Object-Oriented Software Construction

Bertrand Meyer

Lecture 4: 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 “objectspeak”

- 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
      - Query
        - Returns result
          - Computation
            - Memory
              - Attribute
                - Function
                  - Returns result
Feature categories by implementation

- Procedure
  - No result
  - Routine
  - Returns result

- Function

- Attribute

- Computation
- Feature
- Memory
Feature categories

Command → Procedure

No result

Feature

Returns result

Query

Function → Attribute

Computation

Memory

Routine

Computation

Memory

Attribute

No result

Feature

Returns result

Computation

Memory
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.
Uniform access through feature call

- 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

\[ x: \text{POINT} \rightarrow \text{REAL} \]
\[ y: \text{POINT} \rightarrow \text{REAL} \]
\[ \rho: \text{POINT} \rightarrow \text{REAL} \]
\[ \theta: \text{POINT} \leftrightarrow \text{REAL} \]

- Class \textit{POINT}: Choose a representation (polar, cartesian)
- In polar representation, \( \rho \) and \( \theta \) are attributes, \( x \) and \( y \) are routines.
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 (continued)

    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
Use of the class in a client

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

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
    p.move (4.0, -2.0)
    -- Compare with Pascal, C, Ada:
    -- move (p, 4.0, -2.0)
  end
end
Use of the class in a client

```pascal
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)
    -- Compare with Pascal, C, Ada:
    -- move (p, 4.0, -2.0)
    p.scale (0.5)

end
end
```

```
end
```

---

*Chair of Software Engineering*

OOSC - Summer Semester 2005
Use of the class in a client

```null
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

end
end
```

Use of the class in a client

```plaintext
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
```

Diagram:

- `p` as a POINT with coordinates (2.0, -1.0)
- `q` as a POINT with coordinates (0.0, 0.0)
Variants of assignment and copy

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

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

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

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

Initial situation:

Result of:

\[ b := a \]

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

\[ d := \text{deep\_clone} \ (a) \]
Where do these mechanisms come from?

- Class *ANY* in the Eiffel “Kernel Library”

- Every class that doesn’t explicitly inherit from another is considered to inherit from *ANY*

- As a result, every class is a descendant of *ANY*. 
Completing the inheritance structure
A related mechanism: Persistence

\[ a.\text{store} \ (\text{file}) \]

\[ b \ ?= \text{retrieved} \ (\text{file}) \]

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

- These features come from the library class \textit{STORABLE}. 
Applying abstraction principles

- 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. $x + y$.)

Chair of Software Engineering

OOSC - Summer Semester 2005
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.

  \[
  \text{set_temperature (u: REAL) is} \\
  \text{-- Set temperature value to } u. \\
  \text{do} \\
  \text{temperature := u} \\
  \text{ensure} \\
  \text{temperature_set: temperature = u} \\
  \text{end}
  \]

- Client will use e.g. \[x.set\_temperature (21.5).\]
**Setter procedures**

- `set_attribute` procedure, e.g.
  
  ```
  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)`.

- Client cannot directly assign to attribute
Other uses of a setter procedure

\begin{verbatim}
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 (u, Current)
  ensure
    temperature_set: temperature = u
end
\end{verbatim}
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; ...
    }
}
```
In clients, with the declaration \texttt{a1: A}, we have:

- \texttt{a1.f, a1.g}: valid in any client

- \texttt{a1.h}: invalid anywhere
  (including in \texttt{A}'s own text).

- \texttt{a1.j}: valid only in \texttt{B, C} and their descendants
  (not valid in \texttt{A}!)

- \texttt{a1.k}: valid in \texttt{B, C} and their descendants,
  as well as in \texttt{A} and its descendants
Information hiding

- Information hiding only applies to use by clients, using dot notation or infix notation, as with $a1.f$ ("Qualified calls").

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

```plaintext
class A
  feature {NONE}
    h is
      -- Does something.
      do
        ...
      end
  feature
    f is
      -- Use h.
      do
        ...
      h
      end
  end
```

---

*Chair of Software Engineering*  
OOSC - Summer Semester 2005
The dynamic model

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

- Operations on references:
  - create \( p \)
  - \( p := q \)
  - \( p := \text{Void} \)
  - if \( p = \text{Void} \) then ...

---

Chair of Software Engineering

OOSC - Summer Semester 2005
Creating an object

- With the class `POINT` as given:
  
  ```
  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 {NONE} -- 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 ...
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
default_create
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
default_create
create
    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.
Associated convention

- The notation
  \texttt{create } x

is understood (if permitted) as an abbreviation for
\texttt{create } x.\texttt{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 ...
  ```
- 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.
Reference types; value of an entity is a reference. Example:
\[ b: POINT \]

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} \textit{C} 
  
  ... The rest as usual ...

- Then you can declare:

  \begin{verbatim}
  a: C
  \end{verbatim}

  with the same effect as

  \begin{verbatim}
  b: \textbf{expanded} C
  \end{verbatim}

  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.
Basic types as expanded classes

expanded class INTEGER …
expanded class BOOLEAN …
expanded class CHARACTER …
expanded class REAL …
expanded class DOUBLE …

n: INTEGER
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

... 
end

Calls are then of the form $i + j$ rather than $i\text{.plus}\ (j)$.
End of lecture 4