Slide 1

BETA – a Scandinavian OO Language

Erik Ernst
DEVISE, Department of Computer Science
University of Århus, Denmark
eernst@daimi.au.dk

Slide 2

Overview

- The Scandinavian approach
- Basic language concepts (and syntax)
- Specialization
- Block structure
- Virtuals
- Recursion
- Modularization

Slide 3

The Scandinavian approach

- Original challenge: simulation
- Kristen Nygaard, O.J. Dahl, Ole Lehrmann Madsen, Birger Møller-Pedersen, Bent Bruun Kristensen...
- Languages: Simula'67, BETA (75–)
- Also: participatory design, CSCW, HCI, programming environments

Slide 4

Priorities

- Conceptual modeling, not reuse
- Coherent design, not adaptation
- Construction, not restriction
- Objects came from procedure activations, not records

Slide 5

The philosophy behind

- Execution: a physical model of the real world from a certain perspective
- Modelling is cleverly supported by natural language
- We want "phenomena" (inevitably)
- ... and "concepts" (to manage complexity)
- ... organized into "processes" (for dynamics)
- A small set of orthogonal concepts and versatile abstractions preferred

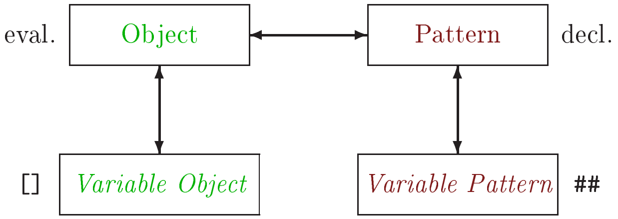
Slide 6

Basic language concepts

- phenomena correspond to *objects* (identity and state)
- concepts correspond to *patterns*
  - class-like: describing the structure of things
  - method-like: describing the structure of behavior
- concepts organized by specialization and composition
- processes (*components*) can be
  - alternating with other processes (coroutines)
  - concurrent with other processes (threads)

Slide 7

Kinds of attributes; transformations



Transformation along arrows, implicitly; *result* derived from context, e.g. "`[]`".

Slide 8

A brief syntax primer is needed

Block delimiters: `(# #)`. Declarations marked by special characters, e.g., `x: @integer`

Decl.	Entity	Std. terminology
'`: @`'	object	static reference
'`: ^`'	*var.object*	*dynamic reference*
'`:`'	pattern	pattern
'`: ##`'	*var.pattern*	*pattern variable*
'`:<`'	virtual pattern	
'`::<`'	further-binding	
'`::`'	final-binding	

Slide 9

A brief syntax primer: “..->..”

- *Evaluation* syntax, based on “->”, unifies assignment, argument transfer, and function returns
- Transfer of values, always in the reading direction
- E.g.

```
5->x;
(x,y)->max->z;
(p1,(3,4))->myRectangle->(p2,p3)
```

Slide 10

A brief syntax primer: enter/exit

- Input/output properties are defined inductively
- Atoms: `x[]`, `x##`, `integer`, `boolean`...
- Composition rule: `enter`, `exit`

```
(# Point: (# x,y: @integer enter (x,y) exit (x,y) #);
   Rect: (# ul,lr: @Point enter (ul,lr) exit (ul,lr) #);
   Read_p2: (# exit p2 #);
   p1,p2: @Point;
   r1,r2: @Rect
do
   ((p1.y,p1.x),Read_p2)->r1->r2->(p2->p1.move,p1)
#)
```


Slide 11

Kinds of attributes: transparent usage

```
(#
   putline: (# .. #);
   t: @text;
   vt: ^text;
   vp: ##putline;
do
   'Hello, world!' -> t;
   t[] -> vt[];
   putline## -> pv##;
   vt -> pv;
#)
```

The declared properties and the syntactic context together determines the implicit transformations (such as “create an instance of”, or “follow that pointer”)

Slide 12

Specialization

- Behavior can be specialized
- Attributes are inherited
- New attributes can be added
- I/O properties can be expanded
- Virtuals can be further-bound

Slide 13

Specializing behavior and I/O

```
dbOp: (# db: ^database
      enter db[]
      do db.open; INNER; db.close
      #);
dbPut: dbOp(# r: ^row enter r[] do r[]->db.insert #);
dbScan: dbOp
  (# r: ^row
  do L: (if db.more do
            db.getNext->r[]; INNER; restart L
         if)
  #);
dbPrint: dbScan(# s:^stream enter s[] do s[]->r.print #)
```

“INNER” means execute the next more special do-part; behavior specialization is similar to *syntax tree completion*

Slide 14

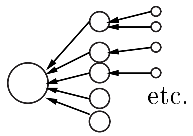
Specializing structure

```
 Point:
   (# x,y: @integer;
      print:<
        (# s: ^stream
        enter s[]
        do x->s.putint; ','->s.put; y->s.putint; INNER
        #)
   enter (x,y) exit (x,y)
   #);
 ColorPoint: Point
   (# c: @Color;
      print::< (# do s[]->c.print; INNER #)
   enter c exit c
   #)
```

Adding or specializing attributes; using I/O to refine assignment semantics; could use do-part to check/maintain invariant

Slide 15

Block structure

- Every entity (object or pattern) except the root object has *origin* in another object; the origin determines the meaning of non-local names
- 

NB: the arrows have nothing to do with attributes
- This makes patterns *complete* descriptions of objects, and introduces a distinction between “the same pattern in two different objects”

Slide 16

Why patterns depend on their context

```
Person: (# name: @text #);
University:
  (# Course:
       (# register: (# s: ^Student enter s[] .. #)#);
     Courses: @list(# element::Course; lookup: .. #);
     Student: Person
       (# go505:
            (# do this(Student)[]->
                 ('505'->Courses.lookup).register
            #)
       #)
  #);
uw,mit: @university
```

We have uw.Student## $\neq$ mit.Student##, but both $\leq$ Person##

Slide 17

Sharing of state

Many languages have "static" or "class" attributes; that is a special case of block structure: move the decl. out one level

```
(* per object *)
Person: (# .. height: @real #);

(* per class *)
University:
  (# sCount: @integer;
     Student: (# .. sCount .. #)
  #);

(* per group of classes *)
sCount: @integer;
University: (# Student: (# count: (# exit sCount #)#)#);
SelfMadeStudent: (# .. count: (# exit sCount #)#);
```

Slide 18

Virtual patterns

- A virtual attribute is a pattern, but not a compile-time constant; it may be more special in context of a more special enclosing object
- For each virtual there is a sequence of declarations of the shape `:<` `::<`* `::`$^?$ down any inheritance path for the enclosing object; "`:<`" introduces the virtual, "`::<`" further-binds it, and "`::`" final-binds it
- When the pattern is used as a method this is similar to "virtual" (overridable) methods, except that they are specialized, not overridden
- When the pattern is used as class it is similar to type parameterization of the enclosing pattern

Slide 19

A virtual pattern example

```
list:
  (# element:< object;
     max:<
       (# l,r,m: ^element
       enter (l[],r[])
       do INNER
       exit m[]
       #); ..
  #);
textList: list
  (# element::text;
     max::(#
       do (if l[]->r.less then r[]->m[]
           else l[]->m[]
          if)
       #)
  #)
```

Slide 20

Recursion and specialization of behavior

- without behavioral specialization: irrelevant
- with virtual patterns: automatic
- with simple patterns used in behavioral hierarchies: special consideration needed
- this is just a small thing—a reminder that specialization of behavior requires some new considerations

Slide 21

Recursion and specialization example

```
midRecur:
  (# test:< booleanValue;
     before:< object;
     after:< object;
     pv: ##midRecur;
     reccall: ^this(midRecur)
  do (if test then
         before;
         this(midRecur)##->pv##;
         &pv[]->reccall[]; INNER; reccall[];
         after
     if)
  #);
foo: midRecur
  (# test::(# do i>0->value #);
     i: @integer
  enter i do i-1->reccall.i
  #)
```

Slide 22

Mutually recursive (class) patterns

- Sometimes it is natural to model a class for related concepts with several levels of specialization (vehicle/operator, car/driver, limousine/chauffeur)
- for this we need to support mutually recursive families of classes which can be specialized as a whole
- virtual patterns do this rather naturally
- to preserve type safety and still allow subtyping we need inferred supertypes

Slide 23

Mutual recursion example

```
Graph:
  (# Node:< (# .. #);
     Edge:< (# from,to: ^Node .. #);
     Path:< list(# element::Node #);
     scanPaths:
       (# n1,n2: ^Node;
          current: ^Path
       enter (n1[],n2[])
       do .. ->current[]; INNER ..
       #);
     ..
  #)
```

Slide 24

Mutual recursion... (cont'd)

```
DGraph: Graph
  (# Edge::< (# len: @real #);
     Path::< (# length: (# .. #)#);
     distance:
       (# dist,d: @real
       do MAXREAL->d;
          scanPaths
          (#
          do (if (current.length->d)<dist then
                 d->dist
             if)
          #)
       exit dist
       #)
  #)
```

Slide 25

The module system

- based on *specification* and *application* of named pieces of code
- works “as-if” code was actually moved (cut’n’paste) from the specification location to the application location
- ...except for visibility: that is determined by the declared relations between modules
- very flexible granularity: can package the implementation of lots of classes together; can also single out one statement
- the placement of a piece of code determines the visible universe for it

Slide 26

Fragmentation in one file

```
-- program:descriptor --
(#
<<SLOT main:dopart>>
#)
-- main:dopart --
do 'Hello, world!'->putline
```

works (except for visibility control) just like

```
-- program:descriptor --
(#
do 'Hello, world!'->putline
#)
```

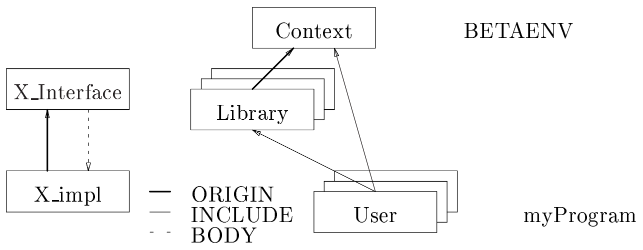

Slide 27

Multiple modules

- separate compilation and dependency management is based on the module system, as well as visibility control
- modules are related to each other by explicit declarations: `ORIGIN`, `INCLUDE`, `BODY`
- code “migrates along” `ORIGIN` paths
- both `ORIGIN` and `INCLUDE` provide visibility
- with `BODY`, the module is just included into the program, not visible, not a target for “migrating” code
- certain subgraphs are typical (interface and implementation, library, implementation-library)

Slide 28

Some typical partial fragment graphs



Slide 29

Conclusions

- The BETA programming language has emerged in an early and (perhaps?) special subculture of OO, hence everything has to be a little bit unusual
- The notion of specialization corresponds to “filling in the details” – not “adapting in whatever way you like”
- The concept of specialization of behavior is useful, even if not widely recognized
- Virtual patterns is a separate concept, not just another syntactic expression of type parameterization
- Nice things to add: multiple inheritance (gbeta), more structural equivalence (gbeta), multi-methods (howto?), ...