Object-Oriented Software Construction

Bertrand Meyer
Lecture 5: Objects
The basic structure: The class

- A class is an implementation of an ADT. It is both:
  - A module.
  - A type.

- Much of the conceptual power of the method comes from the fusion of these two notions.
The basic structure: The class (cont’d)

- From the module viewpoint:
  - Set of available services (“features”).
  - Information hiding.
  - Classes may be clients of each other.

- From the type viewpoint:
  - Describes a set of run-time objects (the instances of the class).
  - Used to declare entities (≈ variables), e.g. \( x : C \)
  - Possible type checking.
  - Notion of subtype.
Avoid “objectsspeak”

- The run-time structures, some of them corresponding to “objects” of the modeled system, are **objects**.

- The software modules, each built around a **type** of objects, are **classes**.

- A system does not contain any “objects” (although its execution will create objects).
A class is an implementation of an abstract data type.

- **Instances** of the class may be created at run-time; they are objects.

Every object is an instance of a class. (In a pure O-O language such as Eiffel and Smalltalk this is true even of basic objects such as integers etc. Not true in C++ or Java where such values have special status.)

A class is characterized by **features**. Features comprise **attributes** (representing data fields of instances of the class) and **routines** (operations on instances).

- Routines are subdivided into **procedures** (effect on the instance, no result) and **functions** (result, normally no effect).

- Every operation (routine or attribute call) is relative to a distinguished object, the **current instance** of the class.
Feature categories by role

- Command
- Procedure
- No result
- Feature
- Computation
- Function
- Returns result
- Query
- Memory
- Attribute
Feature categories by implementation

Procedure

Function

Routine

No result

Returns result

Computation

Feature

Memory

Attribute

Procedure

Function

Routine

No result

Returns result

Computation

Feature

Memory

Attribute
Feature categories

- Command
- Procedure
- Feature
- No result
- Returns result
- Computation
- Memory
- Query
- Function
- Routine
- Returns result
- Computation
- Memory
- Attribute
- Memory
Alternative terminology

- Attributes are also called *instance variables* or *data member*.
- Routines are also called *methods*, *subprograms*, or *subroutines*.
- Feature call — applying a certain feature of a class to an instance of that class — is also called *passing a message* to that object.
- The notion of *feature* is particularly important as it provides a single term to cover both attributes and routines. It is often desirable not to specify whether a feature is an attribute or a routine — as expressed by the Uniform Access principle (see next).
balance = list_of_deposits.total – list_of_withdrawals.total
The Principle of Uniform Access

- Facilities managed by a module must be accessible to clients in the same way whether implemented by computation or storage.
To access a property of a point \( p1 \), the notation is the same regardless of the representation, e.g. \( p1.x \) which is applicable both in cartesian representation (\( x \) is an attribute) and in polar representation (\( x \) is a function without arguments).

In the first case the feature call is a simple field access; in the second it causes a computation to be performed.

There is no difference for clients (except possibly in terms of performance).
Abstract data type POINT

\[
\begin{align*}
    x &: \text{POINT} \rightarrow \text{REAL} \\
    y &: \text{POINT} \rightarrow \text{REAL} \\
    \rho &: \text{POINT} \rightarrow \text{REAL} \\
    \theta &: \text{POINT} \leftrightarrow \text{REAL}
\end{align*}
\]

- Class POINT: Choose a representation (polar, cartesian)
- In polar representation, \( \rho \) and \( \theta \) are attributes, \( x \) and \( y \) are routines.
A simple class

class POINT feature

  x, y: REAL
      -- Point cartesian coordinates

move (a, b: REAL) is
      -- Move by a horizontally and by b vertically.
    do
      x := x + a
      y := y + b
    end

scale (factor: REAL) is
      -- Scale by factor.
    do
      x := factor * x
      y := factor * y
    end
Class POINT (cont’d)

\[
distance (p : POINT): REAL is
  -- Distance to p
  do
    Result := \sqrt{(x - p.x)^2 + (y - p.y)^2)
  end
\]

\[
ro: REAL is
  -- Distance to origin (0, 0)
  do
    Result := \sqrt{x^2 + y^2)
  end
\]

\[
theta: REAL is
  -- Angle to horizontal axis
  do
    ...
  end
\]

end
class GRAPHICS feature
    p, q: POINT
    ...  
    some_routine is
       -- Use p and q.
       local
       u, v: REAL
       do
          -- Creation instructions
          create p
          create q
    end
class GRAPHICS

feature

   p, q: POINT

   ... -- Graphic points

   some_routine is

      -- Use p and q.

      local

      u, v: REAL

      do

         -- Creation instructions

         create p
         create q

         p.move (4.0, -2.0)

         -- Compare with Pascal, C, Ada:
         -- Move (p, 4.0, -2.0)
class GRAPHICS feature
  p, q: POINT

  ... some_routine is
  -- Use p and q.
  local
    u, v: REAL
  do
    -- Creation instructions
    create p
    create q
    p.move (4.0, -2.0)
    p.scale (0.5)

end
end
class GRAPHICS feature

p, q: POINT

-- Graphic points
...

someRoutine is
-- Use p and q.

local

u, v: REAL

do
-- Creation instructions

create p
create q

p.move (4.0, -2.0)
-- Compare with Pascal, C, Ada:
-- Move (p, 4.0, -2.0)

p.scale (0.5)

u := p.distance (q)
v := p.x

p := q

end

end
Use of the class in a client (5/5)

class GRAPHICS feature
    p, q: POINT
    ... some_routine is
    local u, v: REAL
    do
        -- Creation instructions
        create p
        create q
        p.move (4.0, -2.0)
        -- Compare with Pascal, C, Ada:
        -- Move (p, 4.0, -2.0)
        p.scale (0.5)
        u := p.distance (q)
        v := p.x
        p := q
        p.scale (-3.0)
    end
end
The module-type merge

- A class is both:
  - A module
  - A type

- Much of the conceptual power of the method comes from the fusion of these two notions.

- From the module viewpoint:
  - Set of available services ("features").

- From the type viewpoint:
  - Description of set of possible run-time objects (its instances).

- Connection: The services of the class, viewed as a module, are the operations applicable to the instances of the class, viewed as a type.
Forms of assignment and copy

- Reference assignment (\(a\) and \(b\) of reference types):
  \[
  b := a
  \]

- Object duplication (shallow):
  \[
  c := \text{clone} (a)
  \]

- Object duplication (deep):
  \[
  d := \text{deep\_clone} (a)
  \]

- Also: shallow field-by-field copy (no new object is created):
  \[
  e.copy (a)
  \]
Shallow and deep cloning

- Initial situation:

- Result of:

\[ b := a \]

\[ c := \text{clone} (a) \]

\[ d := \text{deep\_clone} (a) \]
A related mechanism: Persistence

```plaintext
a.store (file)
```

...  

```plaintext
b ?= retrieved (file)
```

- Storage is automatic.
- Persistent objects identified individually by keys.

- These features come from the library class `STORABLE`. The above is only an approximate form (see typing discussion).
Privileges of a client $C$ of a class $A$ on an attribute $attrib$:
- Read access if attribute is exported.

Assuming $a1 : A$
Then $a1.attrib$ is an expression.

An assignment such as $a1.attrib := a2$ is syntactically illegal!
(You cannot assign a value to an expression, e.g. $x + y$.)
The privileges of a client

- Secret
- Read-only
- Read, restricted write
- Full write
Applying abstraction principles

- Beyond read access: full or restricted write, through exported procedures.

- Full write privileges: `set_attribute` procedure, e.g.

```plaintext
set_temperature (u: REAL) is
  -- Set temperature value to u.
  do
    temperature := u
  ensure
    temperature_set: temperature = u
  end
```

- Client will use e.g. `x.set_temperature (21.5)`.
set_temperature (u: REAL) is
   -- Set temperature value to u.
   require
      not_under_minimum: u >= -273
      not_above_maximum: u <= 2000
   do
      temperature := u
      update_database
   ensure
      temperature_set: temperature = u
   end
- Allow

\[ x.temperature := 21.5 \]

if there is a “setter”:

```csharp
private int temperature_internal;
public int temperature
{
    get { return temperature_internal; }
    set {
        temperature_internal = value;
        //... Other instructions; ...
    }
}
```
Information hiding

class A

feature

  f ... 
  g ...

feature {NONE}

  h ...

feature {B, C}

  j ...

feature {A, B, C}

  k

class A

In clients, with the declaration $a1: A$, we have:

- $a1.f$, $a1.g$: valid in any client
- $a1.h$: invalid anywhere (including in $A$'s own text).
- $a1.j$: valid only in $B$, $C$ and their descendants (not valid in $A$!)
- $a1.k$: valid in $B$, $C$ and their descendants, as well as in $A$ and its descendants
Information hiding (cont’d)

- Information hiding only applies to use by clients, using dot notation or infix notation, as with \texttt{a1.f} (“Qualified calls”).

- Unqualified calls (within the class itself) are not subject to information hiding:

  ```
  class A
  feature \{NONE\}
    \$\texttt{h}\$ is
      -- Does something.
      do
      \$\texttt{...}\$
      end
  feature
    \$\texttt{f}\$ is
      -- Use \$\texttt{h}\$.
      do
      \$\texttt{...}\$
      \$\texttt{h}\$
      end
  end
  ```
The dynamic model

- States of a reference:
  - \( p := q \) (where \( q \) is attached)
  - \( p := \) Void
  - \( p := q \) (where \( q \) is void)

- Operations on references:

  create \( p \)
  \[ p := q \]
  \[ p := \text{Void} \]
  if \( p = \text{Void} \) then ...
Creating an object

- With the class `POINT` as given:
  
  ```java
  my_point: POINT
  ...
  create my_point
  ```

- Effect of such a creation instruction:
  - Allocate new object of the type declared for `my_point`.
  - Initialize its fields to default values (0 for numbers, false for booleans, null for characters, void for references).
  - Attach it to the instruction’s target, here `my_point`. 
class POINT

create

make_cartesian, make_polar

feature -- Initialization

make_cartesian (a, b: REAL) is
-- Initialize to abscissa a, ordinate b.
do
x := a
y := b
end

make_polar ...

feature

... The rest as before ...

Chair of Software Engineering
If there is a creation clause

- Creation instructions must be “creation calls”, such as
  
  ```
  create my_point.make_polar (1, Pi / 2)
  ```
If there is no creation clause

- An absent creation clause, as in

```plaintext
class POINT
    -- No creation clause
feature
    ... The rest as before ...
end
```

is understood as one that would only list `default_create`, as if it had been written

```plaintext
class POINT
create
default_create
feature
    ... The rest as before ...
end
```

- Procedure `default_create` is defined in `ANY` as doing nothing; any class can redefine it to provide proper default initializations.
The notation

\texttt{create x}

is understood (if permitted) as an abbreviation for

\texttt{create x.default_create}
To allow both forms

- To make both forms valid:
  
  ```
  create my_point
  ``
  as well as
  ```
  create my_point.make_polar (1, Pi / 2)
  ```

  it suffices to make `default_create` (redefined or not) one of the creation procedures:

  ```
  class POINT
  create
  make_cartesian, make_polar, default_create
  feature
  ... The rest as before ...
  ```
To prohibit instantiating a class

```plaintext
class NOT_CREATABLE
create
   -- Nothing here!
feature
   ... The rest as before ...
end
```
An object is a machine
An object has an interface
A List

1. **Cursor**
   - **before**
   - **after**
   - **count**
   - **before**
   - **after**

2. **item**
   - **“Meyer”**
   - **index**

3. **Cursor**
   - **forth**
   - **back**
   - **finish**
   - **start**
An object has an implementation

- start
- forth
- put_right

- item
- index
- before
- after
Information hiding

start
forth
put_right

item
index
before
after
What to do with unreachable objects

- Reference assignments may make some objects useless.

- Two possible approaches:
  - Manual reclamation (e.g. C++, Delphi).
  - Automatic garbage collection (e.g. Eiffel, Smalltalk, Simula, Java, .NET)
Arguments for automatic collection

- Manual reclamation is dangerous. Hampers software reliability.
- In practice bugs arising from manual reclamation are among the most difficult to detect and correct. Manifestation of bug may be far from source.
- Manual reclamation is tedious: need to write “recursive dispose” procedures.
- Modern garbage collectors have acceptable overhead (a few percent) and can be made compatible with real-time requirement.
- GC is tunable: disabling, activation, parameterization....
Properties of a garbage collector (GC)

- **Consistency** (never reclaim a reachable object).
- **Completeness** (reclaim every unreachable object – eventually).

- Consistency (also called safety) is an absolute requirement. Better no GC than an unsafe GC.
- But: safe automatic garbage collection is hard or impossible in a hybrid language environment (e.g. C++): pointers may masquerade as integers or other values.
Types

- Reference types; value of an entity is a reference. Example:
  \[b: \text{POINT}\]

- Expanded types; value of an entity is an object. Example:
  \[d: \text{expanded POINT}\]
## Expanded classes

- A class may also be declared as
  \textbf{expanded class} \( C \)
  
  \ldots The rest as usual \ldots

- Then you can declare:
  
  \( a: C \)
  
  with the same effect as
  
  \( b: \text{expanded } C \)
  
  in the earlier syntax (still permitted, with same meaning).
Expanded classes and entities support the notion of subobject.

```plaintext
class RECTANGLE_R
feature
  corner1, corner2, corner3, corner4: POINT
  ...
end

class RECTANGLE_E
feature
  corner1, corner2, corner3, corner4: expanded POINT
  ...
end
```
The meaning of expanded classes

- More than an implementation notion: a system modeling tool.

- Two forms of client relation:
  - Simple client
  - Expanded client

- What is the difference between these two statements?
  - A car has an originating factory.
  - A car has an engine.
expanded class INTEGER ...
expanded class BOOLEAN ...
expanded class CHARACTER ...
expanded class REAL ...
expanded class DOUBLE ...

n: INTEGER
Infix and prefix operators

expanded class INTEGER

feature

infix "+" (other: INTEGER): INTEGER is
  -- Sum with other
  do
    ...
  end

infix "*" (other: INTEGER): INTEGER is
  -- Product by other
  do
    ...
  end

prefix "-": INTEGER is
  -- Unary minus
  do
    ...
  end

Calls are then of the form $i + j$ rather than $i.plus(j)$. 
End of lecture 5