

Objects

Akim Demaille, Etienne Renault, Roland Levillain

April 8, 2019

Part I

Object Oriented History

Table of Contents

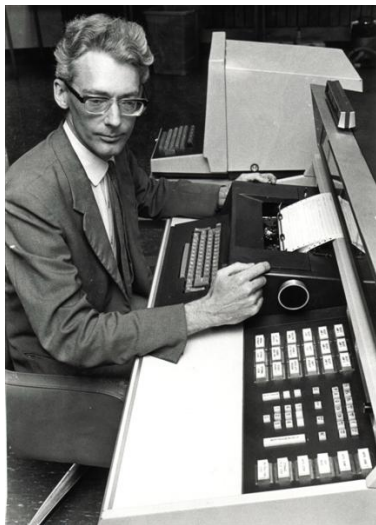
- 1 Simula
- 2 Smalltalk
- 3 The mysterious language

Table of Contents

- 1 Simula
 - People Behind SIMULA
 - SIMULA I
 - Simula 67
- 2 Smalltalk
- 3 The mysterious language

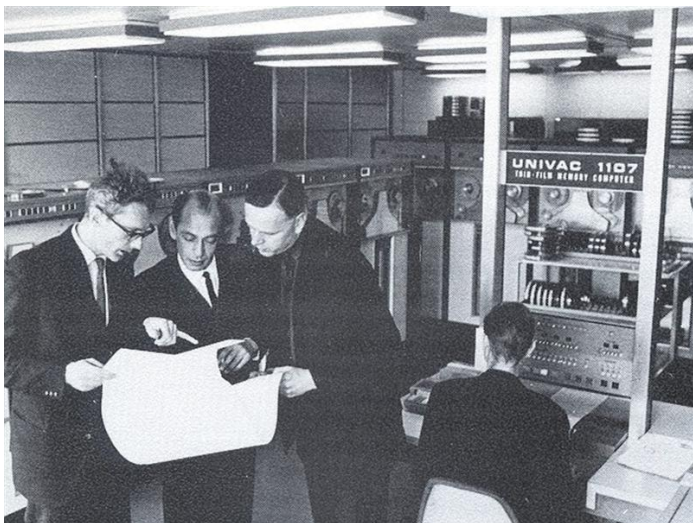


Ole-Johan Dahl



Ole-Johan Dahl

Simula



Dahl & Nygaard

Simula



Ole-Johan Dahl & Kristen Nygaard (ca. 1963)



Nygaard & Dahl: Turing Award 2001

2002... Sad Year

Ole-Johan Dahl



Oct 12, 1931,
Mandal, NO

June 29, 2002, Asker,
NO

"...are there too many
basic mechanisms
floating around doing
nearly the same
thing?"

Kristen Nygaard



Aug 27, 1926, Oslo,
NO

Aug 10, 2002, Oslo,
NO

"To program is to
understand!"

**Edsger Wybe
Dijkstra**



May 11, 1930,
Rotterdam, NL

Aug 06, 2002,
Nuenen, NL

"Do only what only
you can"

Table of Contents

- 1 Simula
 - People Behind SIMULA
 - **SIMULA I**
 - Simula 67
- 2 Smalltalk
- 3 The mysterious language

Simula

In the spring of 1967 a new employee at the NCC in a very shocked voice told the switchboard operator: “two men are fighting violently in front of the blackboard in the upstairs corridor. What shall we do?” The operator came out of her office, listened for a few seconds and then said: “Relax, it’s only Dahl and Nygaard discussing SIMULA.” — Kristen Nygaard, Ole-Johan Dahl.

Physical system models. Norwegian nuclear power plant program.

Simula

In the spring of 1967 a new employee at the NCC in a very shocked voice told the switchboard operator: “two men are fighting violently in front of the blackboard in the upstairs corridor. What shall we do?” The operator came out of her office, listened for a few seconds and then said: “Relax, it’s only Dahl and Nygaard discussing SIMULA.” — Kristen Nygaard, Ole-Johan Dahl.

Physical system models. Norwegian nuclear power plant program.

Process oriented discrete simulation language based on Algol 60.
(1964 - 1965) **Simulation language.**

Basic concepts (1/2)

- A **system**, consisting of a finite and fixed number of active components named **stations**, and a finite, but possibly variable number of passive components named **customers**.

Basic concepts (1/2)

- A **system**, consisting of a finite and fixed number of active components named **stations**, and a finite, but possibly variable number of passive components named **customers**.
- A **station** consisting of two parts: a queue part and a service part. Actions associated with the service part, named the **station's operating rule**, were described by a sequence of ALGOL (or ALGOL-like) statements.

Basic concepts (1/2)

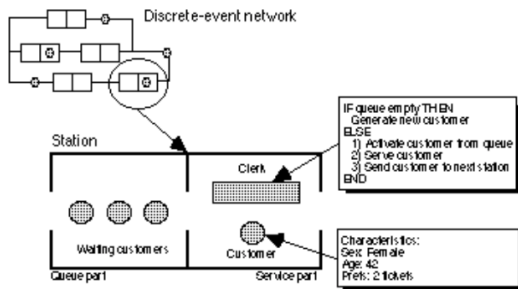
- A **system**, consisting of a finite and fixed number of active components named **stations**, and a finite, but possibly variable number of passive components named **customers**.
- A **station** consisting of two parts: a queue part and a service part. Actions associated with the service part, named the **station's operating rule**, were described by a sequence of ALGOL (or ALGOL-like) statements.
- A customer with no operating rule, but possibly a finite number of variables, named **characteristics** .

Basic concepts (1/2)

- A **system**, consisting of a finite and fixed number of active components named **stations**, and a finite, but possibly variable number of passive components named **customers**.
- A **station** consisting of two parts: a queue part and a service part. Actions associated with the service part, named the **station's operating rule**, were described by a sequence of ALGOL (or ALGOL-like) statements.
- A customer with no operating rule, but possibly a finite number of variables, named **characteristics** .
- A real, continuous variable called **time** and a **function position**, defined for all customers and all values of time.

Basic concepts (2/2)

This structure may be regarded as a **network**, and the events (actions) of the stations' service parts are regarded as **instantaneous and occurring at discrete points of time**, this class of systems was named **discrete event networks**.



Simula I

- An ALGOL 60 preprocessor
- A subprogram library
- An original per “process” stack allocation scheme

Not yet the concept of objects.

Quasi-parallel processing is analogous to the notion of coroutines described by Conway in 1963.

Table of Contents

- 1 Simula
 - People Behind SIMULA
 - SIMULA I
 - Simula 67
- 2 Smalltalk
- 3 The mysterious language

Simula 67

- Introduces:

Simula 67

- Introduces:
 - ▶ the concept of **object**,

Simula 67

- Introduces:
 - ▶ the concept of **object**,
 - ▶ the concept of **class**,

Simula 67

- Introduces:
 - ▶ the concept of **object**,
 - ▶ the concept of **class**,
 - ▶ literal objects (**constructors**),

Simula 67

- Introduces:
 - ▶ the concept of **object**,
 - ▶ the concept of **class**,
 - ▶ literal objects (**constructors**),
 - ▶ the concept of **inheritance**
(introduced by C. A. R. Hoare for records),

Simula 67

- Introduces:
 - ▶ the concept of **object**,
 - ▶ the concept of **class**,
 - ▶ literal objects (**constructors**),
 - ▶ the concept of **inheritance**
(introduced by C. A. R. Hoare for records),
 - ▶ the concept of **virtual method**,

Simula 67

- Introduces:
 - ▶ the concept of **object**,
 - ▶ the concept of **class**,
 - ▶ literal objects (**constructors**),
 - ▶ the concept of **inheritance**
(introduced by C. A. R. Hoare for records),
 - ▶ the concept of **virtual method**,
 - ▶ **attribute hiding**!

Simula 67

- Introduces:
 - ▶ the concept of **object**,
 - ▶ the concept of **class**,
 - ▶ literal objects (**constructors**),
 - ▶ the concept of **inheritance**
(introduced by C. A. R. Hoare for records),
 - ▶ the concept of **virtual method**,
 - ▶ **attribute hiding**!
- Immense funding problems
steady support from C. A. R. Hoare, N. Wirth and D. Knuth.

Simula 67

- Introduces:
 - ▶ the concept of **object**,
 - ▶ the concept of **class**,
 - ▶ literal objects (**constructors**),
 - ▶ the concept of **inheritance**
(introduced by C. A. R. Hoare for records),
 - ▶ the concept of **virtual method**,
 - ▶ **attribute hiding**!
- Immense funding problems
steady support from C. A. R. Hoare, N. Wirth and D. Knuth.
- Standardized ISO 1987.

Shape in Simula (1/5)

```
class Shape(x, y); integer x; integer y;
virtual: procedure draw is procedure draw;;
begin
  comment -- get the x & y components for the object --;
  integer procedure getX;
    getX := x;
  integer procedure getY;
    getY := y;
  comment -- set the x & y coordinates for the object --;
  integer procedure setX(newx); integer newx;
    x := newx;
  integer procedure setY(newy); integer newy;
    y := newy;
  comment -- move the x & y position of the object --;
  procedure moveTo(newx, newy); integer newx; integer newy;
    begin
      setX(newx);
      setY(newy);
    end moveTo;
  procedure rMoveTo(deltax, deltay); integer deltax; integer deltay;
    begin
      setX(deltax + getX);
      setY(deltay + getY);
    end moveTo;
end Shape;
```

Shape in Simula (2/5)

```
Shape class Rectangle(width, height);
  integer width; integer height;
begin
  comment -- get the width & height of the object --;
  integer procedure getWidth;
    getWidth := width;
  integer procedure getHeight;
    getHeight := height;
  comment -- set the width & height of the object --;
  integer procedure setWidth(newwidth); integer newwidth;
    width := newwidth;
  integer procedure setHeight(newheight); integer newheight;
    height := newheight;
  comment -- draw the rectangle --;
  procedure draw;
    begin
      Outtext("Drawing a Rectangle at:");
      Outint(getX, 0); Outtext(","); Outint(getY, 0);
      Outtext(", width"); Outint(getWidth, 0);
      Outtext(", height"); Outint(getHeight, 0);
      Outimage;
    end draw;
end Rectangle;
```

Shape in Simula (3/5)

```
Shape class Circle(radius); integer radius;
begin
  comment -- get the radius of the object --;
  integer procedure getRadius;
    getRadius := radius;

  comment -- set the radius of the object --;
  integer procedure setRadius(newradius); integer newradius;
    radius := newradius;

  comment -- draw the circle --;
  procedure draw;
    begin
      Outtext("Drawing a Circle at:");
      Outint(getX, 0);
      Outtext(",");
      Outint(getY, 0);
      Outtext("), radius");
      Outint(getRadius, 0);
      Outimage;
    end draw;
end Circle;
```

Shape in Simula (4/5)

```
comment -- declare the variables used --;
ref(Shape) array scribble(1:2);
ref(Rectangle) arectangle;
integer i;

comment -- populate the array with various shape instances --;
scribble(1) :- new Rectangle(10, 20, 5, 6);
scribble(2) :- new Circle(15, 25, 8);

comment -- iterate on the list, handle shapes polymorphically --;
for i := 1 step 1 until 2 do
  begin
    scribble(i).draw;
    scribble(i).rMoveTo(100, 100);
    scribble(i).draw;
  end;

comment -- call a rectangle specific instance --;
arectangle :- new Rectangle(0, 0, 15, 15);
arectangle.draw;
arectangle.setWidth(30);
arectangle.draw;
```

Shape in Simula – Execution (5/5)

```
> cim shape.sim
Compiling shape.sim:
gcc -g -O2 -c shape.c
gcc -g -O2 -o shape shape.o -L/usr/local/lib -lcim
> ./shape
Drawing a Rectangle at:(10,20), width 5, height 6
Drawing a Rectangle at:(110,120), width 5, height 6
Drawing a Circle at:(15,25), radius 8
Drawing a Circle at:(115,125), radius 8
Drawing a Rectangle at:(0,0), width 15, height 15
Drawing a Rectangle at:(0,0), width 30, height 15
```

Impact of Simula 67

All the object-oriented languages inherit from Simula.

Smalltalk further with object orientation,
further with dynamic binding.

Impact of Simula 67

All the object-oriented languages inherit from Simula.

Smalltalk further with object orientation,
further with dynamic binding.

Objective-C, Pascal, C++, etc.
further with messages.

Impact of Simula 67

All the object-oriented languages inherit from Simula.

Smalltalk further with object orientation,
further with dynamic binding.

Objective-C, Pascal, C++, etc.
further with messages.

CLOS further with method selections.

Impact of Simula 67

All the object-oriented languages inherit from Simula.

Smalltalk further with object orientation,
further with dynamic binding.

Objective-C, Pascal, C++, etc.
further with messages.

CLOS further with method selections.

Eiffel further with software engineering,
further with inheritance.

Impact of Simula 67

All the object-oriented languages inherit from Simula.

Smalltalk further with object orientation,
further with dynamic binding.

Objective-C, Pascal, C++, etc.
further with messages.

CLOS further with method selections.

Eiffel further with software engineering,
further with inheritance.

C++ further with static typing and static binding,
deeper in the *.

Impact of Simula 67

All the object-oriented languages inherit from Simula.

Smalltalk further with object orientation,
further with dynamic binding.

Objective-C, Pascal, C++, etc.
further with messages.

CLOS further with method selections.

Eiffel further with software engineering,
further with inheritance.

C++ further with static typing and static binding,
deeper in the *.

Hybrid languages logic, functional, assembly, stack based etc.

Table of Contents

- 1 Simula
- 2 Smalltalk
- 3 The mysterious language

Smalltalk

We called Smalltalk Smalltalk so that nobody would expect anything from it.

– Alan Kay

Principles:

- Everything is object;
- Every object is described by its class (structure, behavior);
- Message passing is the only interface to objects.

Origin:

- A programming language that children can understand;
- To create “tomorrow’s computer”: Dynabook.

Table of Contents

1 Simula

2 Smalltalk

- The People Behind Smalltalk
 - Smalltalk 72
 - Smalltalk 76
 - Smalltalk 80

3 The mysterious language

Alan Kay



Quote

I invented the term Object-Oriented and I can tell you I did not have C++ in mind.
– A. Kay

Alan Kay, 1984



Alan Kay



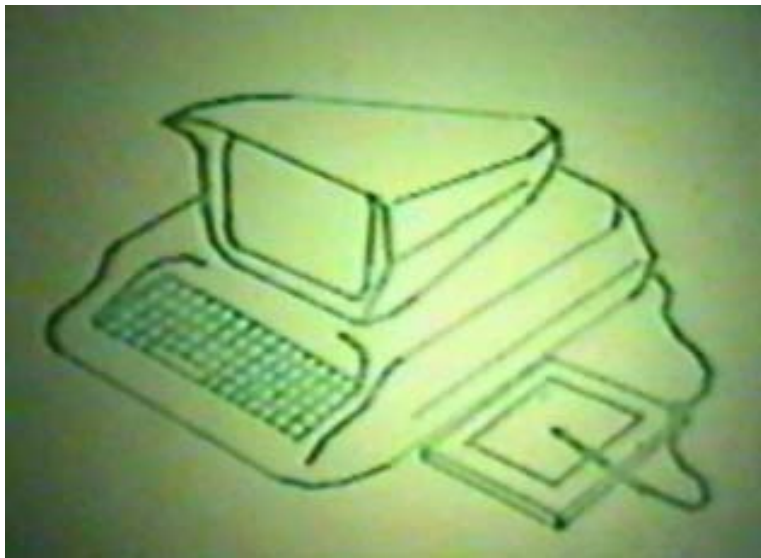
Ivan Sutherland's Sketchpad 1967



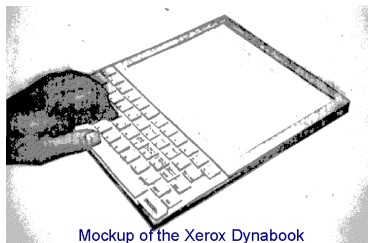
Douglas Engelbart's NLS 1974



Flex Machine 1967



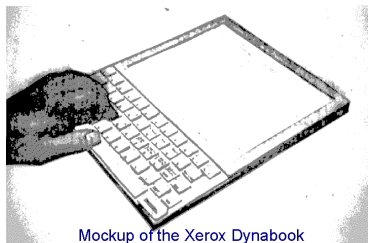
DynaBook



Mockup of the Xerox Dynabook

It would have, "enough power to outrace your senses of sight and hearing, enough capacity to store for later retrieval thousands of page-equivalents of reference material, poems, letter, recipes, records, drawings, animations, musical scores, waveforms, dynamic simulations, and anything else you would like to remember and change..."

DynaBook



Mockup of the Xerox Dynabook

It would have, "enough power to outrace your senses of sight and hearing, enough capacity to store for later retrieval thousands of page-equivalents of reference material, poems, letter, recipes, records, drawings, animations, musical scores, waveforms, dynamic simulations, and anything else you would like to remember and change..."

To put this project in context, the smallest general purpose computer in the early 1970s was about the size of a desk and the word "multimedia" meant a slide-tape presentation.

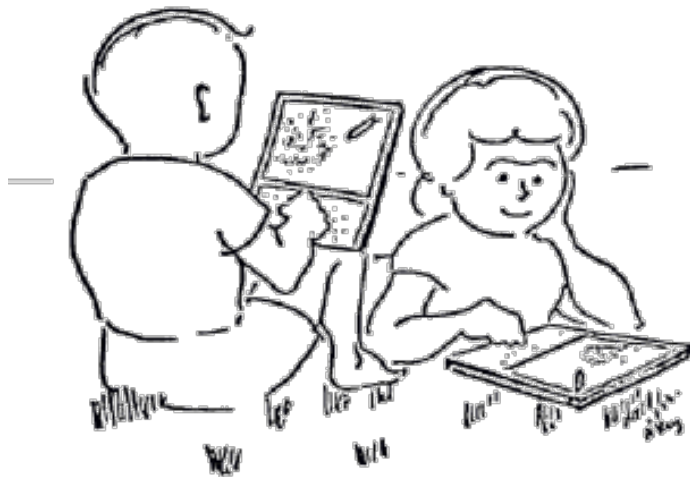


Table of Contents

1 Simula

2 Smalltalk

- The People Behind Smalltalk
- **Smalltalk 72**
- Smalltalk 76
- Smalltalk 80

3 The mysterious language

Smalltalk 72

- Written in BASIC.
- Reuses the classes and instances from Simula 67.
- Adds the concept of “message”.
Dynamic method lookup.

Smalltalk 72 Sample

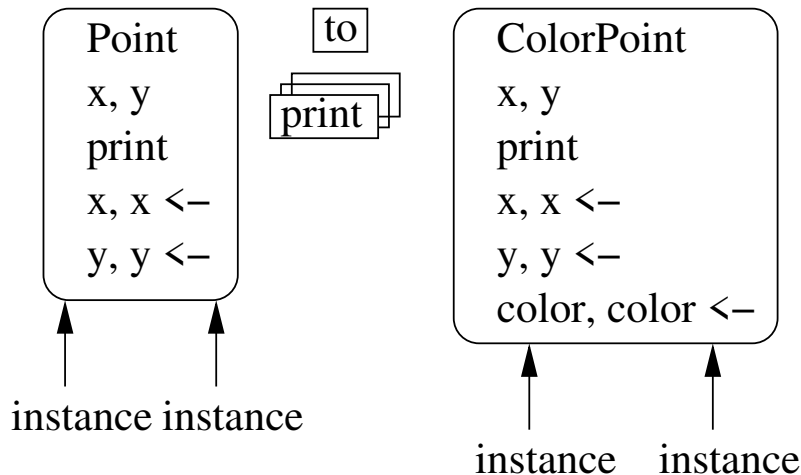
```
to Point
| x y
(
  isNew => ("x <- :.
            "y <- :.)

  <) x
=> ( <) <- => ("x <- : )
      ^ x)

  <) y
=> ( <) <- => ("y <- : )
      ^ y)

  <) print => ("( print.
              x print.   center <- Point 0
              ", print. => (0,0)
              y print.   center x <- 3
              ") print.) => (3,0)
```

Classes and Instances in Smalltalk 72



Smalltalk 72 Criticisms

- to is a primitive, not a method.
- A class is not an object.
- The programmer implements the method lookup.
- Method lookup is too slow.
- No inheritance.

⇒ Programmers were using global procedures.

But some successes:

- Pygmalion
“Programming by examples”
inspired Star.

Table of Contents

1 Simula

2 Smalltalk

- The People Behind Smalltalk
- Smalltalk 72
- **Smalltalk 76**
- Smalltalk 80

3 The mysterious language

Smalltalk 76

- Introduction of the `Class` class.
The class of classes. Instance of itself. *Metaclass*. How to print a class, add method, instantiate etc.

Smalltalk 76

- Introduction of the `Class` class.
The class of classes. Instance of itself. *Metaclass*. How to print a class, add method, instantiate etc.
- Introduction of the `Object` class.
Default behavior, shared between all the objects.

Smalltalk 76

- Introduction of the `Class` class.
The class of classes. Instance of itself. *Metaclass*. How to print a class, add method, instantiate etc.
- Introduction of the `Object` class.
Default behavior, shared between all the objects.
- Introduction of dictionaries.
Message handling is no longer handled by the programmers.

Smalltalk 76

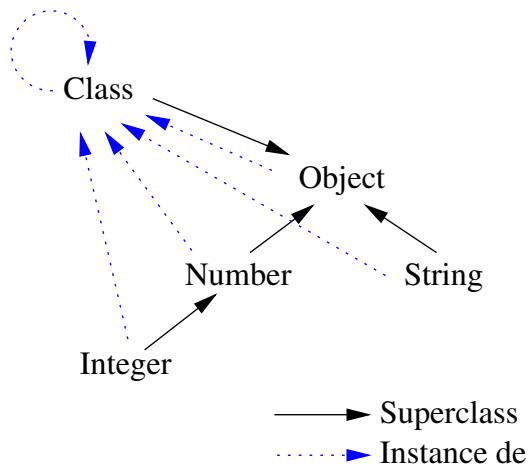
- Introduction of the `Class` class.
The class of classes. Instance of itself. *Metaclass*. How to print a class, add method, instantiate etc.
- Introduction of the `Object` class.
Default behavior, shared between all the objects.
- Introduction of dictionaries.
Message handling is no longer handled by the programmers.
- Introduction of inheritance.

Smalltalk 76

- Introduction of the Class class.
The class of classes. Instance of itself. *Metaclass*. How to print a class, add method, instantiate etc.
- Introduction of the Object class.
Default behavior, shared between all the objects.
- Introduction of dictionaries.
Message handling is no longer handled by the programmers.
- Introduction of inheritance.
- Removal of the to primitive.
Replaced by the new message sent to Class:

```
Class new title: 'Rectangle';  
      fields: 'origin┐corner'.
```

Instantiation, inheritance in Smalltalk 76



- Objects keep a link with their generator: `is-instance-of`

Smalltalk 76 Criticism

- Significant improvement:
 - ▶ Byte-code and a virtual machine provide a 4-100 speedup.
 - ▶ *ThingLab*, constraint system experimentation.
 - ▶ *PIE, Personal Information Environment*.

Smalltalk 76 Criticism

- Significant improvement:
 - ▶ Byte-code and a virtual machine provide a 4-100 speedup.
 - ▶ *ThingLab*, constraint system experimentation.
 - ▶ *PIE, Personal Information Environment*.
- But:
 - ▶ A single metaclass
hence a single behavior for classes
(no specific constructors, etc.).

Table of Contents

1 Simula

2 Smalltalk

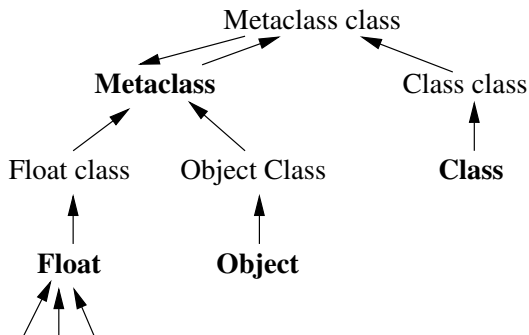
- The People Behind Smalltalk
- Smalltalk 72
- Smalltalk 76
- **Smalltalk 80**

3 The mysterious language

Smalltalk 80

- Deep impact over computer science of the 80's.
- Most constructors take part (Apple, Apollo, DEC, HP, Tektronix...).
- Generalization of the metaclass concept.

Is-instance-of in Smalltalk 80



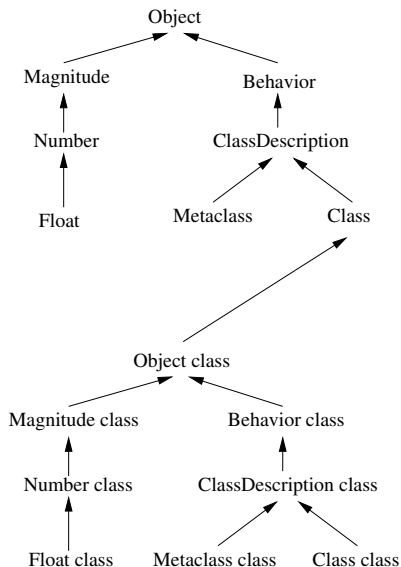
Three layer model:

Metaclass. Class behavior (instantiation, initialization, etc.).

Class. Type and behavior of objects.

Instances. The objects.

Inheritance in Smalltalk 80



The Smalltalk 80 System

More than a language, a system where *everything* is an object, and the only control structure is message passing.

- a *virtual image*;

The Smalltalk 80 System

More than a language, a system where *everything* is an object, and the only control structure is message passing.

- a *virtual image*;
- a byte-code compiler;

The Smalltalk 80 System

More than a language, a system where *everything* is an object, and the only control structure is message passing.

- a *virtual image*;
- a byte-code compiler;
- a virtual machine;

The Smalltalk 80 System

More than a language, a system where *everything* is an object, and the only control structure is message passing.

- a *virtual image*;
- a byte-code compiler;
- a virtual machine;
- more than 500 classes, 4000 methods, 15000 objects.

Smalltalk 80 Standard Library

- System
Class, Object, Number, Boolean, BlockContext etc.
- Programming Environment
Model, View, Controller, etc.
- Standard Library
Collection, Stream, etc.
- Notable inventions
Bitmap, Mouse, Semaphore, Process, ProcessScheduler

Smalltalk 80

The screenshot shows a Smalltalk 80 IDE window titled "Collection->select: [ANSI protocols-collection, enumerating]". The interface includes a menu bar (File, Edit, Workspace, Class, Method, Tools, Help), a toolbar, and a workspace area. On the left, a "System Transcript" window is open, showing a list of messages. The main workspace is divided into three panes: "Instance" (showing "All"), "Class" (showing "select"), and "Method source". The "Method source" pane displays the following Smalltalk code:

```
select: discriminator
  "Evaluate the <monadicValuable> argument, discriminator, for each of the receiver's elements.
  Answer a new <collection> like the receiver containing only those elements for which
  the discriminator evaluates to true."

  | newCollection |
  newCollection := self species new.
  self do: [each | (discriminator value: each) ifTrue: [newCollection add: each]].
  ^newCollection
```

Below the workspace, a terminal window shows a list of commands and a message: "No more new messages in folder". The terminal output includes:

```
GC's G... Help FldrList PrevMsg PrevPage Delete Reply
[2 Comm] OTHER CMDS [UviewMsg] NextMsg Spc NextPage U Undelete Forward
```

The terminal also displays a paragraph of text:

GameCenter has posted a user guide (tutorial) that talks about the technology. Compared to other cards, this one also includes reviews of the 3 cards on the market already. I'm still waiting for the DDR cards. Those are the only ones worth upgrading to over my TNT2 in my opinion. Thanks VE.

Below the terminal, there is a list of items:

- 3dfx Napalm Really Expensive?

The bottom of the screen shows a taskbar with several open windows: Start, Untitled, System, Packag, Collec..., Untitled, [S T O], Telnet, Shugs, Voodoo, HTTP5, and a system clock showing 1:13 PM.

Smalltalk 80

The screenshot displays the Smalltalk 80 environment with several windows:

- Системная информация** (System Information): A red header window.
- Справка о...** (Help): A green window with a list of topics including "начало работы", "клавиши, мышь, курсор", "окна, панели, меню", "редактор текста", "понятия языка", "системное меню", "иерархия классов", "программирование", "учебные примеры", "графический редактор", and "словарь терминов".
- Системный Блокнот** (System Notepad): A white window containing the text: "В этом окне хранятся общепользовательские выражения Смолтока" and "просмотр словаря системы".
- Просмотр иерархии классов** (Class Hierarchy Viewer): A blue window showing a tree structure. The root is "СправкаОСистеме", with children "Суфлер" and "Точка". "Точка" has children "УправлениеКурсором...", "Файл", "ЧтениеКласса", and "Шрифт". A sub-window shows details for "Точка": "конеч: экзТочка", "Видает прямоугольник с началом, равным приемнику, равным аргументу", and "Прямоугольник начало: сам конеч: экзТочка".
- Рабочее окно** (Workspace): A yellow window with a mouse cursor.
- Сортировка по...** (Sort by...): A purple window showing a list of files:

File Name	Size	Creation Date	Modification Date
ansi.sys			
aropur.exe			
append.exe	7682	96-08-08	23
arabic.com	24089	96-08-08	23
aspistub.sys	386	96-08-09	01
acsign.com	2684	96-08-09	08
...

Booleans: Logical Operators

Boolean methods: `and:`, `or:`, `not:`.

- In the `True` class

```
and: aBlock  
"Evaluate aBlock"  
↑ aBlock value
```

- In the `False` class

```
and: aBlock  
"Return receiver"  
↑ self
```

Booleans: Control Structures

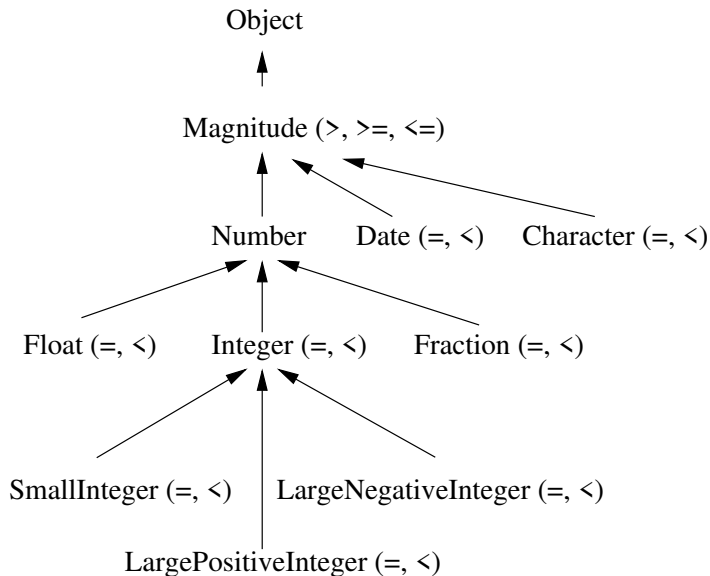
More Boolean methods:

- `ifTrue:`
- `ifFalse:`
- `ifTrue:ifFalse:`
- ... `ifFalse:ifTrue:`

For instance, compute a minimum:

```
| a b x |  
...  
a <= b ifTrue: [ x <- a ]  
        ifFalse: [ x <- b ].  
...
```

Integers in Smalltalk 80



Integers in Smalltalk 80

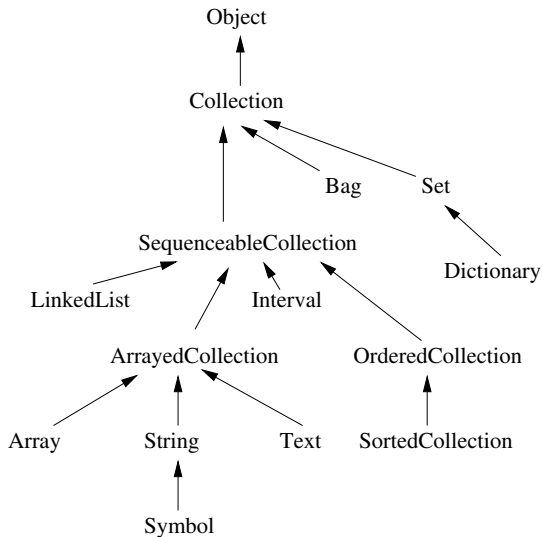
In Magnitude

```
>= aMagnitude  
  ↑ (self < aMagnitude) not
```

In Date

```
< aDate  
  year < aDate year  
    ifTrue: [↑ day < aDate day]  
    ifFalse: [↑ year < aDate year]
```

Collections in Smalltalk 80



Collections in Smalltalk 80

In LinkedList:

```
do: aBlock  
  | aLink |  
  aLink <- firstLink.  
  [aLink = nil] whileFalse:  
    [aBlock value: aLink.  
     aLink <- aLink nextLink]
```


Using Smalltalk 80 Collections

```
sum <- 0.  
#(2 3 5 7 11) do:  
  [ :prime |  
    sum <- sum + (prime * prime) ]
```

or:

```
sum <- 0.  
#(2 3 5 7 11)  
  collect: [ :prime | prime * prime ];  
  do: [ :number | sum <- sum + number ]
```

The Smalltalk 80 Environment

- Everything is sorted, classified, so that the programmers can browse the system.
- Everything is object.
- The system is reflexive.
- The *inspector* to examine an object.
- Coupled to the debugger and the interpreter, a wonderful programming environment.
- Big success of Smalltalk in prototyping.

Sub-classing in Smalltalk 80: Complexes

- Chose a superclass: Number.
- Browse onto it (look in the Numeric-Numbers *category*). A skeleton is proposed.

```
Number subclass: #Complex
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Numeric-Numbers'
```

- Complete.

```
Number subclass: #Complex
  instanceVariableNames: 're_im'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Numeric-Numbers'
```

Sub-classing in Smalltalk 80: Complexes

- Validate.
- Go into the Complex class, class methods, and create:

```
re: realPart im: imPart
  ↑ (self new) setRe: realPart setIm: imPart
```

Sub-classing in Smalltalk 80: Complexes

Instance methods:

```
setRe: realPart setIm: imPart
  re <- realPart.
  im <- imPart
```

- im

"Return the imaginary part of the receiver

↑ im

- + aComplex

```
↑ Complex re: (re + aComplex re)
           im: (im + aComplex im)
```

But then:

```
(Complex re: 42 im: 51) + 666
```

yields message not understood: re.

Sub-classing in Smalltalk 80: Complexes

First solution: implement asComplex in Number and Complex

```
"Class Number: addition."
```

```
+ aNumber
```

```
| c |
```

```
c <- aNumber asComplex.
```

```
↑ Complex re: (re + c re) im: (im + c im)
```

Second solution: implement re and im in Number.

But these don't address:

```
666 + (Complex re: 42 im: 51)
```

This issue was known by Smalltalk designers who faced it for other Number subclasses; they introduced the generality class method.

Smalltalk 80 Criticism

- Some loopholes in the semantics.
- The metaclass concept was considered too difficult.
- No typing!
- Dynamic dispatch exclusively, that's slow.
- The GC is nice, but slow too.
- The virtual image prevents collaborative development.
- No security (one can change *anything*).
- No means to produce standalone applications.
- No multiple inheritance.

Demo with Squeak

<https://squeak.org>

1 Simula

2 Smalltalk

3 C++

- The Man Behind C++
- C++

The Man Behind C++

1 Simula

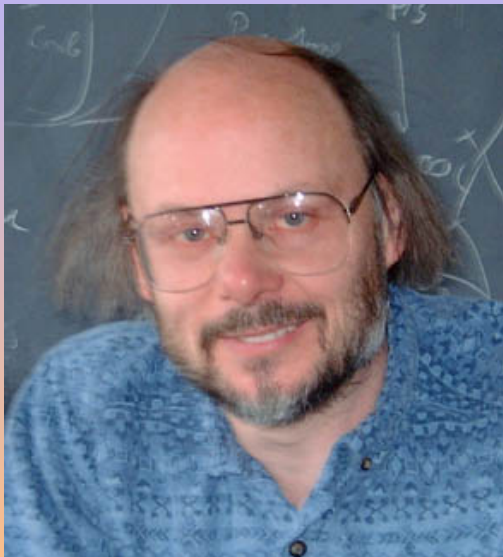
2 Smalltalk

3 C++

- The Man Behind C++
- C++



Bjarne Stroustrup



Bjarne Stroustrup



Young Bjarne Stroustrup



Bjarne Stroustrup 2008



- 1 Simula
- 2 Smalltalk
- 3 C++
 - The Man Behind C++
 - C++

- Bjarne Stroustrup, BellLabs, 1982.
- cfront, a C preprocessor.
- G++, the first real C++ compiler.
- Standardized in 1998.

C++: A better & safer C

- introduction of `const`;
- introduction of reference;
- introduction of prototypes;
- introduction of Booleans;
- declaring variable anywhere;
- introduction of `void`;
- introduction of `inline`;
- introduction of namespace;
- introduction of overloading etc.

Most features made it into more modern Cs.

C++: A better & safer C

- introduction of `const`;
- introduction of reference;
- introduction of prototypes;
- introduction of Booleans;
- declaring variable anywhere;
- introduction of `void`;
- introduction of `inline`;
- introduction of namespace;
- introduction of overloading etc.

Most features made it into more modern Cs.

C++: A better & safer C

- introduction of `const`;
- introduction of reference;
- introduction of prototypes;
- introduction of Booleans;
- declaring variable anywhere;
- introduction of `void`;
- introduction of `inline`;
- introduction of namespace;
- introduction of overloading etc.

Most features made it into more modern Cs.

C++: A better & safer C

- introduction of `const`;
- introduction of reference;
- introduction of prototypes;
- introduction of Booleans;
- declaring variable anywhere;
- introduction of `void`;
- introduction of `inline`;
- introduction of namespace;
- introduction of overloading etc.

Most features made it into more modern Cs.

C++: A better & safer C

- introduction of `const`;
- introduction of reference;
- introduction of prototypes;
- introduction of Booleans;
- declaring variable anywhere;
- introduction of `void`;
- introduction of `inline`;
- introduction of `namespace`;
- introduction of overloading etc.

Most features made it into more modern Cs.

C++: A better & safer C

- introduction of `const`;
- introduction of reference;
- introduction of prototypes;
- introduction of Booleans;
- declaring variable anywhere;
- introduction of `void`;
- introduction of `inline`;
- introduction of `namespace`;
- introduction of overloading etc.

Most features made it into more modern Cs.

C++: A better & safer C

- introduction of `const`;
- introduction of reference;
- introduction of prototypes;
- introduction of Booleans;
- declaring variable anywhere;
- introduction of `void`;
- introduction of `inline`;
- introduction of `namespace`;
- introduction of overloading etc.

Most features made it into more modern Cs.

C++: A better & safer C

- introduction of `const`;
- introduction of reference;
- introduction of prototypes;
- introduction of Booleans;
- declaring variable anywhere;
- introduction of `void`;
- introduction of `inline`;
- introduction of namespace;
- introduction of overloading etc.

Most features made it into more modern Cs.

C++: A better & safer C

- introduction of `const`;
- introduction of reference;
- introduction of prototypes;
- introduction of Booleans;
- declaring variable anywhere;
- introduction of `void`;
- introduction of `inline`;
- introduction of namespace;
- introduction of overloading etc.

Most features made it into more modern Cs.

C++: A better & safer C

- introduction of `const`;
- introduction of reference;
- introduction of prototypes;
- introduction of Booleans;
- declaring variable anywhere;
- introduction of `void`;
- introduction of `inline`;
- introduction of namespace;
- introduction of overloading etc.

Most features made it into more modern Cs.

Class declaration

```
#ifndef SHAPE_HH_
# define SHAPE_HH_ 1

class Shape
{
public:
    Shape(int x, int y);

    void x_set(int x); int x_get() const;
    void y_set(int y); int y_get() const;

    void move_to(int x, int y);
    void rmove_to(int deltax, int deltay);

    virtual void draw() const;
private:
    int x_, y_;
};

#endif SHAPE_HH_
```

Class implementation

```
#include "shape.hh"

// Constructor.
Shape::Shape(int x, int y) : x_(x), y_(y) {}

// Accessors for x & y.
int Shape::x_get() const { return x_; }
int Shape::y_get() const { return y_; }
void Shape::x_set(int x) { x_ = x; }
void Shape::y_set(int y) { y_ = y; }

// Move the shape.
void Shape::move_to(int x, int y) { x_set(x); y_set(y); }
void Shape::rmove_to(int x, int y) {
    move_to(x_get() + x, y_get() + y);
}

// Abstract draw method.
void Shape::draw() const { abort(); }
```

Class definition

```
#pragma once

class Shape
{
public:
    Shape(int x, int y) : x_(x), y_(y) {}

    int x_get() const { return x_; }
    int y_get() const { return y_; }

    void x_set(int x) { x_ = x; }
    void y_set(int y) { y_ = y; }

    void move_to(int x, int y) { x_ = x; y_ = y; }
    void rmove_to(int x, int y) { x_ += x; y_ += y; }
    virtual void draw() const = 0;
private:
    int x_, y_;
};
```

Sub-classing: rectangle.hh

```
#pragma once

#include <iostream>
#include "shape.hh"
class Rectangle: public Shape {
public:
    Rectangle(int x, int y, int w, int h) :
        Shape(x, y), width_(w), height_(h) {}
    int width_get() const { return width_; }
    int height_get() const { return height_; }
    int width_set(int w) { width_ = w; }
    int height_set(int h) { height_ = h; }
    void draw() const {
        std::cout << "Drawing a Rectangle at: (" << x_get()
            << "," << y_get() << "), " << "width " << width_
            << ", height " << height_ << std::endl;
    }
private:
    int width_, height_;
};
```

Sub-classing: circle.hh

```
#pragma once

#include <iostream>
#include "shape.hh"
class Circle: public Shape {
public:
    Circle(int x, int y, int r) :
        Shape(x, y), radius_(r) {}

    int radius_get() const { return radius_; }
    void radius_set(int r) { radius_ = r; }

    void draw() const {
        cout << "Drawing a Circle at: ("
            << x_get() << "," << y_get() << "), "
            << "radius " << radius_ << endl;
    }
private:
    int radius_;
};
```


Polymorphism

```
#include "shape.hh"
#include "circle.hh"
#include "rectangle.hh"

int
main()
{
    Shape *shapes[2];
    shapes[0] = new Rectangle(10, 20, 5, 6);
    shapes[1] = new Circle(15, 25, 8);

    for (int i = 0; i < 2; i++)
    {
        shapes[i]->draw();
        shapes[i]->remove_to(100, 100);
        shapes[i]->draw();
    }
}
```

Result:

Drawing a Rectangle at: (10,20), width 5, height 6

Drawing a Rectangle at: (110,120), width 5, height 6

Drawing a Circle at: (15,25), radius 8

Drawing a Circle at: (115,125), radius 8

Parameterized Polymorphism

```
template <typename T>
T
id(T t)
{
    return t;
}

int
main()
{
    id(3);
    id(3.0);
    id("three");
    int three[3] = { 3, 3, 3 };
    id(three);
    id(main);
}
```

Generic Classes

```
#include <iostream>
template <typename T> struct Pair {
    Pair(T fst, T snd): fst_(fst), snd_(snd) {}
    T fst() const { return fst_; }
    T snd() const { return snd_; }
private:
    T fst_, snd_;
};

int main() {
    Pair<int> foo(2, 3);
    std::cout << foo.fst() << ", " << foo.snd() << std::endl;

    Pair<float> bar(2.2, 3.3);
    std::cout << bar.fst() << ", " << bar.snd() << std::endl;

    Pair <Pair<int> > baz(Pair<int>(1, 2), Pair<int>(3, 4));
    std::cout << baz.fst().fst() << baz.fst().snd()
              << baz.snd().fst() << baz.snd().snd()
              << std::endl;
}
```

```
#include <iostream>
#include <iterator>
#include <list>

int
main()
{
    std::list<int> list;
    list.push_back(1);
    list.push_back(2);
    list.push_back(3);
    std::copy(list.begin(), list.end(),
              std::ostream_iterator <int>(std::cout, "\n"));
}
```

Quickly Read Only

```
template <class _Tp, class _Alloc, size_t __bufsize>
template <class _ForwardIterator>
void
deque<_Tp, _Alloc, __bufsize>::
    insert(iterator __pos, _ForwardIterator __first,
           _ForwardIterator __last, forward_iterator_tag) {
size_type __n = 0;
distance(__first, __last, __n);
if (__pos._M_cur == _M_start._M_cur) {
    iterator __new_start = _M_reserve_elements_at_front(__n);
    __STL_TRY {
        uninitialized_copy(__first, __last, __new_start);
        _M_start = __new_start;
    }
    __STL_UNWIND(_M_destroy_nodes(__new_start._M_node,
                                   _M_start._M_node));
}
else if (__pos._M_cur == _M_finish._M_cur) {
    iterator __new_finish = _M_reserve_elements_at_back(__n);
    __STL_TRY {
        uninitialized_copy(__first, __last, _M_finish);
        _M_finish = __new_finish;
    }
    __STL_UNWIND(_M_destroy_nodes(_M_finish._M_node + 1,
                                   __new_finish._M_node + 1));
}
```

Poor Error Messages

```
#include <iostream>
#include <list>

int
main()
{
    std::list<int> list;
    list.push_back(1);
    list.push_back(2);
    list.push_back(3);
    const std::list<int> list2 = list;

    for (std::list<int>::iterator i = list2.begin();
         i != list2.end(); ++i)
        std::cout << *i << std::endl;
}
```

Poor Error Messages

G++ 2.95:

```
bar.cc: In function 'int main()':
```

```
bar.cc:13: conversion from
```

```
  '_List_iterator<int,const int &, const int *>'
```

```
to non-scalar type
```

```
  '_List_iterator<int,int &, int *>' requested
```

```
bar.cc:14: no match for
```

```
  '_List_iterator<int,int &,int *> & !=
```

```
  _List_iterator<int,const int &,const int *>'
```

```
/usr/lib/gcc-lib/i386-linux/2.95.4/../../../../include/g++-3/stl_list.h:70:
```

```
candidates are:
```

```
bool _List_iterator<int,int &,int *>::operator !=
```

```
  (const _List_iterator<int,int &,int *> &) const
```


(A Bit Less) Poor Error Messages

Some progress: G++ 3.3.

```
list-invalid.cc: In function 'int main()':  
list-invalid.cc:13: error: conversion from  
  'std::_List_iterator<int, const int&, const int*>'  
to non-scalar type  
  'std::_List_iterator<int, int&, int*>' requested
```

G++ 3.4, 4.0 and 4.1, 4.2, 4.3 and 4.4.

```
list-invalid.cc: In function 'int main()':  
list-invalid.cc:13: error: conversion from  
  'std::_List_const_iterator<int>' to non-scalar type  
  'std::_List_iterator<int>' requested
```

G++ 4.5.

```
list-invalid.cc: In function 'int main()':  
list-invalid.cc:13:50: error: conversion from  
  'std::list<int>::const_iterator' to non-scalar type  
  'std::list<int>::iterator' requested
```

(A Bit Less) Poor Error Messages

Some progress: G++ 3.3.

```
list-invalid.cc: In function 'int main()':  
list-invalid.cc:13: error: conversion from  
  'std::_List_iterator<int, const int&, const int*>'  
  to non-scalar type  
  'std::_List_iterator<int, int&, int*>' requested
```

G++ 3.4, 4.0 and 4.1, 4.2, 4.3 and 4.4.

```
list-invalid.cc: In function 'int main()':  
list-invalid.cc:13: error: conversion from  
  'std::_List_const_iterator<int>' to non-scalar type  
  'std::_List_iterator<int>' requested
```

G++ 4.5.

```
list-invalid.cc: In function 'int main()':  
list-invalid.cc:13:50: error: conversion from  
  'std::list<int>::const_iterator' to non-scalar type  
  'std::list<int>::iterator' requested
```

(A Bit Less) Poor Error Messages

Some progress: G++ 3.3.

```
list-invalid.cc: In function 'int main()':  
list-invalid.cc:13: error: conversion from  
  'std::_List_iterator<int, const int&, const int*>'  
  to non-scalar type  
  'std::_List_iterator<int, int&, int*>' requested
```

G++ 3.4, 4.0 and 4.1, 4.2, 4.3 and 4.4.

```
list-invalid.cc: In function 'int main()':  
list-invalid.cc:13: error: conversion from  
  'std::_List_const_iterator<int>' to non-scalar type  
  'std::_List_iterator<int>' requested
```

G++ 4.5.

```
list-invalid.cc: In function 'int main()':  
list-invalid.cc:13:50: error: conversion from  
  'std::list<int>::const_iterator' to non-scalar type  
  'std::list<int>::iterator' requested
```

(A Bit Less) Poor Error Messages

G++ 4.6 and 4.7.

```
list-invalid.cc: In function 'int main()':
list-invalid.cc:13:50: erreur: conversion from
  'std::list<int>::const_iterator {aka std::_List_const_iterator<int>}'
  to non-scalar type
  'std::list<int>::iterator {aka std::_List_iterator<int>}' requested
```

G++ 4.8.

```
list-invalid.cc: In function 'int main()':
list-invalid.cc:13:50: error: conversion from
  'std::list<int>::const_iterator {aka std::_List_const_iterator<int>}'
  to non-scalar type
  'std::list<int>::iterator {aka std::_List_iterator<int>}' requested
  for (std::list<int>::iterator i = list2.begin ();
```

(A Bit Less) Poor Error Messages

G++ 4.6 and 4.7.

```
list-invalid.cc: In function 'int main()':
list-invalid.cc:13:50: erreur: conversion from
    'std::list<int>::const_iterator {aka std::_List_const_iterator<int>}'
    to non-scalar type
    'std::list<int>::iterator {aka std::_List_iterator<int>}' requested
```

G++ 4.8.

```
list-invalid.cc: In function 'int main()':
list-invalid.cc:13:50: error: conversion from
    'std::list<int>::const_iterator {aka std::_List_const_iterator<int>}'
    to non-scalar type
    'std::list<int>::iterator {aka std::_List_iterator<int>}' requested
    for (std::list<int>::iterator i = list2.begin ();
        ^
```

(A Bit Less) Poor Error Messages

ICC 8.1 and 9.1.

```
list-invalid.cc(8):
```

```
    remark #383: value copied to temporary, reference  
                to temporary used
```

```
list.push_back (1);
```

```
    ^
```

[...]

```
list-invalid.cc(13): error: no suitable user-defined conversion  
from
```

```
"std::list<int, std::allocator<int>>::const_iterator" to
```

```
"std::list<int, std::allocator<int>>::iterator" exists
```

```
for (std::list<int>::iterator i = list2.begin ();
```

```
    ^
```

ICC 10.0 and 11.0.

```
list-invalid.cc(13): error: no suitable user-defined conversion  
from "std::_List_const_iterator<int>"
```

```
to "std::_List_iterator<int>" exists
```

```
for (std::list<int>::iterator i = list2.begin ();
```

```
    ^
```

(A Bit Less) Poor Error Messages

ICC 8.1 and 9.1.

```
list-invalid.cc(8):
```

```
    remark #383: value copied to temporary, reference  
                to temporary used
```

```
list.push_back (1);
```

```
    ^
```

[...]

```
list-invalid.cc(13): error: no suitable user-defined conversion  
from
```

```
"std::list<int, std::allocator<int>>::const_iterator" to
```

```
"std::list<int, std::allocator<int>>::iterator" exists
```

```
for (std::list<int>::iterator i = list2.begin ();
```

```
    ^
```

ICC 10.0 and 11.0.

```
list-invalid.cc(13): error: no suitable user-defined conversion  
from "std::_List_const_iterator<int>"
```

```
to "std::_List_iterator<int>" exists
```

```
for (std::list<int>::iterator i = list2.begin ();
```

```
    ^
```



(A Bit Less) Poor Error Messages

Clang 1.1 (LLVM 2.7)

```
list-invalid.cc:13:33: error: no viable conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'std::list<int>::iterator' (aka '_List_iterator<int>')
for (std::list<int>::iterator i = list2.begin ();
      ^ ~~~~~
```

In file included from list-invalid.cc:2:

In file included from /usr/include/c++/4.2.1/list:69:

```
/usr/include/c++/4.2.1/bits/stl_list.h:113:12: note: candidate
      constructor (the implicit copy constructor) not viable:
      no known conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'struct std::_List_iterator<int> const' for 1st argument
struct _List_iterator
      ^
```

1 error generated.

(A Bit Less) Poor Error Messages

Clang 2.8 (LLVM 2.8).

```
list-invalid.cc:13:33: error: no viable conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'std::list<int>::iterator' (aka '_List_iterator<int>')
for (std::list<int>::iterator i = list2.begin ());
                        ^      ~~~~~
```

In file included from list-invalid.cc:2:
In file included from /usr/include/c++/4.2.1/list:69:
/usr/include/c++/4.2.1/bits/stl_list.h:112:12: note: candidate
constructor (the implicit copy constructor) not viable:
no known conversion from
'const_iterator' (aka '_List_const_iterator<int>') to
'std::_List_iterator<int> const &' for 1st argument
struct _List_iterator
^

1 error generated.

(A Bit Less) Poor Error Messages

Clang 2.9 (LLVM 2.9).

```
list-invalid.cc:13:33: error: no viable conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'std::list<int>::iterator' (aka '_List_iterator<int>')
for (std::list<int>::iterator i = list2.begin ());
                        ^      ~~~~~
```

In file included from list-invalid.cc:2:
In file included from /usr/include/c++/4.2.1/list:69:
/usr/include/c++/4.2.1/bits/stl_list.h:112:12: note: candidate
constructor (the implicit copy constructor) not viable:
no known conversion from
'const_iterator' (aka '_List_const_iterator<int>') to
'const std::_List_iterator<int> &' for 1st argument
struct _List_iterator
^

1 error generated.

(A Bit Less) Poor Error Messages

Clang 3.0 (LLVM 3.0) and Clang 3.1 (LLVM 3.1).

```
list-invalid.cc:13:33: error: no viable conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'std::list<int>::iterator' (aka '_List_iterator<int>')
for (std::list<int>::iterator i = list2.begin ();
      ^ ~~~~~
```

```
/usr/include/c++/4.2.1/bits/stl_list.h:112:12: note: candidate
      constructor (the implicit copy constructor) not viable:
      no known conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'const std::_List_iterator<int> &' for 1st argument;
struct _List_iterator
      ^
```

1 error generated.

4 Bad Engineering Properties of Object Oriented Languages



- Economy of execution.
How fast does a program run?
- Economy of compilation.
How long does it take to go from sources to executables?
- Economy of small-scale development.
How hard must an individual programmer work?
- Economy of large-scale development.
How hard must a team of programmers work?
- Economy of language features.
How hard is it to learn or use a programming language?

Economy of execution

Type information was first introduced in programming to improve code generation and run-time efficiency for numerical computations. In ML, accurate type information eliminates the need for nil-checking on pointer dereferencing.

Object-oriented style intrinsically less efficient than procedural style (`virtual`). The traditional solution to this problem (analyzing and compiling whole programs) violates modularity and is not applicable to libraries.

Much can be done to improve the efficiency of method invocation by clever program analysis, as well as by language features (e.g. `final`). Design type systems that can statically check many of the conditions that now require dynamic subclass checks.

Economy of compilation

Type information can be organized into interfaces for program modules (Modula-2, Ada...). Modules can then be compiled independently. Compilation of large systems is made more efficient. The messy aspects of system integration are thus eliminated.

Often, no distinction between the code and the interface of a class. Some object-oriented languages are not sufficiently modular and require recompilation of superclasses when compiling subclasses. Time spent in compilation may grow disproportionally with the size of the system.

We need to adopt languages and type systems that allow the separate compilation of (sub)classes, without resorting to recompilation of superclasses and without relying on “private” information in interfaces.

Economy of small-scale development

Well designed type systems allow typechecking to capture a large fraction of routine programming errors. Remaining errors are easier to debug: large classes of other errors have been ruled out. Typechecker as a development tool (changing the name of a type when its invariants change even though the type structure remains the same).

Big win of OO: class libraries and frameworks. But when ambition grows, programmers need to understand the details of those class libraries: more difficult than understanding module libraries. The type systems of most OOL are not expressive enough; programmers must often resort to dynamic checking or to unsafe features, damaging the robustness of their programs. Improvements in type systems for OOL will improve error detection and the expressiveness of interfaces.

Economy of large-scale development

Data abstraction and modularization have methodological advantages for development. Negotiate the interfaces, then proceed separately. Polymorphism is important for reusing code modularly. Teams developing/specializing class libraries. Reuse is a big win of OOL, but poor modularity wrt class extension and modification (method “removal”, etc.). Confusion bw classes and object types (limits abstractions). Subtype polymorphism is not good enough for containers. Formulating and enforcing inheritance interfaces: the contract bw a class and its subclasses. Requires language support development. Parametric polymorphism is beginning to appear but its interactions with OO features need to be better understood. Interfaces/subtyping and classes/subclassing must be separated.

Economy of language features

Well-designed orthogonal constructs can be naturally composed (array of arrays; n-ary functions vs 1-ary and tuples). Orthogonality reduces the complexity of languages. Learning curve thus reduced, re-learning minimized.

Smalltalk, good. C++ daunting in the complexity of its many features. Somewhere something went wrong; what started as economical and uniform (“everything is an object”) ended up as a baroque collection of class varieties. Java represents a healthy reaction, but is more complex than many people realize.

Prototype-based languages tried to reduce the complexity by providing simpler, more composable features, but much remains to be done for class-based languages. How can we design an OOL that allows powerful engineering but also simple and reliable engineering?

Table of Contents

- 1 Simula
- 2 Smalltalk
- 3 The mysterious language

Reading...



What is this Book ?

Table of Contents

- 1 Simula
- 2 Smalltalk
- 3 The mysterious language**
 - **People behind Eiffel**
 - Overview of the System
 - Overview of the Language

Bertrand Meyer (1950), MIT



Table of Contents

- 1 Simula
- 2 Smalltalk
- 3 The mysterious language
 - People behind Eiffel
 - Overview of the System
 - Overview of the Language

Introducing Eiffel

- High-level language designed for Software Engineering, portable, with an original and clear syntax
- Modern conception of multiple class inheritance
- High level tools and programmatic concepts (Virtual classes, Generics, Exceptions, etc.)
- Lot of standard libraries

Libraries

EiffelCOM (COM,OLE,ActiveX),
EiffelCORBA,
EiffelMath,
EiffelNet (client-serveur),
EiffelLex & *EiffelParse*,
EiffelStore (BD),
EiffelWEB,
Eiffel DLE (dynamic link),
EiffelVision (GUI),
Graphical Eiffel for Windows, *Eiffel WEL* (Windows),
EiffelThreads,
etc.

An Eiffel Application

An Eiffel Application is called a *system*.

- Classes :
 - ▶ One per file (.e)
 - ▶ Grouped in *clusters*
 - ▶ One one them is the main class
- Eiffel Librairies (only one in practice)
- External Librairies
- A file describing the application
 - ▶ LACE file, *Langage pour l'assemblage des classes en Eiffel*

Clusters

LOGICAL point-of-view

Set of classes building an autonomous part of the application

PHYSICAL point-of-view

All these classes lay in the same repository

LACE point-of-view

A cluster is a name associated to a repository

LACE File Example

```
system
  geo

root
  TEST(TEST): "main"

default
  precompiled("$EIFFEL3/precomp/spec/$PLATFORM/base")

cluster
  TEST:    "$EIFFELDIR/TEST" ;
  FIGS:    "$EIFFELDIR/FIGURES" ;

external
  object:  "$$(EIFFEL3)/library/lex/spec/$$(PLATFORM)/lib/lex.a"

end
```

Original Concepts

Adaptation clauses for inheritance

resolve multiple inheritance problems

Contract Programming

Promote reusability and modularity

Graphical User Interface

A full dedicated GUI: drag-and-drop, etc.

A smart compiler

Compiler with three modes *really usefull in developpement phases*

A smart compiler

Compiler with three modes *really usefull in developpement phases*

FINALIZING Optimisation and production of an executable file where all optimizations hgave been applied. May be very slow!

A smart compiler

Compiler with three modes *really usefull in developpement phases*

FINALIZING Optimisation and production of an executable file where all optimizations hgave been applied. May be very slow!

FREEZING compile and produce an executable file

A smart compiler

Compiler with three modes *really usefull in developpement phases*

FINALIZING Optimisation and production of an executable file where all optimizations hgave been applied. May be very slow!

FREEZING compile and produce an executable file

MELTING compilation by **patches**. Very fast, a modification only recompile what is necessary (not good performance, useful for developpement)

A full System

EiffelBench the visual workbench for object-oriented development

A full System

EiffelBench the visual workbench for object-oriented development

EiffelBuild the editor to build GUI

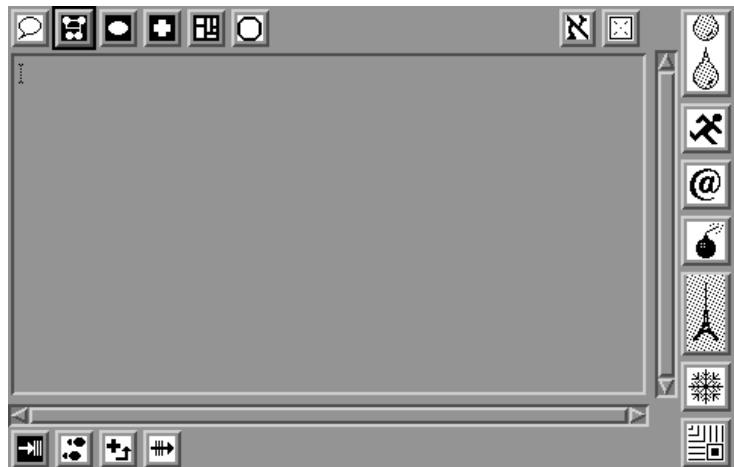
A full System

EiffelBench the visual workbench for object-oriented development

EiffelBuild the editor to build GUI

EiffelCase the tools dedicated to build and design application

Ebench



Edit a Class

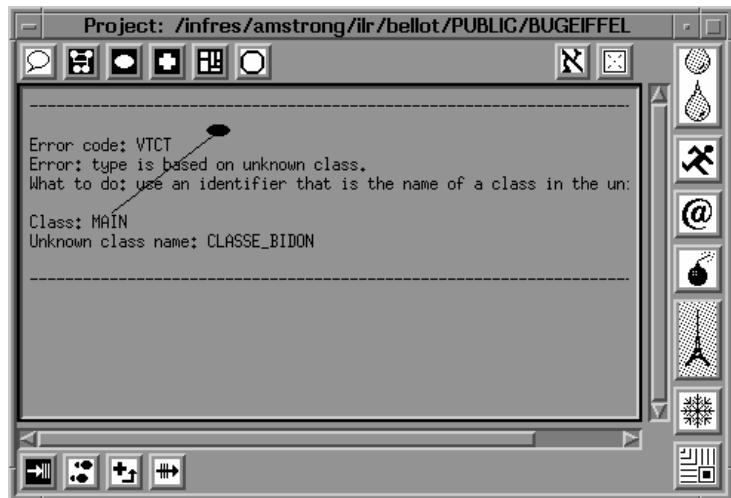


Table of Contents

- 1 Simula
- 2 Smalltalk
- 3 The mysterious language
 - People behind Eiffel
 - Overview of the System
 - Overview of the Language

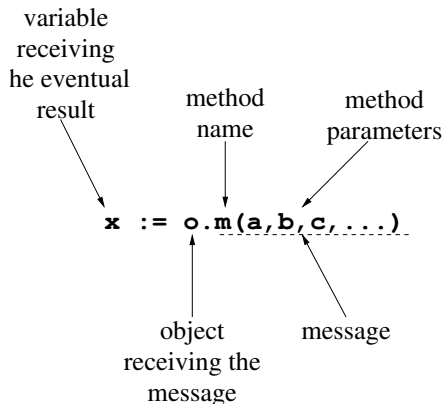
Example of a Class

```
class POINT
  -- un point dans un dessin géométrique

feature
  -- deux attributs : les coordonnées
  xc,yc : INTEGER ;

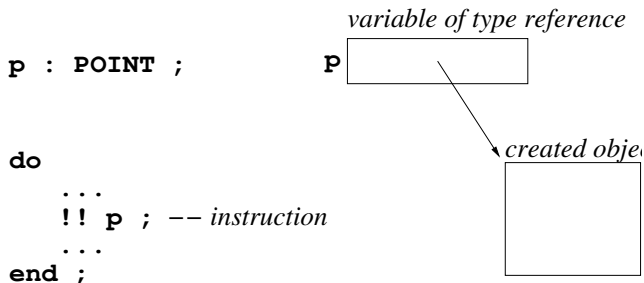
  -- une méthode : changer les coordonnées
  set_x_y(x,y : INTEGER) is
    do
      xc := x ;
      yc := y ;
    end ;
end -- class POINT
```

Methods Calls



- The object execute the method `m` which is executed in its own context.
- For **distributed** objects, a message is sent, **otherwise** it is a simple procedure call.

Creating an Object



Except for explicit declaration, all the object's variables are *references*: they handle pointers.

Creation with Initialization 1/2

```
class POINT
create
  make  -- init method
feature
  -- init method
  make(x,y : INTEGER) is
    do
      set_x_y(x,y) ;
    end ;

  -- same as previously
end -- class POINT
```

- Attributes are initialized with a default value (e.g., 0 for an Integer, Void for a variable with type reference).
- **If we want to initialize an object during creation, we must build an initialization method**

Creation with Initialization 2/2

The object can then be created using its initialization method

```
p : POINT ;  
create p.make(23,64) ;; -- create and initialize a Point
```

- Multiple initialization methods can be defined for a same class.
The correct method is chosen during the creation.
- When (at least) one initialization method is declared for an class, this class cannot be created without calling one of these routines.
⇒ Security

Access to Class Member Variables

READING By default, all members are readable: everyone can know the value of it (but restriction can be applied).

WRITTING Members are **NEVER** writable **except for the current object**. The object mus provide a setter!

`set_x_y(x, y : INTEGER)` de la classe POINT.

⇒ **Security**

Access to Class Member Variables

READING By default, all members are readable: everyone can know the value of it (but restriction can be applied).

WRITTING Members are **NEVER** writable **except for the current object**. The object mus provide a setter!
`set_x_y(x, y : INTEGER)` de la classe POINT.
⇒ **Security**

Method without arguments doesn't have an empty pair of parenthesis: **this helps to keep API stable**

Eiffel Overview

- An object-oriented program structure in which a class serves as the basic unit of decomposition
- Static Typing
- Protection against calls on null references, through the attached-types mechanism
- Objects that wrap computations (closely connected with closures and lambda calculus)
- Garbage Collection
- Simple Concurrent Object-Oriented Programming
- Constrained and unconstrained generic programming *in a latter lecture*
- Design by contract *(latter in this lecture)*
- Fine grained (multiple) inheritance handling *(latter in this lecture)*

Part II

Object-Oriented Paradigms

Table of Contents

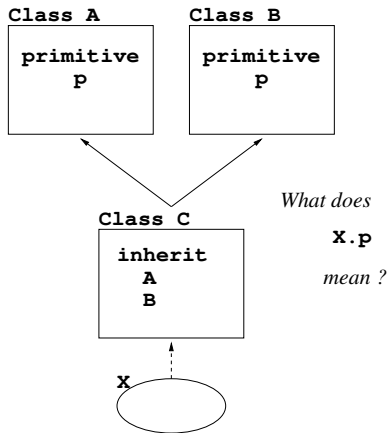
- 4 Handling Multiple inheritance
- 5 Programming "by Contract"
- 6 Reflection in OOP
- 7 Other models of OOP

Problem Statement

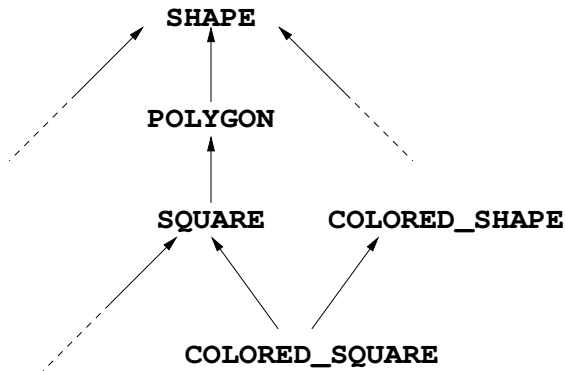
- **Simple Inheritance:** a class may inherit at most from only one class
- **Multiple Inheritance:** more powerful than the simple inheritance **but** introduces problems.
Eiffel proposes the adaptation clauses to solve these problems.

Problem Statement

- **Simple Inheritance:** a class may inherit at most from only one class
- **Multiple Inheritance:** more powerful than the simple inheritance **but** introduces problems.
Eiffel proposes the adaptation clauses to solve these problems.

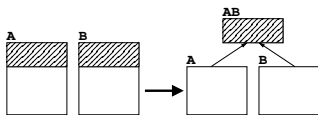


Multiple Inheritance is Sometimes Necessary

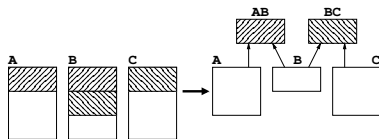


Inheritance for factorization

Simple inheritance helps to factorization:



And multiple inheritance is sometimes mandatory



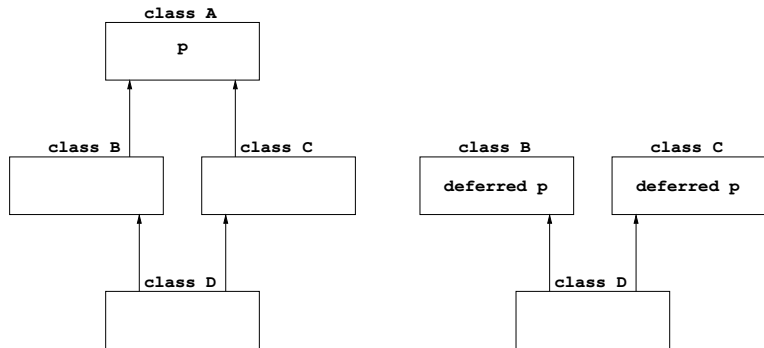
Smalltalk, Java, ... only propose a solution for modelisation while Eiffel also solves the factorization problems.

Quick Overview of the Other Languages

- Multiple inheritance is forbidden because it raises numerous problems and it is not necessary.
⇒ Java, Smalltalk, Ada
- Choose a lookup strategy and the programmer must conform it:
⇒ C++
- Propose tools (in the language) for solving problems related to multiple inheritance
⇒ Eiffel's inheritance adaptation clauses.

Jointure of primitives

Two corner cases :



deferred is an Eiffel keyword meaning **virtual** in C++

Adaptation Clauses

Features:

- Rename inherited primitives

Adaptation Clauses

Features:

- Rename inherited primitives
- Modify Visibility of inherited primitives

Adaptation Clauses

Features:

- Rename inherited primitives
- Modify Visibility of inherited primitives
- **A-definition** inherited primitives (make a primitive virtual)

Adaptation Clauses

Features:

- Rename inherited primitives
- Modify Visibility of inherited primitives
- **A-definition** inherited primitives (make a primitive virtual)
- Redefine inherited primitives

Adaptation Clauses

Features:

- Rename inherited primitives
- Modify Visibility of inherited primitives
- **A-definition** inherited primitives (make a primitive virtual)
- Redefine inherited primitives
- Selection clauses

Adaptation Clauses

Features:

- Rename inherited primitives
- Modify Visibility of inherited primitives
- **A-definition** inherited primitives (make a primitive virtual)
- Redefine inherited primitives
- Selection clauses

With these operations, we can resolve all problems related to multiple inheritance.

Table of Contents

4 Handling Multiple inheritance

- Renaming Clauses
- Visibility Filter
- Redefinition Clauses
- Selection Clauses
- A-definition

5 Programming "by Contract"

6 Reflection in OOP

7 Other models of OOP

Renaming Clauses

```
class SQUARE

inherit
  SHAPE
    rename
      make as make_shape
    end ;

feature
  width : INTEGER ;
  make(x,y : INTEGER ;
      w : INTEGER) is
  do
    make_shape(x,y) ;
    width := w ;
  end ;

end -- class SQUARE
```

- The renamed primitive is still accessible but with a different name.
- The original name can then be used for another primitive even with a different signature.

(French) Example

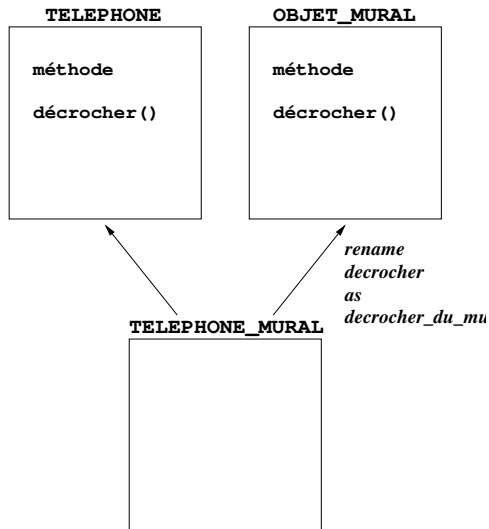


Table of Contents

4 Handling Multiple inheritance

- Renaming Clauses
- **Visibility Filter**
- Redefinition Clauses
- Selection Clauses
- A-definition

5 Programming "by Contract"

6 Reflection in OOP

7 Other models of OOP

Visibility Filter

```
class SQUARE

inherit
  SHAPE
    rename make as make_shape
    export {NONE} make_shape
end ;

feature

width : INTEGER ;

make(x,y : INTEGER ;
     w : INTEGER) is
do
  make_shape(x,y) ;
  width := w ;
end ;

end -- class SQUARE
```

- make_shape was accessible without reasons in class SQUARE
- May help to mask inherited primitive

Access Restrictions

`feature` ou `feature{ANY}`

primitives with default access value
(All objects derive from ANY)

`feature{A,B,C,...}`

primitives with access restricted only to some classes A, B, C

`feature{}` ou `feature{NONE}`

unreachable primitives
(NONE : no instance from this classe)

Table of Contents

4 Handling Multiple inheritance

- Renaming Clauses
- Visibility Filter
- **Redefinition Clauses**
- Selection Clauses
- A-definition

5 Programming "by Contract"

6 Reflection in OOP

7 Other models of OOP

Redefinition Clauses

```
class SQUARE
```

```
  inherit
```

```
    SHAPE
```

```
      rename make as make_shape
```

```
      export {NONE} make_shape
```

```
      redefine draw
```

```
    end ;
```

```
feature
```

```
  draw(g : GRAPHICS) is
```

```
    do
```

```
      ...
```

```
    end ;
```

```
  ...
```

- Constraints on redefintions
- Each redefinition must be declared
- Redefined methods are targetted by the dynamic lookup

Redfinir et conserver

On redéfinit pour profiter de la recherche dynamique.

Here, we loose dynamic lookup

```
class B  
  
inherit  
  A  
  rename p as pa end;  
  
feature  
  
  p ... is ...
```


Redfinir et conserver

On redéfinit pour profiter de la recherche dynamique.

Here, we loose dynamic lookup

```
class B

inherit
  A
  rename p as pa end;

feature

  p ... is ...
```

Here, dynamic lookup will work:

```
class B

inherit
  A
  rename p as pa end;
  A
  redefine p end;

feature

  p ... is ...
```

Table of Contents

4 Handling Multiple inheritance

- Renaming Clauses
- Visibility Filter
- Redefinition Clauses
- Selection Clauses
- A-definition

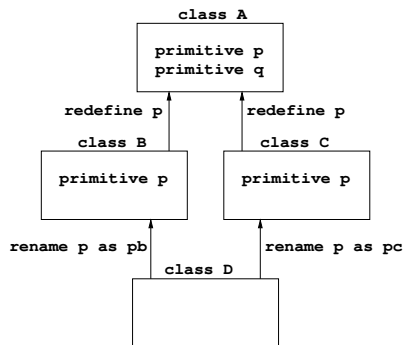
5 Programming "by Contract"

6 Reflection in OOP

7 Other models of OOP

Selection Clauses

How to resolve this problem:



Given $x : A$, what does $x.p$ means? If x references an instance of class A, it is the primitive p from A. Same thing happens for an object of B or C. What about instances of class D ?

Example:

```
q() is do p() end ;
```

Table of Contents

4 Handling Multiple inheritance

- Renaming Clauses
- Visibility Filter
- Redefinition Clauses
- Selection Clauses
- **A-definition**

5 Programming "by Contract"

6 Reflection in OOP

7 Other models of OOP

A-definition

The A-definition allows to undefine methods

Useful to "delete" methods that don't make sense anymore.

```
class TELEPHONE_MURAL
inherit
  TELEPHONE ;
  OBJET_MURAL
      undefine decrocher
  end ;
...
```

Table of Contents

- 4 Handling Multiple inheritance
- 5 Programming "by Contract"**
- 6 Reflection in OOP
- 7 Other models of OOP

Table of Contents

- 4 Handling Multiple inheritance
- 5 Programming "by Contract"
 - Definitions & Goals
 - Pre/Post-conditions
 - Redefinitions
 - Class Invariants
 - Assertions & Loop (in)variants
- 6 Reflection in OOP
- 7 Other models of OOP

What is that?

"It is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea?"

—

Charles Antony Richard Hoare

Goals

In everyday life a service or a product typically comes with a contract or warranty: **an agreement in which one party promises to supply the service or product for the benefit of some other party.**

An effective contract for a service specifies requirements:

- Conditions that the consumer must meet in order for the service to be performed
⇒ **Preconditions**
- Condition that the provider must meet in order for the service to be acceptable
⇒ **Postconditions**

Some History

- Has roots in work on formal verification, formal specification and Hoare logic
- First introduced by Eiffel
- Supported natively by Ada (2012), D, C#
- Libraries to emulate it in Java (cofoja), Javascript (contract.js), Python (pycontracts), C++ (Boost) ...

Contracts

A lot of ontracts:

- Pre-conditions and postconditions of a method
- Class invariants
- Assertions
- Loop invariants

Contracts

A lot of ontracts:

- Pre-conditions and postconditions of a method
- Class invariants
- Assertions
- Loop invariants

Contracts are part of the language:

- a dedicated syntaxe
- compiled (or not) according to the given options
- used by the compiler
- used by the environnemnt
- used by the documentation

Table of Contents

- 4 Handling Multiple inheritance
- 5 Programming "by Contract"
 - Definitions & Goals
 - Pre/Post-conditions
 - Redefinitions
 - Class Invariants
 - Assertions & Loop (in)variants
- 6 Reflection in OOP
- 7 Other models of OOP

Pre-conditions

Pre-conditions must be fulfilled by the client, i.e. based on arguments

```
class SHAPE
feature
  xc, yc : INTEGER ; -- coordinates

  set_x_y(x, y : INTEGER) is
    require
      x >= 0 and y >= 0
    do
      xc = x ;
      yc = y ;
    end ;
...

```

Pre-conditions in **Eiffel**

Post-conditions

Post-conditions must be fulfilled by the provider, i.e. if the client fulfills preconditions, the provider will fulfill postconditions.

```
class SHAPE
feature
  ...
  set_x_y(x,y : INTEGER) is
    require
      x >= 0 and y >= 0
    do
      xc := x ;
      yc := y ;
    ensure
      xc = x and yc = y
    end ;
```

Post-conditions in Eiffel

Referencing previous version of an expression

old x reference the value of x before the execution of the method

```
class RECTANGLE

feature
  width, height : INTEGER ;

  set_width(w : INTEGER) is
    require
      w > 0
    do
      width := w
    ensure
      width = w and height = old height
    end ;

  ...
```

Referencing previous value in Post-conditions (**Eiffel**)

Stripping Objects

In a postcondition, `strip(x,y,...)` references an object where all attributes `x` and `y, ...` have been removed

```
class RECTANGLE

feature
  width, height : INTEGER ;

  set_width(w : INTEGER) is
    -- change the width
    require
      w > 0
    do
      width := w
    ensure
      width = w and strip (width) = old strip (width)
    end ;
```

Table of Contents

- 4 Handling Multiple inheritance
- 5 Programming "by Contract"**
 - Definitions & Goals
 - Pre/Post-conditions
 - Redefinitions**
 - Class Invariants
 - Assertions & Loop (in)variants
- 6 Reflection in OOP
- 7 Other models of OOP

Redefinition (1/2)

class A

```
routine p is require ... ensure ... end ;  
  
routine q is do p() ; end ;
```

redefine p

class B

```
routine p is do ... end ;
```

The redefined method `p` in `B` can be used instead of the original method `p` de `A`.

⇒ Assertions are inherited

Redefinition (2/2)

The redefined method must satisfy old assertions but can be more precise:

Redefinition (2/2)

The redefined method must satisfy old assertions but can be more precise:

- Release some preconditions

Redefinition (2/2)

The redefined method must satisfy old assertions but can be more precise:

- Release some preconditions
- Add (Restrict) postconditions

Redefinition (2/2)

The redefined method must satisfy old assertions but can be more precise:

- Release some preconditions
- Add (Restrict) postconditions

```
class B
  inherit
    A redefine p end ;
  feature
    p is
      require else
        ... -- other restrictions
      do
        ... -- new definition
      ensure then
        ... -- additional postconditions
      end ;
  end -- class
```

Table of Contents

- 4 Handling Multiple inheritance
- 5 Programming "by Contract"
 - Definitions & Goals
 - Pre/Post-conditions
 - Redefinitions
 - **Class Invariants**
 - Assertions & Loop (in)variants
- 6 Reflection in OOP
- 7 Other models of OOP

Class Invariants

A Class Invariant is an assertion attached to an object. The inherited class also inherits invariants.

```
class RECTANGLE
  ...

  invariant
    (xc < 0 implies width > -xc) -- visible
  and
    (yc < 0 implies height > -yy) -- visible
  and
    width >= 0
  and
    height >= 0

end -- class RECTANGLE
```

Table of Contents

- 4 Handling Multiple inheritance
- 5 Programming "by Contract"**
 - Definitions & Goals
 - Pre/Post-conditions
 - Redefinitions
 - Class Invariants
 - **Assertions & Loop (in)variants**
- 6 Reflection in OOP
- 7 Other models of OOP

Assertions

Can be inserted anywhere in the code.

```
-- Code
check
  x > 0 ;
  y < 0 implies largeur > -y
end ;
```

Loop (in)variants

Only one (complex) kind of loop in Eiffel

```
from
    -- initialization
    ...
invariant
    -- checked each iteration
    ...
variant
    -- positive integer expression
    ...
until
    -- exit condition
    ...
loop
    -- loop body
    ...
end ;
```

Table of Contents

- 4 Handling Multiple inheritance
- 5 Programming "by Contract"
- 6 Reflection in OOP**
- 7 Other models of OOP

What is Reflection?

Reflection is the ability of a program to examine, introspect, and modify its own structure and behavior at runtime.

What is Reflection?

Reflection is the ability of a program to examine, introspect, and modify its own structure and behavior at runtime.

Reflection is not limited to OOP!

History

Introspection & Intercession

Introspection

The ability of a program to observe and therefore reason about its own state.

```
class MyReflectionClass{
    public static boolean classequal(Object o1, Object o2)
        Class c1, c2;
        c1 = o1.getClass();
        c2 = o2.getClass();
        return (c1 == c2);
    }
}
```

Reflection in **Java**

Introspection & Intercession

Introspection

The ability of a program to observe and therefore reason about its own state.

```
class MyReflectionClass{
    public static boolean classequal(Object o1, Object o2)
        Class c1, c2;
        c1 = o1.getClass();
        c2 = o2.getClass();
        return (c1 == c2);
    }
}
```

Reflection in **Java**

Encoding execution state as data (c1, c2) is called **reification**.

Intercession

Intercession

The ability of a program to modify its execution state or alter its own interpretation or meaning. *Create, Manipulate and call method.*

```
Class c = obj.getClass();  
Object o = c.newInstance();
```

```
String s = "FooBar".  
Class c = Class.forName(s);  
Object o = c.newInstance();
```

Intercession in Java

Deeper in Introspection

```
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
for (int i = 0; i < constructors.length; i++){
    Class params[] =
        constructors[i].getParameterTypes();
}
```

Enumeration of Constructors in **Java**

```
Class c = obj.getClass();
Field[] fields = c.getFields();
Object o = fields[2].get(obj);
...
fields[3].set(obj, value);
```

Modifying attributes in **Java**

What about other Programming Languages?

- **C#**: provides facilities to create CIL and assembly this code
- **Go**: reflections even on channels
- **Perl**: *Moose* (built on top of `Class::MOP`, a metaobject protocol) provides complete introspection for all Moose-using classes.
- **Delphi (Objective-Pascal)/C++**: only via RTTI (run-time type information)
- **Python**: the `dir(..)` function details the attributes of an object

Table of Contents

- 4 Handling Multiple inheritance
- 5 Programming "by Contract"
- 6 Reflection in OOP
- 7 Other models of OOP**

Table of Contents

- 4 Handling Multiple inheritance
- 5 Programming "by Contract"
- 6 Reflection in OOP
- 7 Other models of OOP**
 - Message Passing OOP

The operation that define the operation on objects of a class are called **methods**. The call to these methods are sometimes called **messages**. The collection of methods of a class is called the **message protocol**.