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.
    \[ x \in C \]
  - 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).
Terminology

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 implementation

Feature categories by role

Feature categories
**Alternative terminology**

- Attributes are also called *instance variables* or *data members*.
- 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).

---

**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**

\[
\text{balance} = \text{list-of-deposits.total} - \text{list-of-withdrawals.total}
\]

- A1: `list_of_deposits`  
  `list_of_withdrawals`  
  `balance`

- A2: `list_of_deposits`  
  `list_of_withdrawals`

---

**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

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

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

Class POINT (continued)

\[ \text{distance} (p : \text{POINT}) : \text{REAL} \text{ is} \]
\[ \begin{align*}
\text{do} & \\
\text{Result} & := \sqrt{(x - p.x)^2 + (y - p.y)^2} \\
\text{end}
\end{align*} \]

\[ \text{ro} : \text{REAL} \text{ is} \]
\[ \begin{align*}
\text{do} & \\
\text{Result} & := \sqrt{x^2 + y^2} \\
\text{end}
\end{align*} \]

\[ \text{theta} : \text{REAL} \text{ is} \]
\[ \begin{align*}
\text{do} & \\
& \ldots \\
\text{end}
\end{align*} \]

Use of the class in a client

\[ \text{class GRAPHICS feature} \]
\[ p, q : \text{POINT} \text{ is} \]
\[ \ldots \]
\[ \text{some_{routine}} : \text{Use p and q,} \]
\[ \text{local} \]
\[ u, v : \text{REAL} \]
\[ \begin{align*}
\text{do} & \\
& \text{Creation instructions} \\
& \text{create } p \\
\end{align*} \]
Use of the class in a client

```plaintext
class GRAPHICS feature
  p, q: POINT
  -- Graphic points
  -- same_routine is --- Use p and q.
    local
    u, v: REAL
      -- Creation instructions
      create p
      create q
      p.move (4.0, -2.0)
      v := p.distance (o)
      p := q
      p.scale (0.5)
    end
    do
      end

2.0
-1.0

0.0
0.0

(POINT)

 Variants of assignment and copy

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

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

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

- Also: shallow field-by-field copy (no new object is created):
  \[ e.copy \left( a \right) \]

Shallow and deep cloning

Initial situation:

Result of:

\[ b := a \]

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

\[ d := \text{deep\_clone} \left( a \right) \]

Where do these mechanisms come from?

- Class \textit{ANY} in the Eiffel “Kernel Library”

- Every class that doesn’t explicitly inherit from another is considered to inherit from \textit{ANY}

- As a result, every class is a descendant of \textit{ANY}. 

Completing the inheritance structure

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.)

A related mechanism: Persistence

- a.store (file)
  ....
- b ?= retrieved (file)

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

- These features come from the library class STORABLE.

The privileges of a client

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

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

Other uses of a setter procedure

```
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
```

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

Delphi/C# “properties”

- Allow
  
  ```
  x.temperature = 21.5;
  ```
  
  if there is a “setter”:
  
  ```
  private int temperature_internal;
  public int temperature
  {
    get {return temperature_internal; }
    set {
      temperature_internal = value;
      //... Other instructions; ...
    }
  }
  ```
Information hiding

- 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

THE DYNAMIC MODEL

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

- Operations on references:
  `create p`
  `p := q`
  `p := Void`
  `if p = Void then ...`

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:

```
class A
  feature (NONE)
    h is
      -- Does something.
    do
    end
  feature f is
    -- Use h.
    do
    end
end
```

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`.
Specific creation procedures

```plaintext
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 no creation clause

- An absent creation clause, as in
  ```plaintext
  class POINT
  feature ...
  ```

If there is a creation clause

- Creation instructions must be "creation calls", such as
  ```plaintext
  create my_point.make_polar (1, PI / 2)
  ```

Associated convention

- The notation
  ```plaintext
  create x
  ```
  is understood (if permitted) as an abbreviation for
  ```plaintext
  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 ...
  ```

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....

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)

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:
  \[ d: \text{expanded POINT} \]

Subobjects

- Expanded classes and entities support the notion of subobject.
  
  ```
  class \text{RECTANGLE}_R
  feature
  \text{corner1, corner2, corner3, corner4: POINT}
  ...
  end
  
  class \text{RECTANGLE}_E
  feature
  \text{corner1, corner2, corner3, corner4: expanded POINT}
  ...
  end
  ```

Expanded classes

- A class may also be declared as
  ```
  \text{expanded class C}
  ```
  ... The rest as usual ...

- Then you can declare:
  ```
  a: C
  ```
  with the same effect as
  ```
  b: \text{expanded C}
  ```
  in the earlier syntax (still permitted, with same meaning).

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

Infix and prefix operators

expanded class INTEGER feature
  infix "+" (other: INTEGER): INTEGER is
    do
      -- Sum with other
    end

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

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

end ...

Calls are then of the form i + j rather than i.plus (j).