SSE2-PLDE_6

Contents:

- Object-Oriented Modeling, BETA language

Literature:

- Kristensen, Madsen, Møller-Pedersen.
  The when, why and why not of the BETA programming language:
  - Sec. 3.0-3.1, 4.0-4.4, 5.0-5.1

Without abstraction ...

"Without abstraction
we only know
that everything is different"

(Grady Booch, private communication)

A painting by the American Edward Hicks (1780–1849), showing the animals boarding Noah's Ark two by two.

Abstraction: Procedure/Function & Class/Method

```
P: Procedure ( Input Parameters )
(   Local Variables
    Body
);


...   P( ... )   ...
```

```
F: Function ( Input Parameters ): Result
(   Local Variables
    Body
);


 ...   F( ... )   ...
```

```
C: Class
(   Representation

    M: Method ( Input Parameters ): Result
    (   Local Variables
        Body
    );

...
);


…

O: C;
…


...   O.M( ... )   ...
```

Imperative &

```
Point: ( x, y: real );

Move: procedure (p, dx, dy)
(  dx, dy: real;  p: Point;
    p.x:= p.x+dx; p.y:= p.y+dy;
);

Distance: function (fromPoint, toPoint): real
(  fromPoint, toPoint: Point;
   dx, dy: real;
   dx:= toPoint.x-fromPoint.x;
   dy:= toPoint.y-fromPoint.y;
   return sqrt(dx*dx+dy*dy);
);


aPoint, anotherPoint:  Point;


...   Move (aPoint, 23, 45)    ...
...   Distance (aPoint, anotherPoint)   ...
```

Object-Oriented

```
Point:
(  x, y: real;

   getx: method ( ): real ( return x);
   gety: method ( ): real ( return y);

   Move: method (dx, dy)
   (  dx, dy: real;
      x:= x+dx; y:= y+dy
   );

   Distance: method (toPoint): real
   (  toPoint: Point;
      dx, dy: real;
      dx:= toPoint.getx-x;
      dy:= toPoint.gety-y;
       return sqrt(dx*dx+dy*dy);
   );
);


aPoint, anotherPoint:  Point;


...   aPoint.Move (23, 45)  ...
...   aPoint.Distance (anotherPoint)    ...
```

Imperative & Object-Oriented

```
T: (    ...   );
...



D: T;
...

P: Procedure ( Input Parameters )
(   Local Variables
    Body
);
...



...   P( ... )   ...
```

```
C: class
(
    T: (    ...   );
    ...

    D: T;
    ...

    M: Method ( Input Parameters ): Result
    (   Local Variables
        Body
    );
    ...

);
...

O: C
…

...    O.M( ... )    ...
```

Paradigms — Programming Languages

Object-oriented programming language

Object-oriented: objects + classes + inheritance

(P.Wegner 1987)

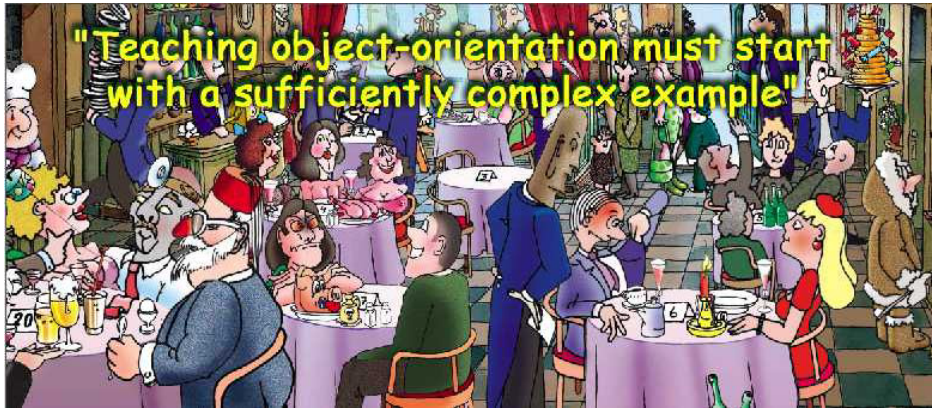
Object-oriented programming?

A program execution is regarded as a set of objects (and classes) responding to messages

Object-oriented programming

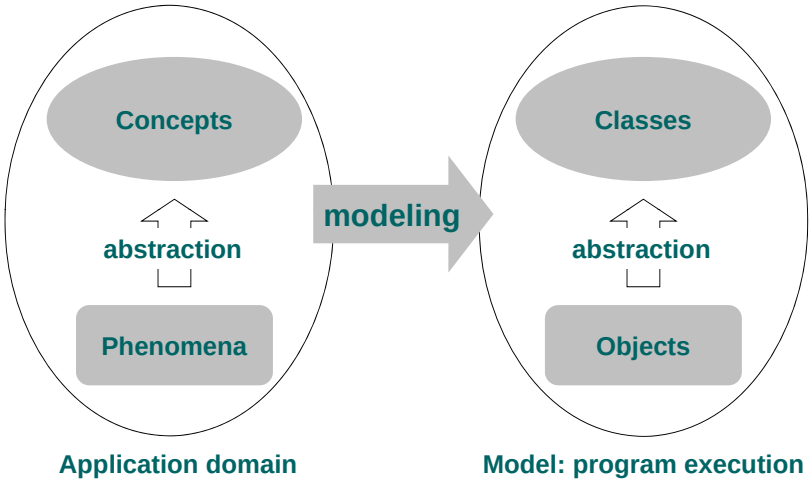
A program execution is regarded as a physical model simulating the behavior of either a real or an imaginary part of the world. Physical modeling is based upon the conception of reality in terms of phenomena and concepts

(Beta language)



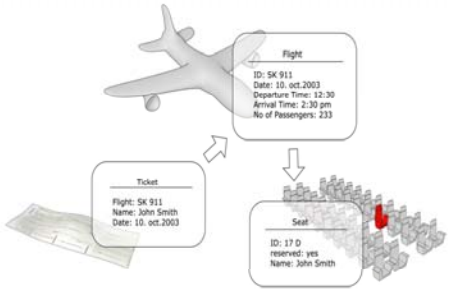
What do you see?

Modeling



Definition of OO perspective

A program execution is viewed as a physical model simulating the behavior or a real or imaginary system



- The program execution is the model
- The program text is a description of the model and application domain

Webster on modeling

actual representation *model*

A program execution **is viewed as a** physical model **simulating the behavior or a** real **or** imaginary **system**

existing *planned*

Webster: "In general a model refers to a
small,
abstract or *actual representation*
of a *planned* or *existing* object or *system*
from a *particular viewpoint*"

Concepts and Abstraction

A phenomenon is

a thing that has definite, individual existence in reality or in the mind; anything real in itself

A concept is

a generalized idea of a collection of phenomena, based on knowledge of common properties of instances in the collection

Aspects of Concepts

- Extension

  The collection of phenomena that the concept somehow covers

  Fido living at our neighbors
  Balder, the dog of the Queen of Denmark
  the famous **Lassie**

- Intention

  A collection of properties that in some way characterize the phenomena in the extension of the concept

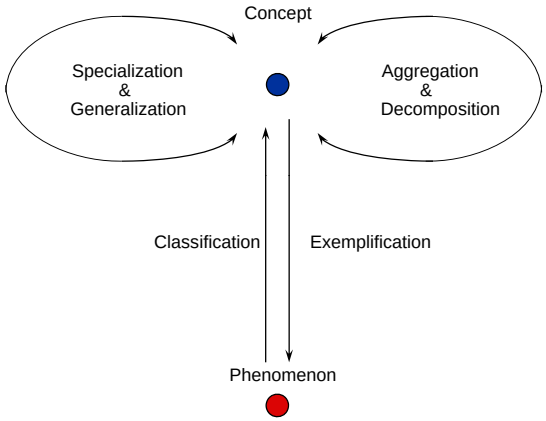
  dogs **Bark**
  dogs **have four Legs**
  dogs **. . .**

- Designation

  The collection of names by which the concept is known

  "Dog"

Abstraction Processes



Abstraction Processes

Classification: P -> C

A concrete phenomenon is related to an abstract concept.

Balder is (classified as) a Dog.

A phenomenon classified by some concept is in the extension of that concept.

Aggregation: S -> C

A (more complicated) concept is formed on the basis of (more simple) concepts.

The concepts Leg, Body, Head and Tail are aggregated to Dog.

Specialization: C -> C

A closely related concept is formed by adding more properties to the concept.

Dog may be specialized to Watchdog and to HuntingDog. Lassie is a Watchdog, but no HuntingDog; Balder is no Watchdog but may be a HuntingDog.

Only a subset of the phenomena in the extension of the general concept is in the extension of the more special concept.

Exemplification: C -> P

A concept is related to a phenomenon.

Balder is an example (an exemplification) of a Dog.

Decomposition: C -> S

The (more complicated) concept is decomposed into several (more simple) concepts.

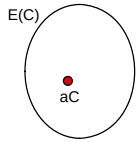
Dog is decomposed into Leg, Body, Head and Tail.

Generalization: C -> C

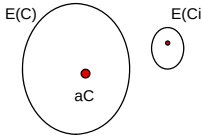
A closely related concept is formed by omitting some properties from the concept.

Dog may be generalized to Animal. Balder and also Tom and Jerry are Animals.

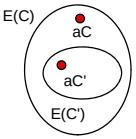
aC is **classified** by C



C is **aggregated** from C1, C2, ...



C' is **specialized** from C

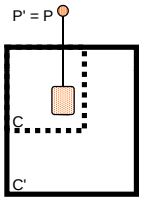


Properties: Specialization

C' is specialized from C:

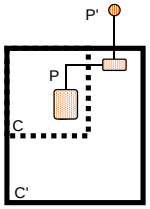
Inherited Property, P': if P' is equal to some P.
(P' is just some of the properties of C)

Car is a specialization of Vehicle and Age is a property of Vehicle:
The Age of Car is the Age of the Vehicle.



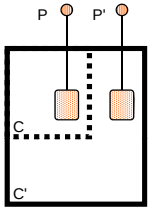
Modified Property, P': if P' is a refinement of some P.
(P' is seen as a refinement of some of the properties of C)

Wheel is a property of Vehicle and Tractor is a specialization of Vehicle:
Frontwheel and Rearwheel of Tractor are refinements of the Wheel of the Vehicle.



Additional Property, P': if P' is not an inherited or a modified property.
(P' is some of property added in the description of C' as a specialization of C)

Cab is a specialization of Car: The On/Off Sign on the Cab is an additional property.

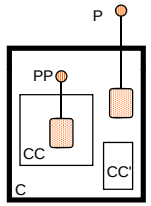


Properties: Aggregation

C is aggregated from concepts CC, CC', ...:

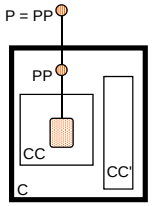
Emerging Property, P: if P is not any PP.
(P is not just some of the properties of some of the components (or parts) of C)

The Weight of a Car is the sum of the Weights for all the parts of the Car.



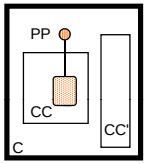
Hereditary Property, P: if P is equal to some PP.
(P is exactly the same as one of the properties of some of the components (or parts) of C)

The Color of a Car is the Color of the Body of the Car -- and no other part of the Car.



Concealed Property, PP: if PP is not any P.
(PP is a property of some part CC but it is not also a property of C)

The Make of the Tires of a Car is not the Make of the Car.
The Make of the Tires is not a property of the Car.



BETA

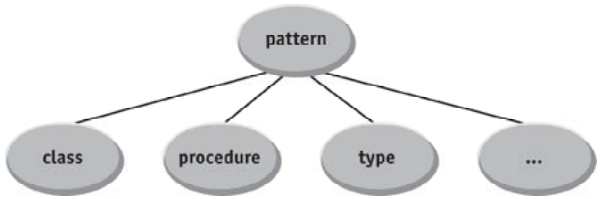
- **Pattern:**
  - **One Unifying …**
  - **Subpattern**
  - **Virtual Pattern**

- Many innovative elements (some are in contemporary object oriented languages—and some are (still) not)

- Nested Patterns
- Singular Objects
- …

- Syntax (weird ☺)

The term "pattern" was introduced before — and has nothing to do with — the term "design pattern"



"records"

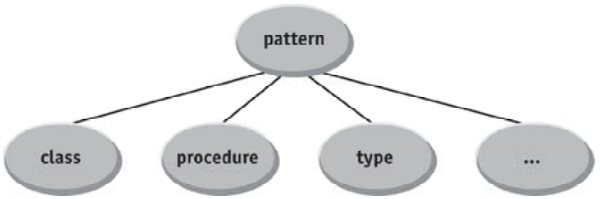
```
Point: (# x, y : @ Real #)
p:@ Point
        ... p.x ...
```

"types"

```
Point: (# x, y : @ Real
        enter ( x, y ) exit ( x, y )
        #)
p1, p2:@ Point
        ... p1 -> p2 ...
```

"procedure / function"

```
Dist : (# p1, p2 : @ Point;
          d: @ Real;
        enter ( p1, p2 )
        do   (* compute distance  d
                between p1 and p2 *)
             ...
        exit ( d )
        #)

q1, q2:@ Point
r:@ Real;
        ... (q1, q2) -> &Dist -> r ...
```



"class with methods"

```
Point:
(# x, y: @ Real;

   Move: (# dx, dy: @ Real;
         enter (dx, dy)
         do  x + dx -> x;
             y + dy -> y
         #);

                ...
#)

p : @ Point;

     ... (1, 2) -> p.Move ...
```

"control-structure"

```
Cycle: (# do
          (Loop:
              inner;
              restart Loop
          :Loop)
       #)

kb:@ Keyboard;
s:@ Screen;

     ... Cycle(# do kb.get -> s.put #) ...
```

BETA

Pattern: One, unifying abstraction mechanism

```
P: (# … #)

(# … #)
```

```
aP:@ P

aX:@ (# … #)

rP:^ P
```

Either patterns or objects
—nothing else
(except for references ☺)

```
Identifier:
(# Declaration Part

enter Enter Part

do Do Part

exit Exit part
#)
```

- `Declaration Part` is a list of attribute declarations that describes the attribute-part of the object
- `Enter Part` is a list of input parameters which may be entered prior to execution of the object
- `Do Part` is an imperative that describes the actions to be performed when the object is executed
- `Exit Part` is a list of output parameters which may be produced as a result of execution of the object

BETA

```
Record:
(# Key:@ Integer;
   Display: (# … #);
#);

Person: Record
(# Name:@ Text;
   Sex:@ …
   Display: (# … #);
#);

Employee: Person
(# Salary:@ Integer;
   Position:@ …
   Display: (# … #);
#);

Student: Person
(# Status:@ …
   Display: (# … #);
#);

Book: Record
(# Author:^ Person;
   Title:@ …
   Display: (# … #);
#);
```

```
Directory:
(# Table: [100]^Record; Top:@ Integer;
   …
   Has:
   (# Subject:@ Record; Result:@ Boolean;
   enter Subject do … Subject.Key … exit Result
   #);
   …
   Remove:
   (# Key:@ Integer; Subject:@ Record;
   enter Key do … Key … ->Subject … exit Subject
   #);
   …
   Show: (# do … Table[inx].Display … #);
#);

d:@ Directory;  r:^ Record;  p:^ Person;

… d.Show; … p->d.Has-> … ->d.Remove->r
```

BETA

Subpattern

```
P: (# … #)

PP: P (# … #)
```

```
P:
(#
   …
   Q: (# … #)
   …
#)

PP: P
(#
   …
   QQ: Q(# … #)
   …
#)
```

Anything can have a subpattern —
i.e. be specialized, extended, …

BETA

```
Record:
(# Key:@ Integer;
   Display: (# … #);
#);

Person: Record
(# Name:@ Text;
   Sex:@ …
   Display: Display(# … #);
#);

Employee: Person
(# Salary:@ Integer;
   Position:@ …
   Display: Display(# … #);
#);

Student: Person
(# Status:@ …
   Display: Display(# … #);
#);

Book: Record
(# Author:^ Person;
   Title:@ …
   Display: Display(# … #);
#);
```

```
Directory:
(# Table: [100]^Record; Top:@ Integer;
   …
   Has:
   (# Subject:@ Record; Result:@ Boolean;
   enter Subject do … Subject.Key … exit Result
   #);
   …
   Find:
   (# Key:@ Integer;
   enter Key do … Key … inner …
   #);
   Remove: Find
   (# Subject:@ Record;
   do … ->Subject … exit Subject
   #);
   …
   Show: (# do … Table[inx].Display … #);
#);

d:@ Directory;  r:^ Record;  p:^ Person;

… d.Show; … p->d.Has-> … ->d.Remove->r
```

BETA

Virtual Pattern

```
P:
(#
   …
   V:< …
   …
   aV:@ V
   …
#)

PP: P
(#
   …
   V::< …
   …
   bV:@ V
   …
#)

aPP:@ PP

aPP.bV

aPP.aV
```

```
P:
(#
   …
   V:< P
   …
   rV:^ V
   …
#)

PP: P
(#
   …
   V::< PP
   …
   sV:^ V
   …
#)

aPP:@ PP

aPP.sV

aPP.rV
```

A pattern can be partially defined only — and the become further specified in subpatterns (*further binding*)

BETA

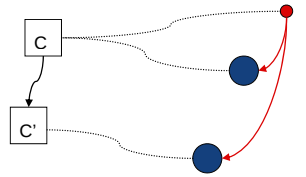
```
Record:
(# Key:@ Integer;
   Display:< (# … #);
#);

Person: Record
(# Name:@ Text;
   Sex:@ …
   Display::< (# … #);
#);

Employee: Person
(# Salary:@ Integer;
   Position:@ …
   Display::< (# … #);
#);

Student: Person
(# Status:@ …
   Display::< (# … #);
#);

Book: Record
(# Author:^ Person;
   Title:@ …
   Display::< (# … #);
#);
```

```
Directory:
(# Content:< Record
   Table: [100]^Content; Top:@ Integer;
   …
   Has:
   (# Subject:@ Content; Result:@ Boolean;
   enter Subject do … Subject.Key … exit Result
   #);
   …
   Remove: Find
   (# Subject:@ Content;
   do … ->Subject … exit Subject
   #);
   …
   Show: (# do … Table[inx].Display … #);
#);

SSE2:@ Directory (# Content::Student #);
s:^ Student;

… SSE2.Show; … s->SSE2.Has-> … ->SSE2.Remove->s …
```

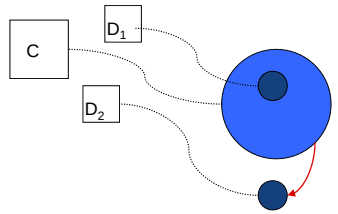
"Procedural OO" & "Conceptual OO"

Language Construct ⟷ Conceptual Modeling

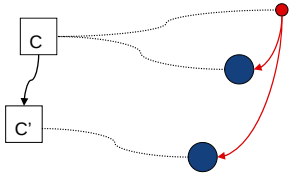
Method Overriding or Action Specialization?



Final Reference or Part Object?



Modified Property: `super(…) & inner`



Java-like

```
C: class
(  …

   m: method(…)
   ( … 2 … )

   …
)

C': C class
(  …

   m: method(…)
   (… 1 … super.m() … 3 …)

   …
)


reference aC': C';
… aC'.m (…) …
```

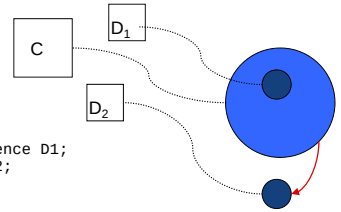
Method Overriding

BETA-like:

```
C:
(  …

   m:< (…)
   (… 1 … inner … 3 …)

   …
)

C': C
(  …

   m:< (…)
   ( … 2 … )

   …
)


aC': ^C';
… aC'.m (…) …
```

Action Specialization

Concealed Property: Part or Reference



Java-like:

```
C: class
(  …
   final aR1: reference D1;
   aR2: reference D2;
   …
)

D1: class (…);
D2: class (…);


aC: C;
… aC.aR1.…  …
… aC.aR2.…  …
```

Final Reference

BETA-like

```
C:
(  …
   aR1: @D1;
   aR2: ^D2;
   …
)

D1: (…);
D2: (…);


aC: ^C;
… aC.aR1.…  …
… aC.aR2. …  …
```

Part Object