Object technology

- Source: Simula 67 language, Oslo, mid-sixties
- Spread very slowly in the seventies
- Smalltalk, developed at Xerox PARC in late seventies, made O-O hip by combining it with visual technologies
- First OOPSLA conference in 1986 revealed O-O to the unwashed masses
- Spread quickly in 1990s through O-O languages like Objective C, C++, Eiffel, Java, as well as O-O tools, O-O databases, O-O analysis...
- Largely accepted today
- Non O-O approaches are also called "procedural".
Eiffel

- Dates back to 1985 in first version
- Constantly refined and improved since then
- Fully object-oriented; not a hybrid with other approaches
- Focuses on quality, especially reliability, extendibility and reusability
- Emphasizing simplicity
- Used for many mission-critical projects in industry (see next)
- Based on concepts of “Design by Contract”.
- Implementations: from Eiffel Software, Object Tools, University of Nancy (“SmartEiffel”)
- International standard in preparation through ECMA

Large Eiffel projects in industry

- Chicago Board of Trade
- AMP Investments
- Lockheed Martin
- Hewlett Packard
- Cap Gemini Ernst & Young
- AXA Rosenberg
- Environmental Protection Agency
- EMC
- Swedish National Health Board
- ENEA
- Boeing
- Northrop Grumman

Why use Eiffel?

- Simple, clean O-O model
- Enables you to focus on concepts, not language
- Little language “baggage”
- Development environment (EiffelStudio)
- Portability: Windows / Linux & others
- Prepares you to learn other O-O languages, e.g. C++, Java, C# (assuming you ever want to)
Scaling up

- One of the toughest issues in learning software is to find solutions that work well both "in the small" and "in the large".

- That's the goal for the techniques we teach in this course.

Classes: The fundamental structure

- Merging of the notions of module and type:
  - Module = Unit of decomposition: set of services
  - Type = Description of a set of run-time objects ("instances" of the type)

- The connection:
  - The services offered by the class, viewed as a module, are the operations available on the instances of the class, viewed as a type.

Class relations

- Two relations:
  - Client
  - Heir
Overall system structure

A command

A query
MP2  Picture to be included
Michela Pedroni; 10.10.2003
The class

- 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.

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.

Bad use of terminology: "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).
Feature categories

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

A command

A query
Picture to be included
Command-query separation principle

Queries should not change the state

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).

Uniform Access: An example

balance = list_of_deposits.total – list_of_withdrawals.total

(A1) list_of_deposits

(A2) list_of_withdrawals

balance

Ada, Pascal, C/C++, Java, C#: Simula, Eiffel:

(a) a.balance
(balance (a)) a.balance()
### 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

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

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

### Class POINT

```plaintext
class POINT
feature
  x, y: REAL -- Point cartesian coordinates
  move (a, b: REAL) is
    do
      x := x + a
      y := y + b
    end
  scale (factor: REAL) is
    do
      x := factor * x
      y := factor * y
    end
end
```

**Class POINT (continued)**

```pascal
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**

```pascal
class GRAPHICS
feature
  p, q: POINT
  -- Graphic points
  ...
  some_routine is
    local
      u, v: REAL
    do
      create p
      create q
      ...
    end
end
```

```pascal
class GRAPHICS
feature
  p, q: POINT
  -- Graphic points
  ...
  some_routine is
    local
      u, v: REAL
    do
      create p
      create q
      ...
    end
end
```
class GRAPHICS feature
  p, q: POINT  -- Graphic points
  some_routine be
    local  
      u, v: REAL
      do
        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

```pascal
class GRAPHICS feature
  p, q: POINT  -- Graphic points
procedure someroutine be
  local
    u, v: REAL
  begin
    create p
    create q
    p.move (4.0, -2.0)  -- Compare with Pascal, C, Ada:
    p.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.

Applying abstraction principles

- Privileges of a client \( C \) of a class \( A \) on an attribute \( \text{attrib} \):
  - Read access if attribute is exported.

- Assuming \( a1 : A \)
  - Then \( a1.\text{attrib} \) is an expression.
- An assignment such as \( a1.\text{attrib} := a2 \) is syntactically illegal! (You cannot assign a value to an expression, e.g. \( x + y \).)
The privileges of a client

- Read-only
- Secret
- Read, restricted write
- Full write
- Full write

Applying abstraction principles

- Beyond read access: full or restricted write, through exported procedures.
- Full write privileges: set_attribute procedure, e.g.

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

  \(\text{Client will use e.g. x.set_temperature (21.5).}\)

Other uses of a setter procedure

\[
\text{set_temperature (u: REAL) is} \\
\text{-- Set temperature value to u.} \\
\text{require} \\
\text{not_under_minimum: u >= -273} \\
\text{not_above_maximum: u <= 2000} \\
\text{do} \\
\text{temperature :} = \text{u} \\
\text{update_database} \\
\text{ensure} \\
\text{temperature_set: temperature = u} \\
\text{end}
\]
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

- 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 do ...
  end
  feature (B, C)
  j do ...
  end
  feature (A, B, C)
  k
  end
  ```
  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:
  - VOID
  - ATTACHED

- Operations on references:
  - create $p$
  - $p := q$ (where $q$ is attached)
  - $p := \text{Void}$ (where $q$ is void)

Creating an object

- With the class \texttt{POINT} as given:
  - \texttt{my\_point: POINT}
  - ...\texttt{create my\_point}

- Effect of such a creation instruction:
  - Allocate new object of the type declared for \texttt{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 \texttt{my\_point}.

Specific creation procedures

\begin{verbatim}
class POINT
create
  make_cartesian, make_polar
feature
  -- Initialization
  make_cartesian (a, b: REAL) is
  do
    x := a
    y := b
  end
  make_polar ...
feature
  ... The rest as before ...
\end{verbatim}
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
  ```
  class POINT
    feature
      ... The rest as before ...
  end
  ```
  is understood as one that would only list default_create, as if it had been written
  ```
  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.

Associated convention

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

class NOT_CREATABLE
create
  -- Nothing here!
feature
  ... The rest as before ...
end

An object is a machine
An object has an interface

A list

An object has an implementation
Information hiding

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
  
  \[ \text{expanded class C} \]
  
  ... 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).
Subobjects

- 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

```plaintext
expanded class INTEGER ...
expanded class BOOLEAN ...
expanded class CHARACTER ...
expanded class REAL ...
expanded class DOUBLE ...

n: INTEGER
```
Complementary material

- OOSC2:
  - Chapter 7: The static structure: classes
  - Chapter 8: The run-time structure: objects

End of lecture 5