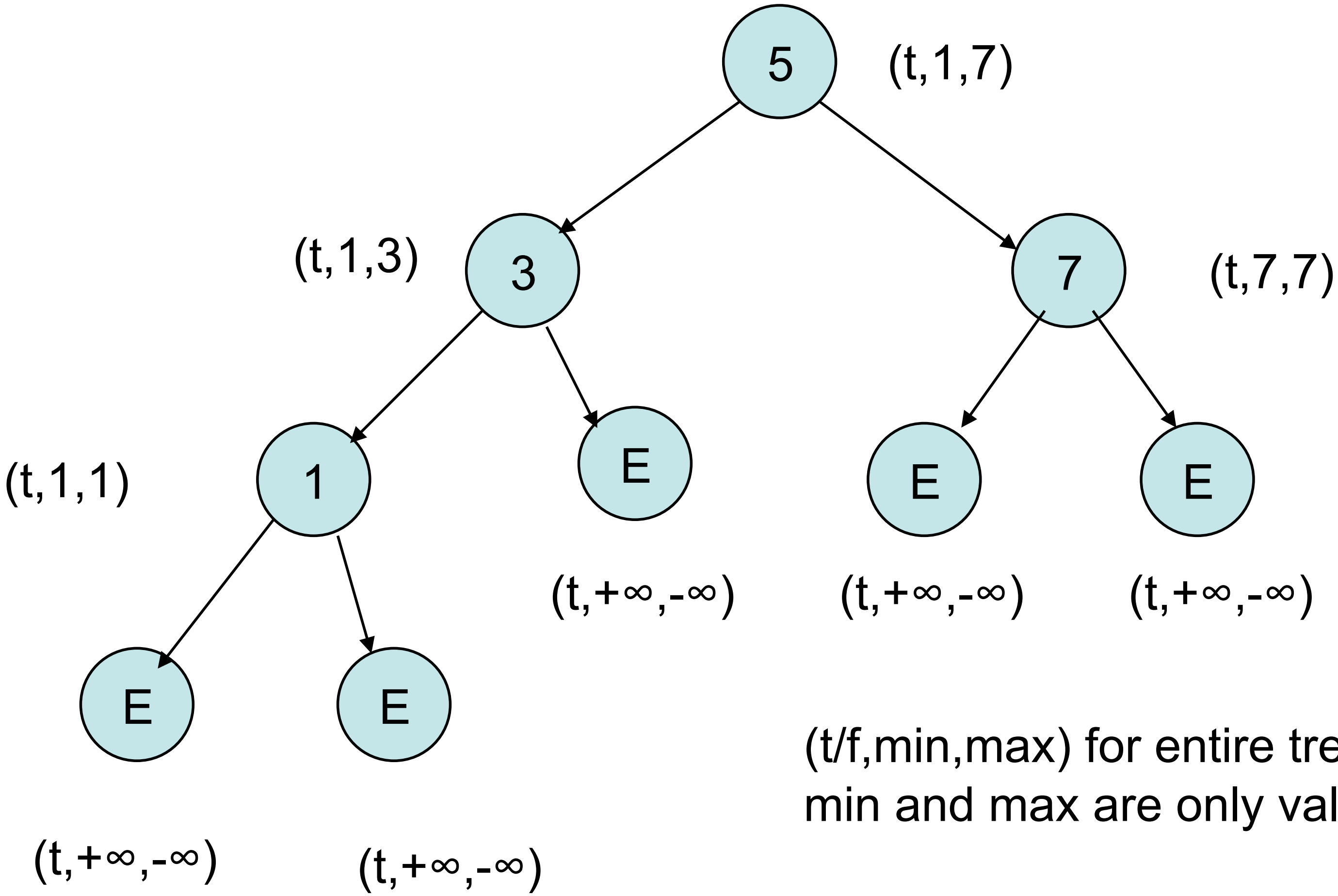
Check for BST Property

```
class Check extends ID{
  Pack up(leaf l){ return new Pack(true); }
  Pack up(node t, int d, Pack lt, Pack rt){
    return new Pack((lt.good && rt.good &&
                     (lt.max < d) && (d <= rt.min)),
                    Math.min(lt.min,d),
                    Math.max(rt.max,d)); }
  static boolean check(bst tree){
    return new Traversal(new Check())
      .<Pack>traverse(tree).good; }}
```

Pack Helper Class

```
class Pack{
   boolean good;
   int min,max;
   Pack(boolean g, int mn, int mx)
   { good = g; min = mn; max = mx; }
      Pack(boolean g){ this(g,
        Integer.MAX_VALUE,
        Integer.MIN_VALUE); }
}
```

Example



(t/f,min,max) for entire tree.
min and max are only valid if tree is BST.

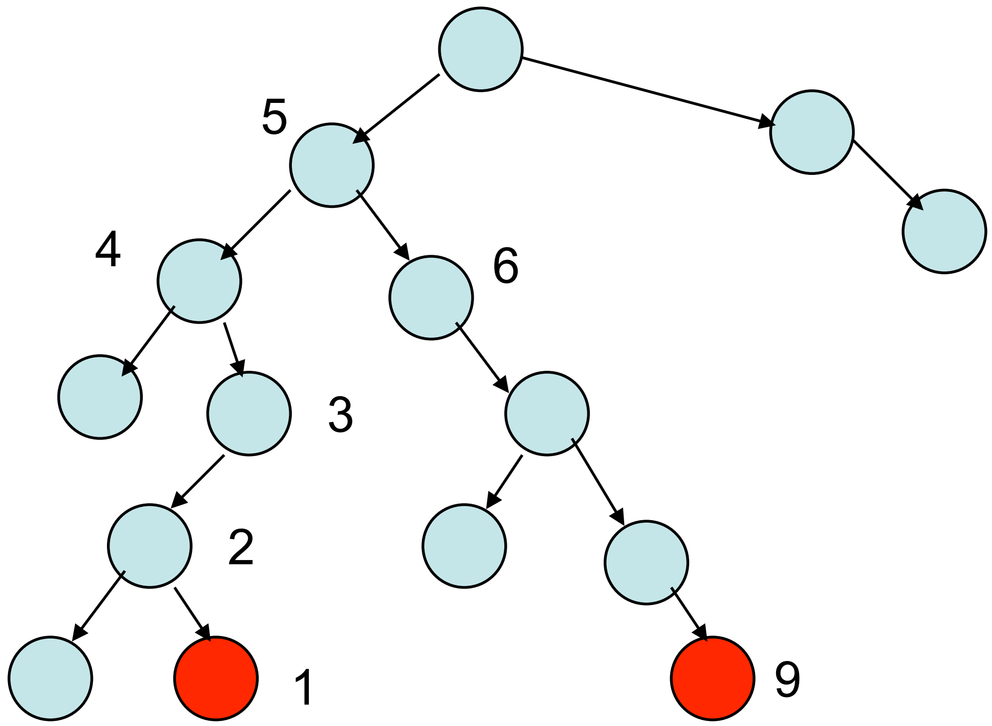
Composition (Fusion)

- How to compose traversals?

Tree Diameter

- The diameter of a tree T is the largest of the following quantities:
  - (1) diameter of T's left subtree
  - (2) diameter of T's right subtree
  - (3) the longest path between two leaves that goes through the root of T (computed from the heights of the subtrees of T)

Why do we need 1 and 2?



diameter 9, NOT through root

Diameter

```
class Diameter extends ID{ // type unifying
  DiameterPair up(leaf l) {
     return new DiameterPair(0,0);}
  DiameterPair up(node t, Integer d,
    DiameterPair l, DiameterPair r){ return
    // normal up: new DiameterPair(l.get_height() +
    // r.get_height(), l.get_diameter() + r.get_diameter());
    new DiameterPair(
      Math.max(l.get_height(), r.get_height())+1,
      Math.max(l.get_height() + r.get_height() +1,
        Math.max(l.get_diameter(), r.get_diameter())));
  }
}
```

Diameter

```
class Diameter extends ID{ // type unifying
    DiameterPair up(leaf l) {
       return new DiameterPair(0,0);}
    DiameterPair up(node t, Integer d,
      DiameterPair l, DiameterPair r){ return
      // normal up: new DiameterPair(l.get_height() +
      // r.get_height(), l.get_diameter() + r.get_diameter());
      new DiameterPair( // recursion
         Math.max(l.get_height(), r.get_height())+1, // height
         Math.max(l.get_height() + r.get_height() +1,
           Math.max(l.get_diameter(), r.get_diameter()))); // diameter
    }
}
```

HTrip

```
class HTrip{
  int ch,lh,rh;
  HTrip(){ this(0,0,0); }
  HTrip(int c, int l, int r){ ch = c; lh = l; rh = r; }
}
```

Tree Height Calculation

// Produces a HTrip (height triple), so that we can use the values of current/left/right subtrees in other calculations

```
class Height extends ID{
  HTrip up(leaf l){ return new HTrip(); }
  HTrip up(node t, int d, HTrip l, HTrip r) { return new
   HTrip(1+Math.max(l.ch, r.ch),l.ch, r.ch); }
 // Static Height calculation
  static int height(bst t){ return new Traversal(new
   Height()).<HTrip>traverse(t).ch; } }
```

Tree Diameter Calculation

```
class Diam extends ID{
  int up(leaf l){ return 0; }
  int up(node t, int d, int l, int r, HTrip h) { return
   Math.max(h.lh+h.rh+1, Math.max(l,r)); }
// Static Diameter calculation
  static int diameter(bst t){
    return new CompTraversal(
      new Functor(new Height()),
      new Functor(new Diam())).
        <Integer>traverseLast(t);  } }
```

Some References

- [AP-book] http://www.ccs.neu.edu/research/demeter/book/book-download.html
- [DemeterF home page] http://www.ccs.neu.edu/research/demeter/DemeterF/
- [DemeterF technical report] http://www.ccs.neu.edu/research/demeter/DemeterF/papers/demf.pdf
- [Functional Visitor Pattern] author = "Bruno C. d. S. Oliveira, Meng Wang, and Jeremy Gibbons", note = "Accepted at OOPSLA 2008", title = "The Visitor Pattern as a Reusable, Generic, Type-Safe Component", url = "http://www.comlab.ox.ac.uk/people/Bruno.Oliveira/Visitor.pdf", year = "2008"

Conclusions

- Better Software Variability through better abstraction of traversals. Building on:
  - AP: functions don’t traverse; traversals don’t refer to functions. Traversal(ID, EdgeControl). Both make assumptions on class graph.
  - FP, GF, SYB, DGP

Conclusions

- Delegate the writing of boiler plate traversal code to a Java Library, a C# Library, a Scheme Library, etc.
- Write the code that is interesting in `up` and `down` methods, using OO abstraction.
- You get code that is better! One important quality: possibility to execute in parallel.

Thank You!

Questions

- Can I write the traversal manually, if I want?
  - Need: ExplicitTraversal(ID).traverse(t)
  - interesting: the class graph concern is no longer separate: the explicit traversal contains information about the class graph. The class dictionary, however, is needed for data binding.

Conclusions

- Rely more on your friends. Listen to them on the agreed upon communication channels like your Facebook wall.
- Don't tightly couple structure (volatile) and computation (more stable).
- Use AP-F ideas as your design notations. To execute your designs: DemeterF home page.

Programming in DemeterF

- Every programming technology has difficulties associated with it.
- Programmers then use style rules to avoid those difficulties.
- Examples:
  - – null pointer exceptions in Java. Rule: A = [X]. -> A = XOpt. XOpt : X | EmptyX.
  - – excessive structural dependencies in OO. Rule: Law of Demeter.

DemeterF Example

- when I write:

  Integer up(Object o, Integer n1, Integer n2)
  {return n1+n2;}

  I intend that n1 and n2 are returned by a up method. They should not be accidental integers that happen to be parts of object o.

Possible style rule: Capture Avoidance Rule

- The return type of a up method should not appear in the structure being traversed (Do not return a traversed type).
- Example: If the structure contains Integer parts and the purpose is to add, we could use Float to count (see project 3).

Implementation of Capture Avoid.: Terminal Buffer Rule with Built-ins

- The terminal buffer rule (TBR) says that all terminal classes should be buffered by a class that has the terminal class as a part class. TBR improves readability of programs.
- violation: City = <name> String <zip> Integer.
- with TBR: City = CityName Zip. CityName = String. Zip = Integer.

Terminal Buffer Rule with Built-ins

- CityName and Zip are made built-in classes so that the traversal does not go further.
- Then we are free to use Integer as return type of up.

Dispatch function of DemeterF

- chooses most specific method
  - more information means more specific
  - only list arguments up to last one that is needed

Is pred lost?

– How well can we approximate maximization problem for predicate pred? Hastad etc.
– 7/8

Analysis of SDG(Max): Ideal

$t_{pred} =$ lim

n -> ∞

min

all raw materials rm of size n

satisfying predicate pred

max

all finished products fp

produced for rm

q(fp)

Analysis of SDG(Max): Real

$t_{pred} =$ lim

n -> ∞

min

all raw materials rm of size n

satisfying predicate pred

max

all finished products fp

produced for rm

q(fp)

The better the Buyer approximates the maximum, the harder the Seller has to hide it.

The better the Seller hides the maximum, the harder the Seller has to work to find it.

Seller approximates minimum efficiently

Buyer approximates maximum efficiently

```
class DerivativesFinder extends TUCombiner<List<String>>{
      Class clas;
    DerivativesFinder(Class c){ clas = c; }

    List<String> empty(){ return new Empty<String>(); }
    public List<String> apply(){ return empty(); }
    public List<String> fold(List<String> a, List<String> b) { return a.append(b); }

    List<String> apply(Derivative d) { return empty().push(d.name); }
    boolean up(TransactionType t) { return (clas.isInstance(t)); }
    List<String> up(Transaction t, boolean b, List<String> d){
        return (b?d:empty());
    }

      static List<String> findDerivatives(Round round, Class c){
              return TUCombiner.traverse(round,new DerivativesFinder(c));
      }
}
```

Grammar for SDG

History = "history" "[" <rounds>
List(Round) "]".

Round =
"round""["<ptransactions>List(PlayerTr
ansaction)"]".

PlayerTransaction = <playerName>String
<transactions>List(Transaction).

Transaction = <ttype>TransactionType
<deriv>Derivative.

TransactionType: Buy|Create|Reoffer|
Deliver|Finish.

Buy = "BUY".

Create = "CREATE".

Reoffer = "REOFFER".

Deliver = "DELIVER".

Finish = "FINISH".

Real connection: SDG/DemeterF

- artificial?
- commonalities: adaptive:
  - Buyer’s FP must adapt to Seller’s RM and achieve a goal: never lose.
  - Seller’s RM must approximate minimum making it hard or impossible to find a high quality assignment independent of Buyer’s FP.
- differences

Why is SDG a good example for DemeterF?

- relevant for ABB talk
- ABB builds robots
- analysis of trading robot problem leads to simple solution.

SDG and DemeterF for correlation workshop

- focus on MAXCSP.
- Correlations: Reductions, Symmetrization.
  - make minimization and maximization tractable.

Composition

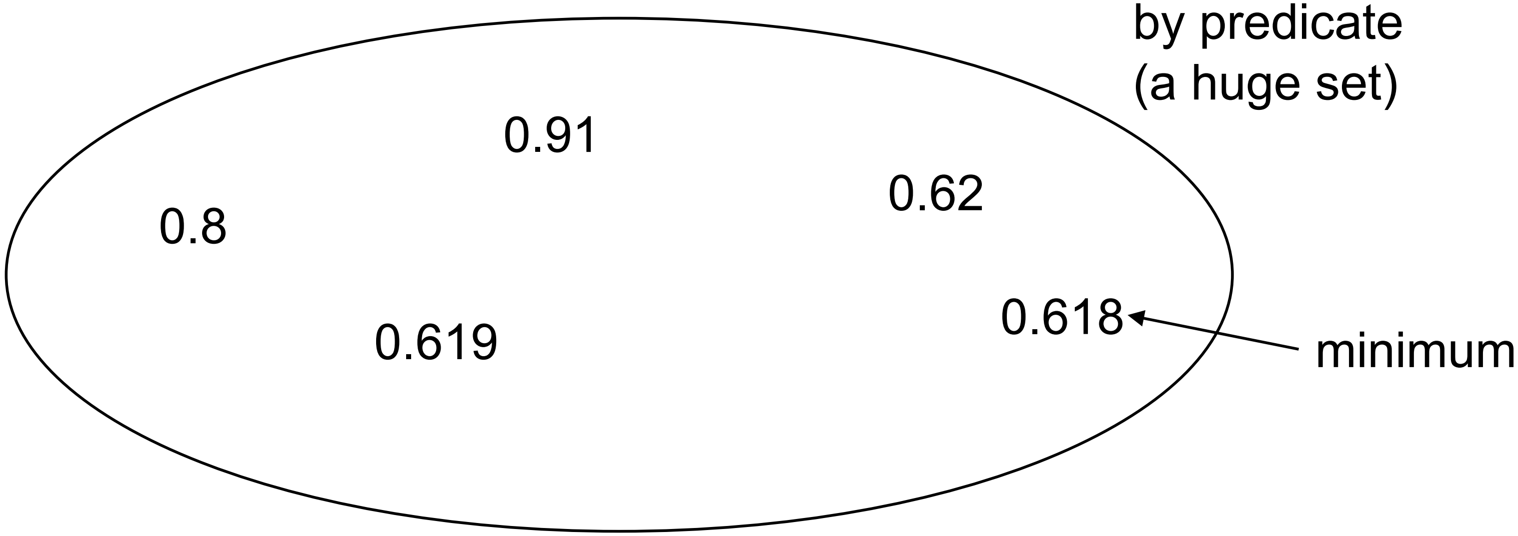
Conclusions

- Trading Robots
- DemeterF
  - Traversal Control: Traversal Graph Abstraction
  - Programming in DemeterF

To play well: solve min max

maximum
solutions

instances
selected
by predicate
(a huge set)



Analysis for one Derivative

SDG(MAXSAT)

- Choose a maximization problem: Maximum Satisfiability. Instances = weighted clauses.
- Predicates = Allowed Clause Types.
- Derivative = (Predicate, Price in [0,1], Player)
  - Example: (((2,0),(1,1)), 0.55, Ahmed)
- Players offer and buy derivatives.
- Buying a derivative gives you the rights:
  - to receive raw material R satisfying the predicate.
  - upon finishing the raw material R at quality q (trying to find the maximum solution), you receive q in [0,1].

SDG(MAXSAT)

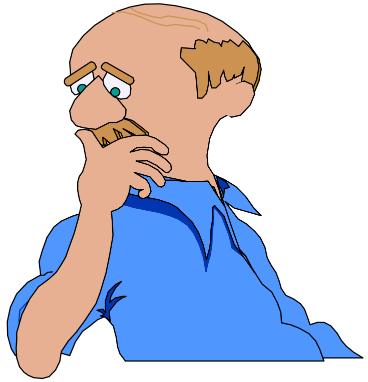
- Choose a maximization problem: Maximum Satisfiability. Instances = weighted clauses.
- Predicates = Allowed Clause Types.
- Derivative = (Predicate, Price in [0,1], Player)
  - Example: (((2,0),(1,1)), 0.55, Ahmed)
- Players offer and buy derivatives.
- Buying a derivative gives you the rights:
  - to receive raw material R satisfying the predicate.
  - upon finishing the raw material R at quality q (trying to find the maximum solution), you receive q in [0,1].
- Break-even Price: (((2,0),(1,1)), 0.618, Player) = (((2,0),(1,1)), GoldenRatio, Player)
- Phase Transition Point separating P from NP.

Unknown strategies

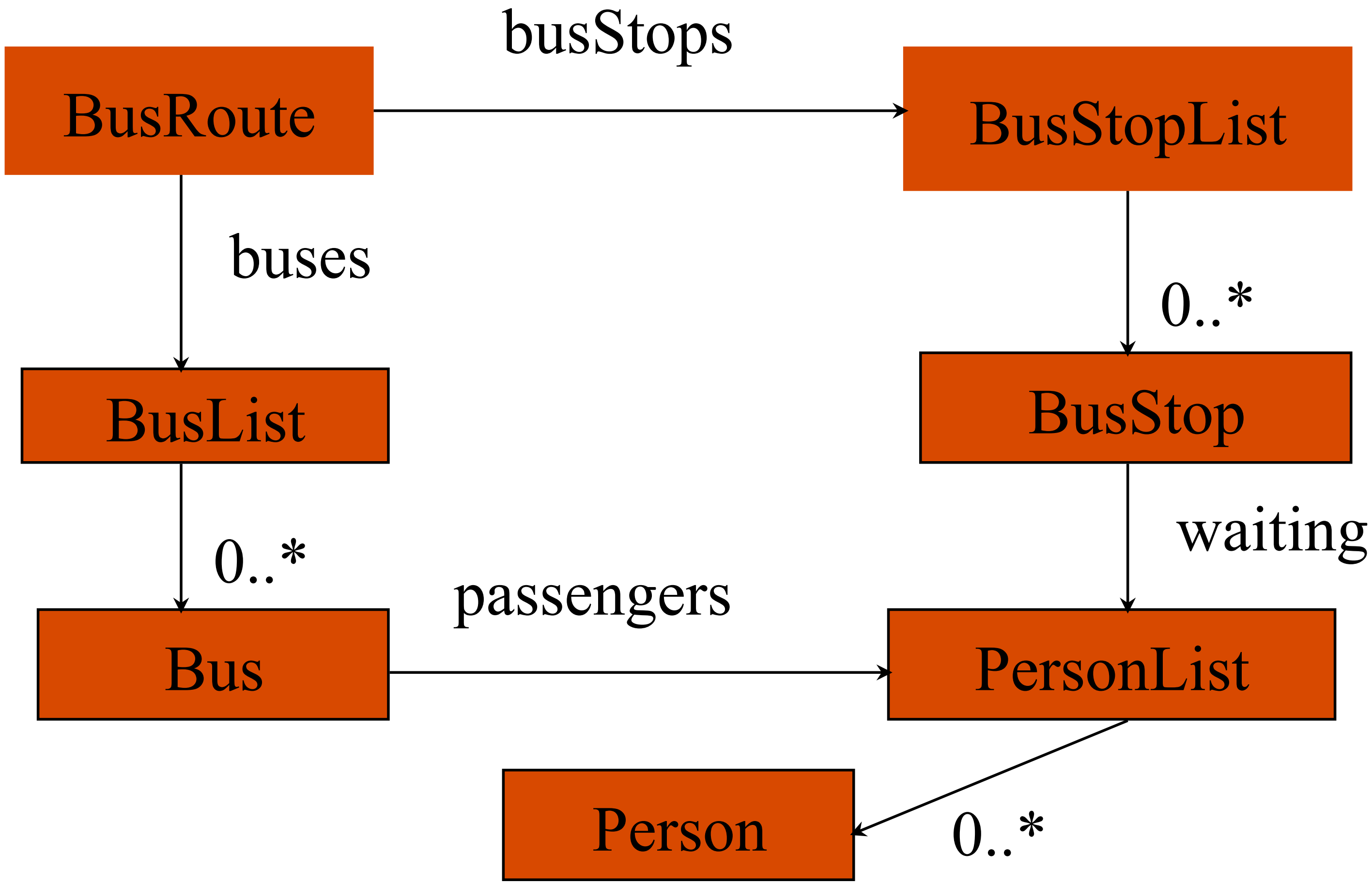
- seller (does not know the algorithmic strategy FP the buyer will use)
  - wants to hide best assignment, making it costly to find by algorithm FP of buyer, using instance satisfying predicate pred.
  - wants to find an instance where best assignment found by FP has low satisfaction ratio (for all).
- buyer (after buying)
  - wants to find the best assignment using FP

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

Traversal Strategy



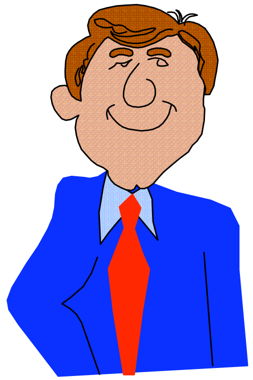
first try: `from` BusRoute `to` Person

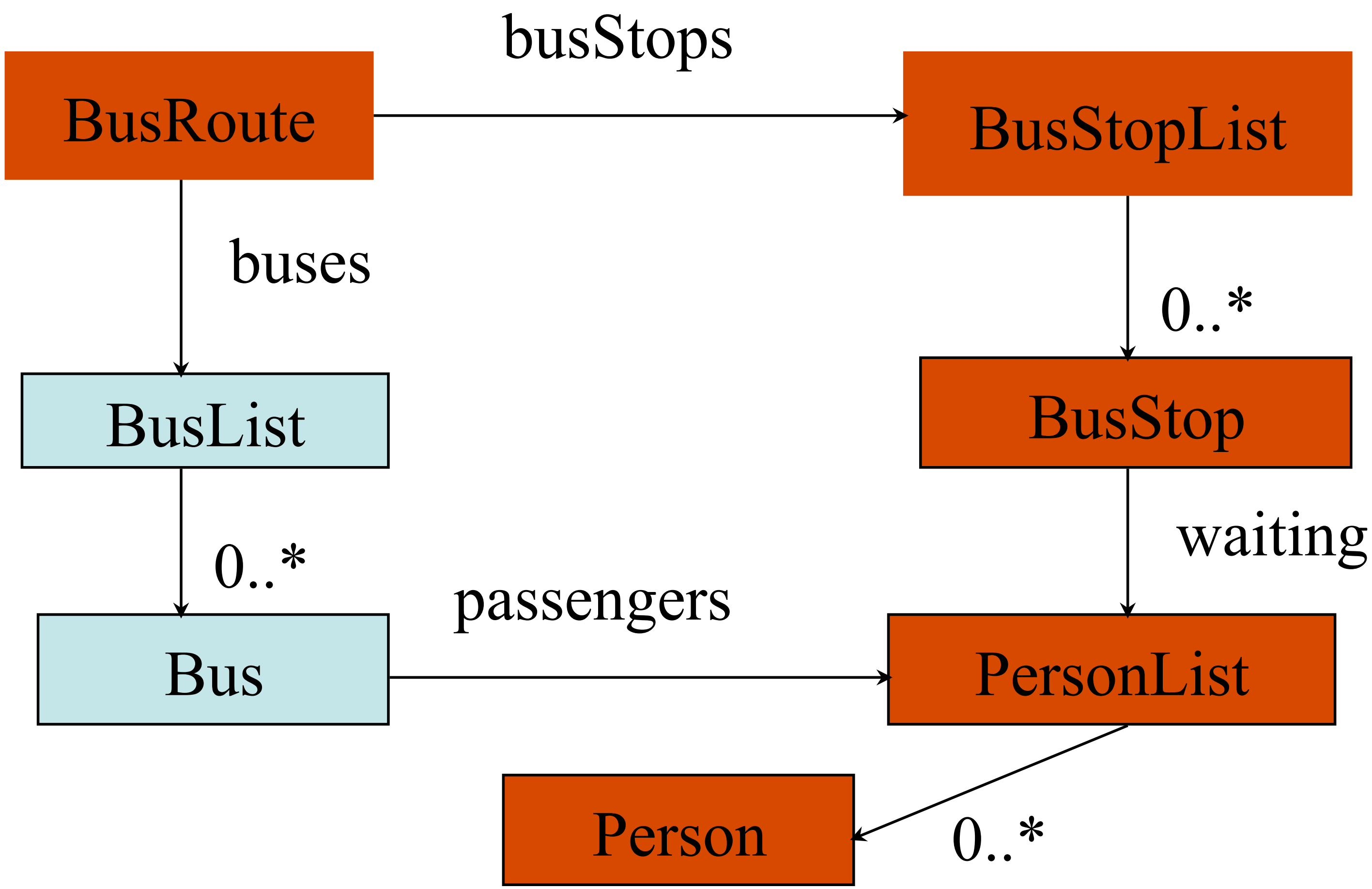


find all persons waiting at any bus stop on a bus route

Traversal Strategy

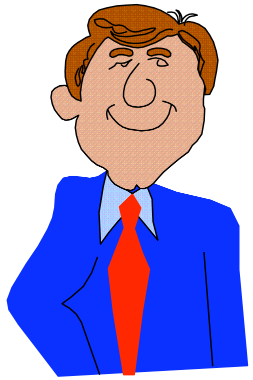
from BusRoute through BusStop to Person



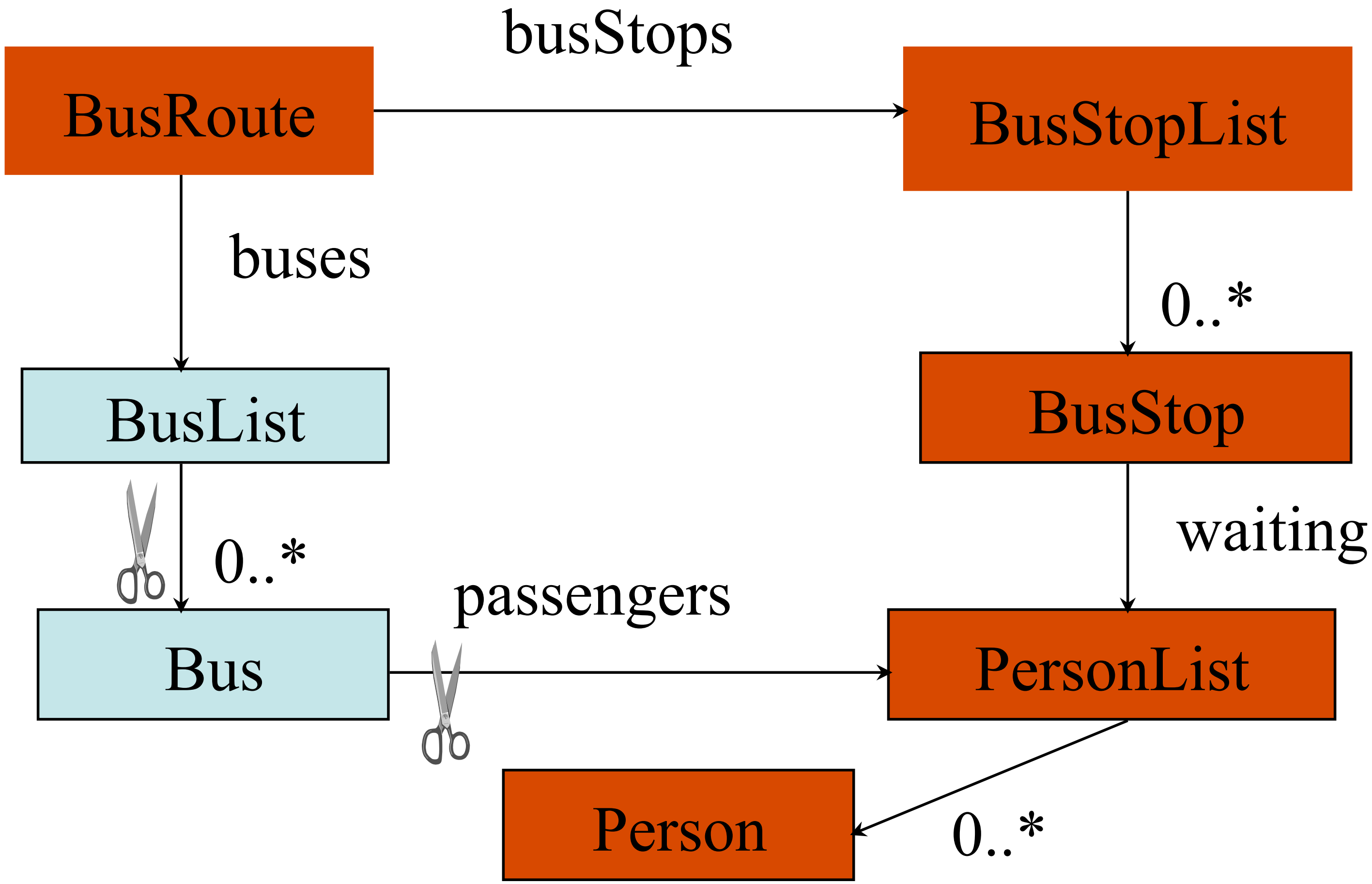


find all persons waiting at any bus stop on a bus route

Traversal Strategy

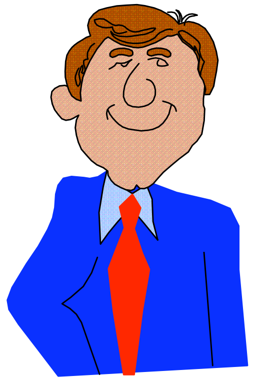


Altern.: `from` BusRoute `bypassing` Bus `to` Person

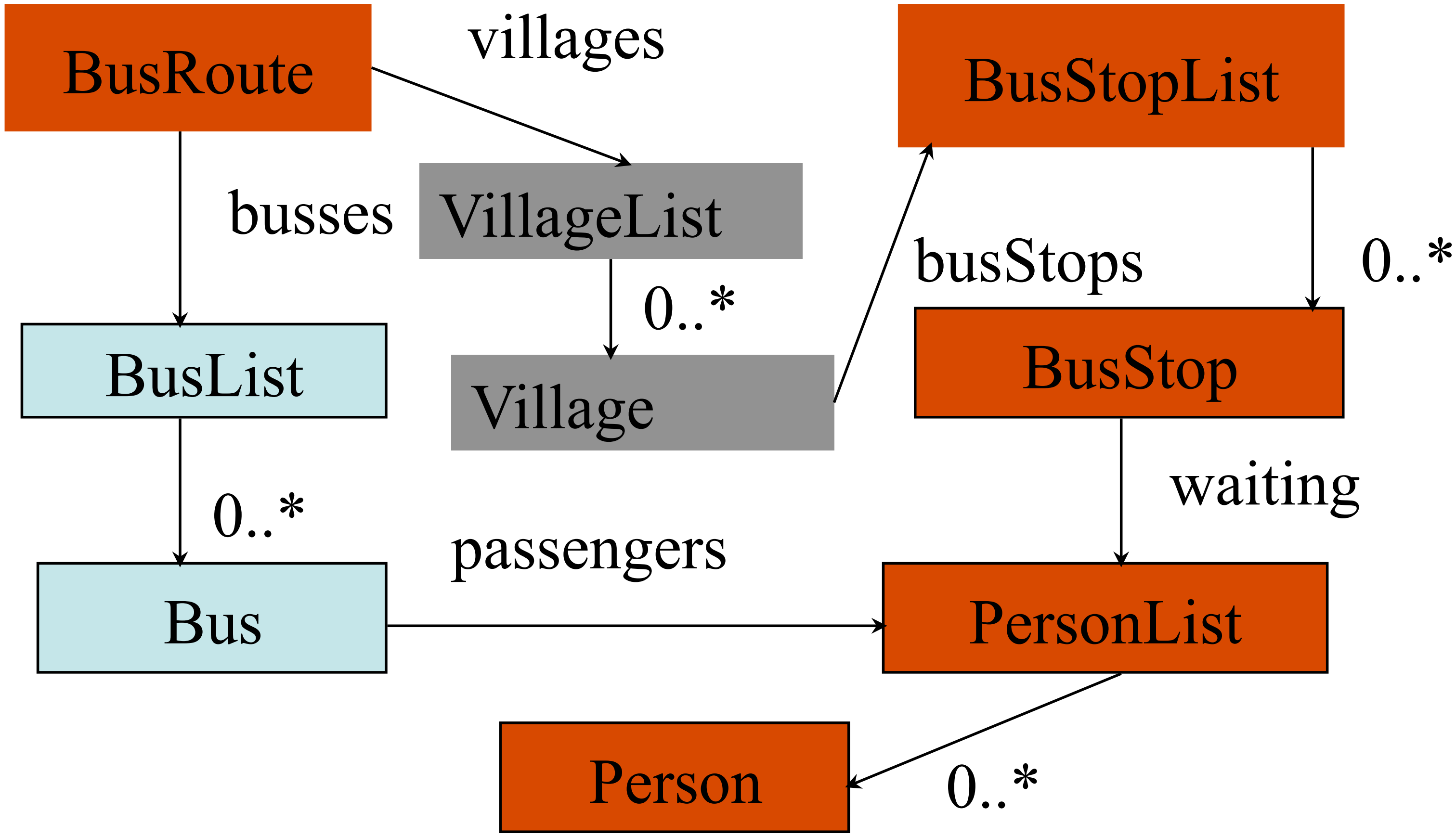


find all persons waiting at any bus stop on a bus route

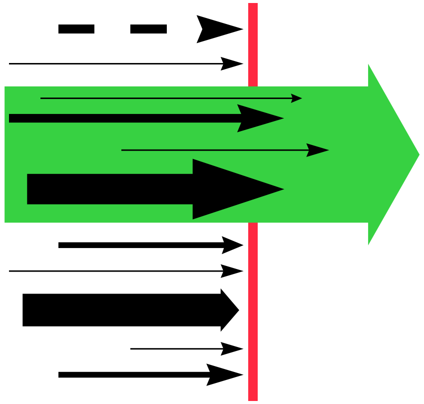
Robustness of Strategy



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



Filter out noise in class diagram



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

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

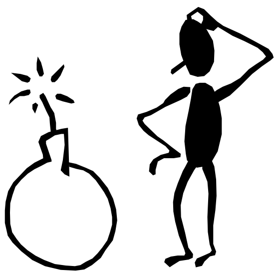
replaces traversal methods for the classes

BusRoute VillageList Village BusStopList BusStop PersonList Person

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

Connections

- Stefan Richter has seen similar ideas in a course he took at his university in New Zealand, including traversal control. Professor: Neil.Leslie@mcs.vuw.ac.nz

Builds on

- **Adaptive Programming (AP):** Graph-based traversal control based on expansions, separation of concerns into ClassGraph, WhereToGo, WhatToDo. Mostly the separation of functionality (WhatToDo) and traversal (WhereToGo). Functions don't need to traverse, and traversals don't mention any specific function.
- **Functional Programming (FP):** Side-effect-free programming, WhatToDo without local state. Programming with map and fold. Generalized fold.
- **Generic functions (GF):** dynamic dispatch for down and up methods. Predicate Dispatch. Socrates. PolyD.
- **Scrap Your Boilerplate (SYB):** separation into type preserving and type unifying computations.
- **Datatype-Generic Programming (DGP):** Polytypic Programming.
- **Attribute Grammars (AG):** AOSD 2008.
- **Visitor Pattern (Visitor)**

What are the concerns?

```
class U  { V v; W w;
   int t(E e){ return up(v.t(e), w.t(e));}
   int up(int t1, int t2) {return t1 + t2;} }
class V { X x; Y y; Z z;
   int t(E e){ return up(x.t(e), e);}
   int up(int t1) {return t1;} }
```

What are the concerns?

```
class U  { V v; W w;
  U t(E e){ return U(v.t(e), w.t(e));} }
class V { X x; Y y; Z z;
  V t(E e){ return V(x.t(e),y.t(e),z.t(e));} }
```

structure, form: structure clutters software. CRA -> RRM

What are the concerns?

```
class U  {V v; W w;
   int t(E e){ return up(v.t(e), w.t(e));}
   int up(int t1, int t2) {return t1 + t2;} }
class V { X x; Y y; Z z;
   int t(E e){ return up(x.t(e), e);}
   int up(int t1, E e) {return t1 + f(e);} }
```

new Traversal(ID(UP), new (TraversalControl (…)).traverse(e)

U: v,w

V: x

What are the concerns?

Structure
WhereToGo
WhatToDo (DOWN/UP)

```
class U  {V v; W w;
  int t( E e ){ return up(v.t( e ), w.t( e )); }
  int up(int t1, int t2) {return t1 + t2;} }
class V { X x; Y y; Z z;
  int t( E e ){ return up(x.t( e ), e ); }
  int up(int t1, E e) {return t1 + f(e);} }
```

```
new Traversal(ID(DOWN/UP), new (WhereToGo (…) ).traverse(new U(…), e);
```

U: v,w
V: x

What are the concerns?

Structure
WhereToGo
WhatToDo (DOWN/UP)

```
class U  {V v; W w;
   int t( E e ){ return up(v.t( e ), w.t( e )); }
   int up(int t1, int t2) {return t1 + t2;} }
class V { X x; Y y; Z z;
   int t( E e ){ return up(x.t( e ), e ); }
   int up(int t1, E e) {return t1 + f(e);} }
```

new Traversal(ID(DOWN/UP), new (WhereToGo (…)).traverse(new U(…), e);

U: v,w
V: x

```
class U  {V v; W w;
  int t( E e ){ return up(v.t( e ), w.t( e )); }
  int up(int t1, int t2) {return t1 + t2;} }
class V { X x; Y y; Z z;
  int t( E e ){ return up(x.t( e ), e ); }
  int up(int t1, E e) {return t1 + f(e);} }
          new U(…).t(e);
```

Structure
WhereToGo
WhatToDo (DOWN/UP)

```
new Traversal(ID(DOWN/UP),
              new (WhereToGo ({U: v,w; V: x})).traverse(new U(…), e);
```

class U {V v; W w; }   class V { X x; Y y; Z z; }

UP = {
int up(U u, int t1, int t2) {return t1 + t2;}
int up(V v, int t1, E e) {return t1 + f(e);} }

DOWN = {}

What are the concerns?

```
class U  { V v; W w;
   U t(E e){ return up(U(v.t(e), w.t(e)));}
   U up(U u) {return u;}
class V { X x; Y y; Z z;
   V t(E e){ return up(V(x.t(e),y.t(e),z.t(e)));}
   V up(V v) {return v;} }
```

Structure
WhereToGo
WhatToDo (DOWN/UP)

new U(…).t(e);

new Traversal(Bc(DOWN/UP),
new (WhereToGo (everywhere)).traverse(new U(…), e);

class U {V v; W w; }   class V { X x; Y y; Z z; }

DOWN = {}   UP = {}

Example: Class bst

```
public static abstract class bst {
    public abstract bst insert(int d);
    public static bst create(int ... ns) {
        bst t = new leaf();
        for(int i:ns)
            t = t.insert(i);
        return t;
    }
}
```

Example: Class leaf

```
// Functional Leafs
public static class leaf extends bst {
    public bst insert(int d) {
        return new node(d, this, this); }
}
```

Example: class node

```
// Functional Nodes
  public static class node extends bst {
     int data;
     bst left, right;
     public node(int d, bst l, bst r) { data = d; left = l; right = r; }
     public bst insert(int d) {
        if(d <= data)
           return new node(data, left.insert(d), right);
        return new node(data, left, right.insert(d));
     }
     // Field classes... Must be public+static.
     public static class data { }
     public static class left { }
     public static class right { }
  }
}
```