Objects

Akim Demaille, Etienne Renault, Roland Levillain

March 20, 2020

Table of Contents

Table of Contents

Simula

Figure: Ole-Johan Dahl

Simula

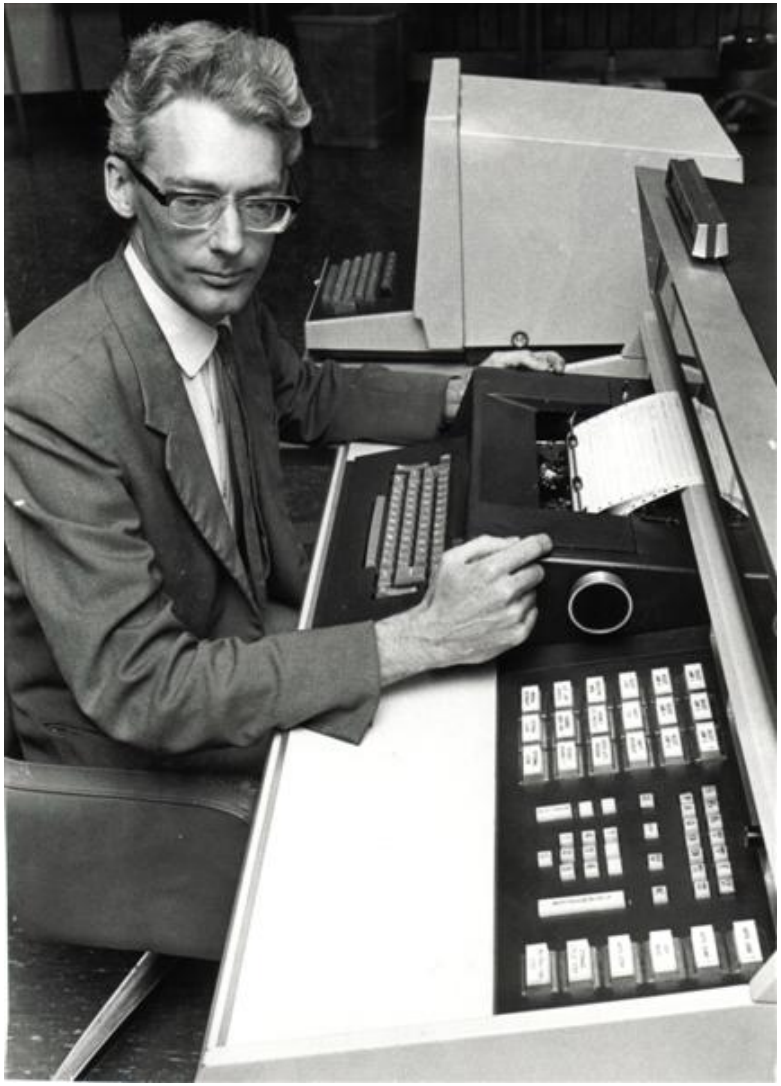


Figure: Ole-Johan Dahl

Simula

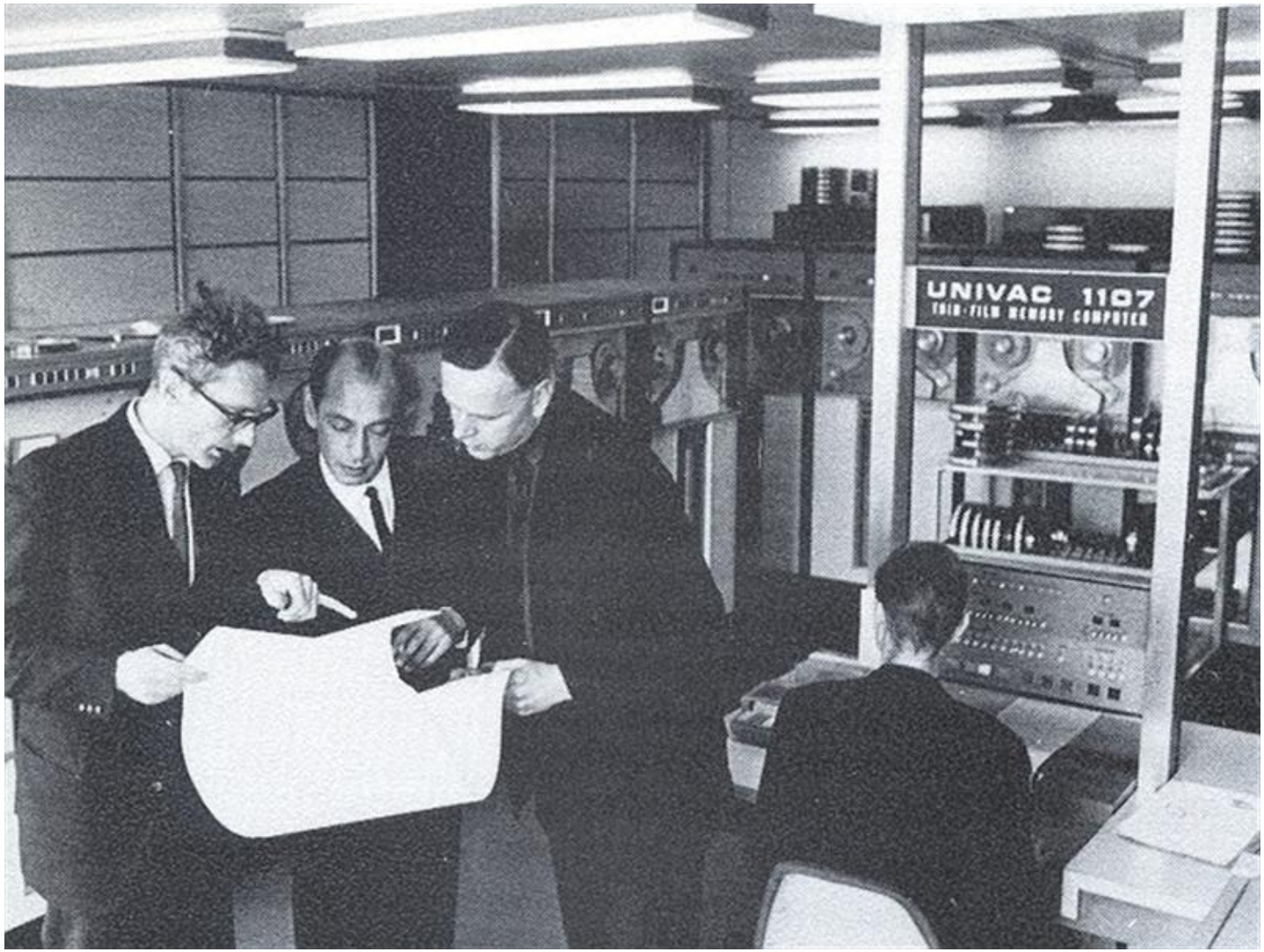


Figure: Dahl & Nygaard

Simula

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

Simula

Figure: Nygaard & Dahl: Turing Award 2001

2002... Sad Year

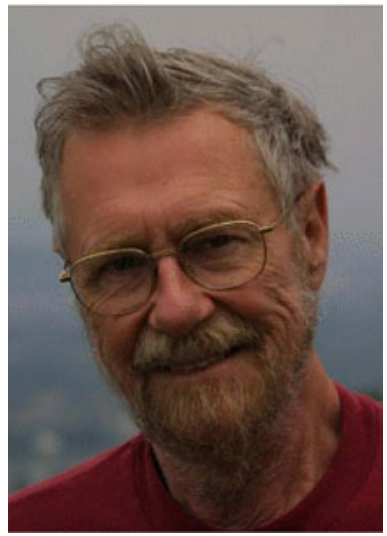
Ole-Johan Dahl

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

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) **Simu**lation **la**nguage.

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)

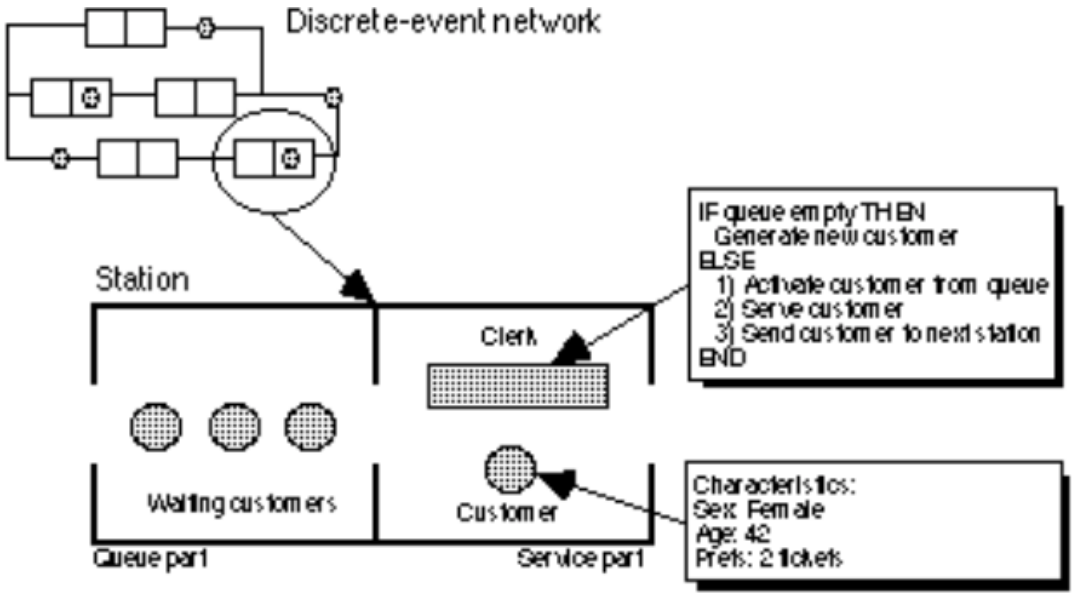
- A **system**, consisting of a finite and fixed number of active components named **stations**, and a finite, but possibly variable number of passive components named **customers**.
- A **station** consisting of two parts: a queue part and a service part. Actions associated with the service part, named 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

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

Smalltalk

> *We called Smalltalk Smalltalk so that nobody would expect anything from it.*
> – Alan Kay

Principles:

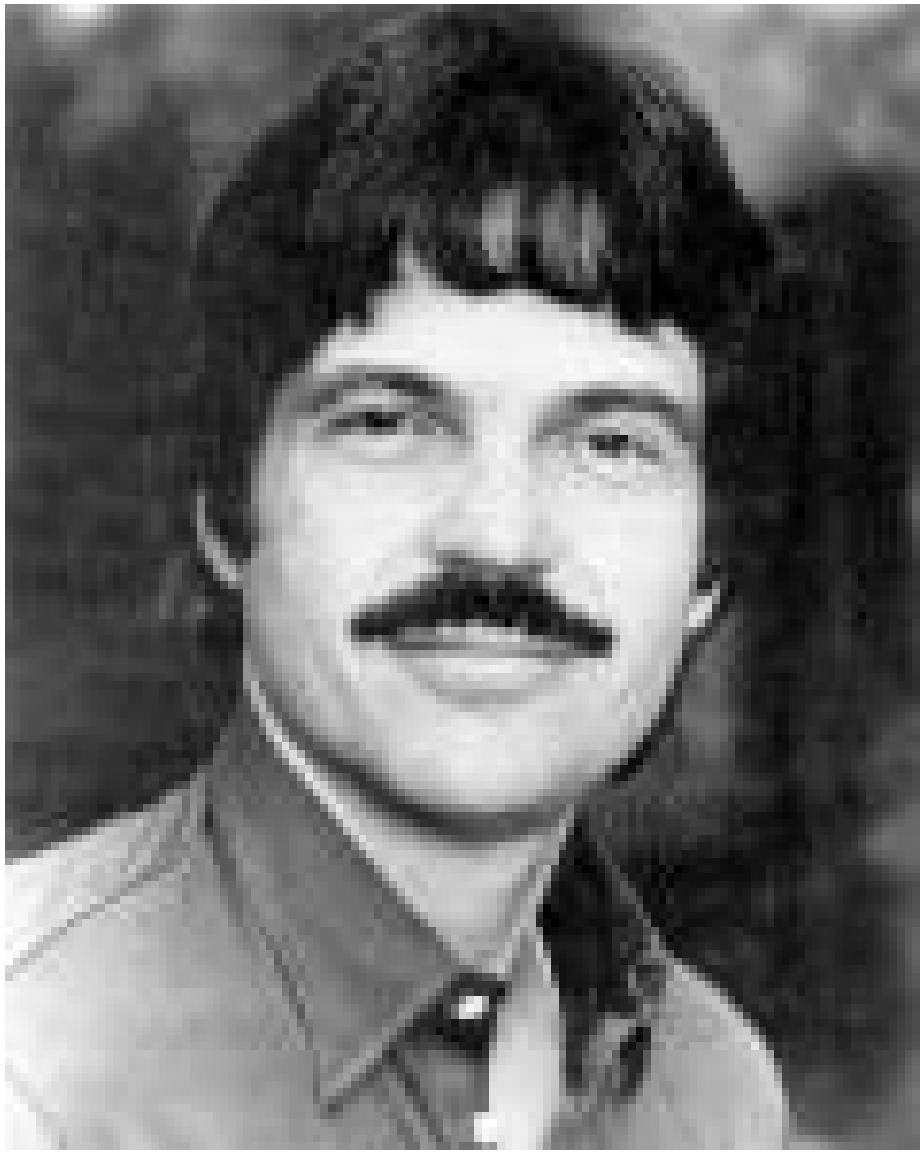
- Everything is object;
- Every object is described by its class (structure, behavior);
- Message passing is the only interface to objects.

Origin:

- A programming language that children can understand;
- To create "tomorrow's computer": Dynabook.

Table of Contents

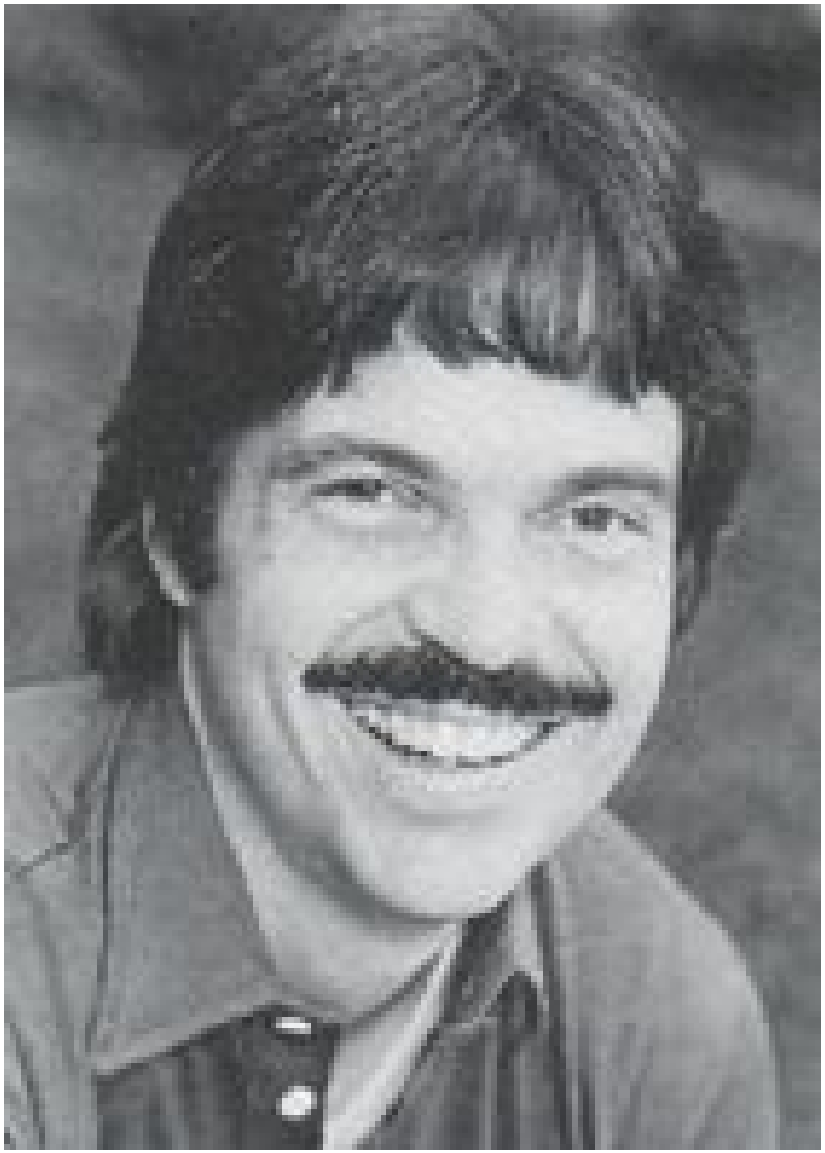
Alan Kay



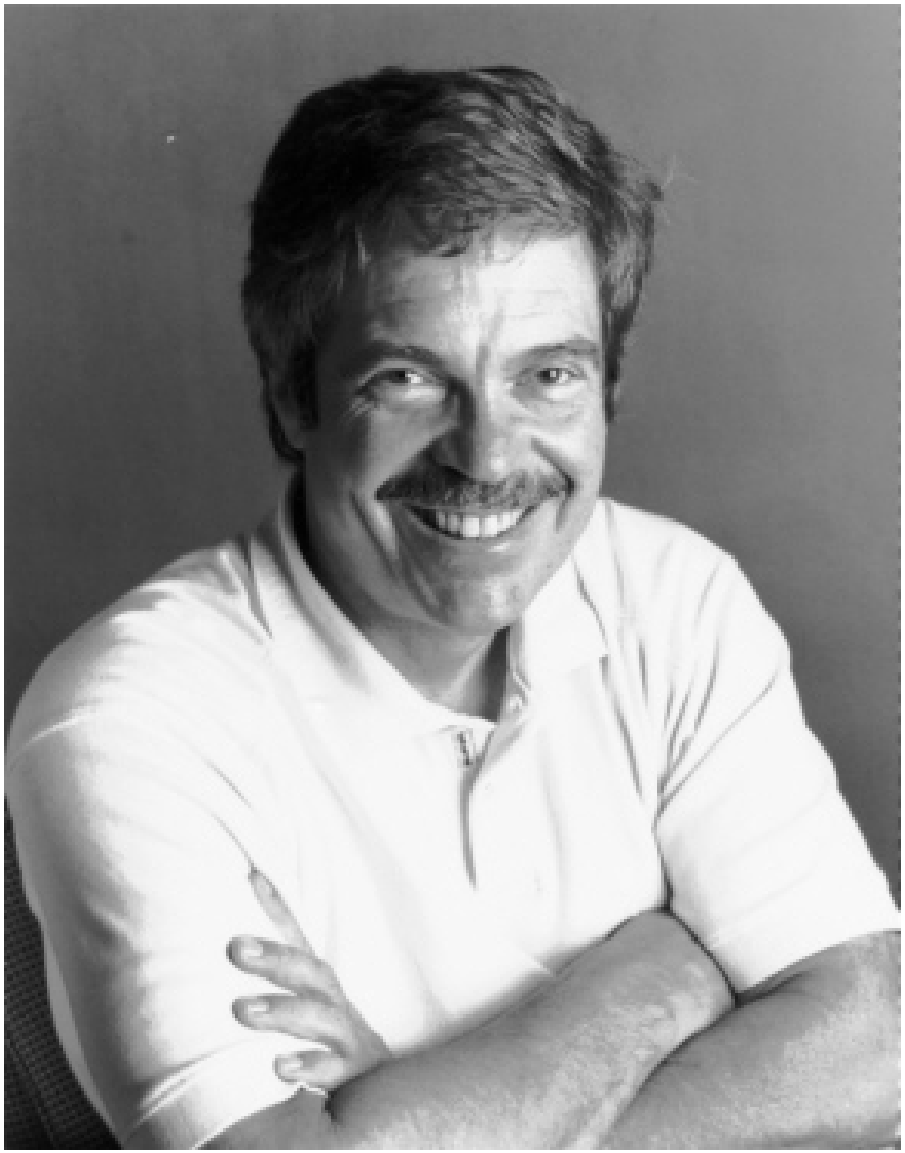
Quote

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

Alan Kay, 1984



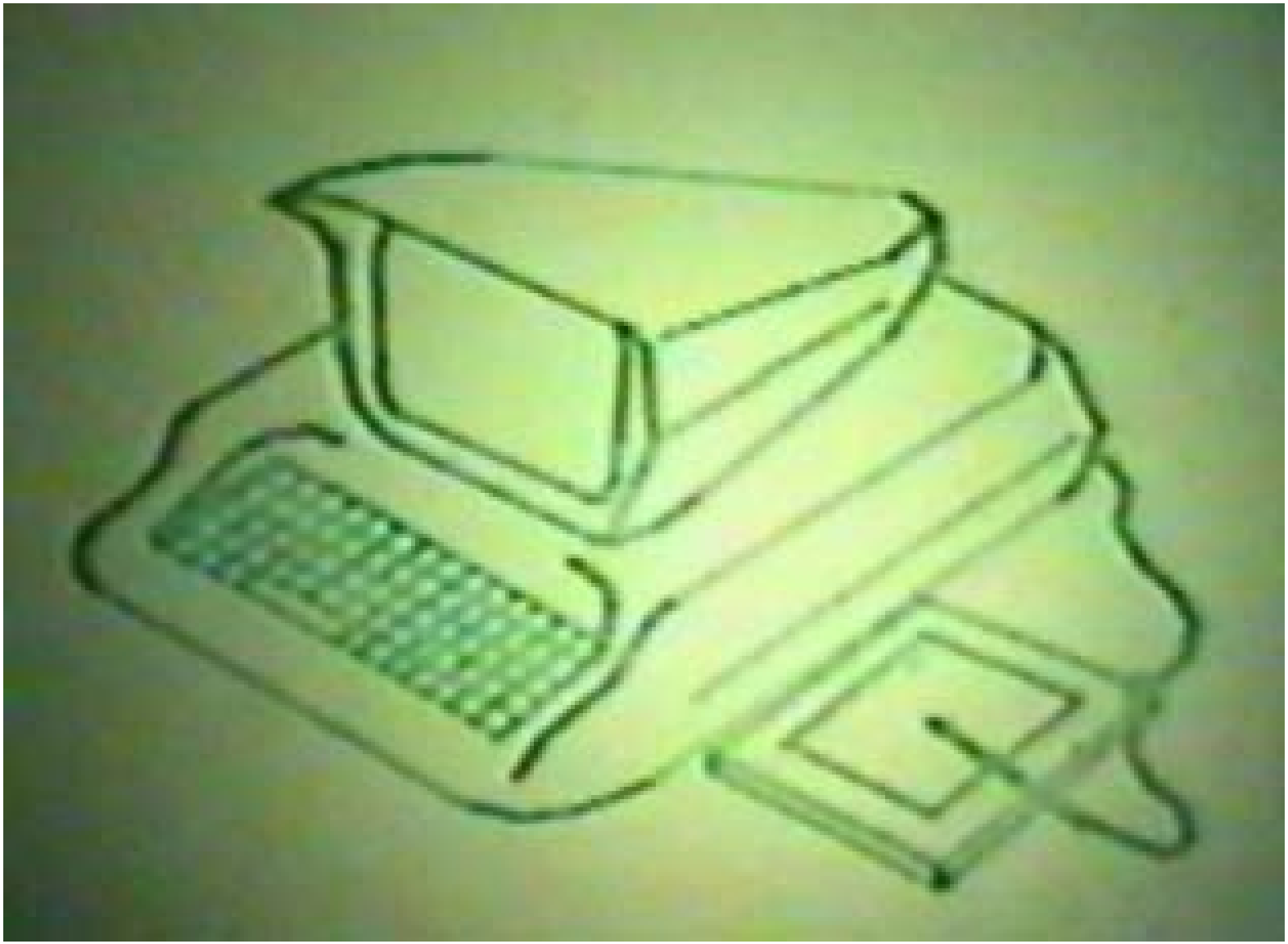
Alan Kay



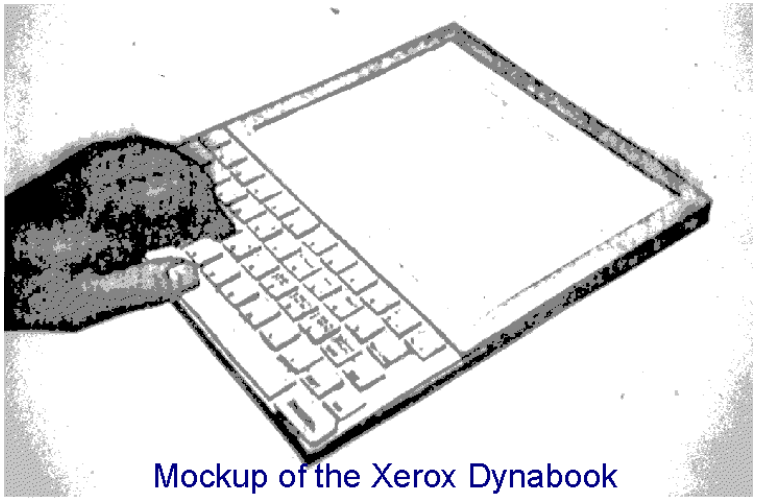
Ivan Sutherland's Sketchpad 1967

Douglas Engelbart's NLS 1974

Flex Machine 1967

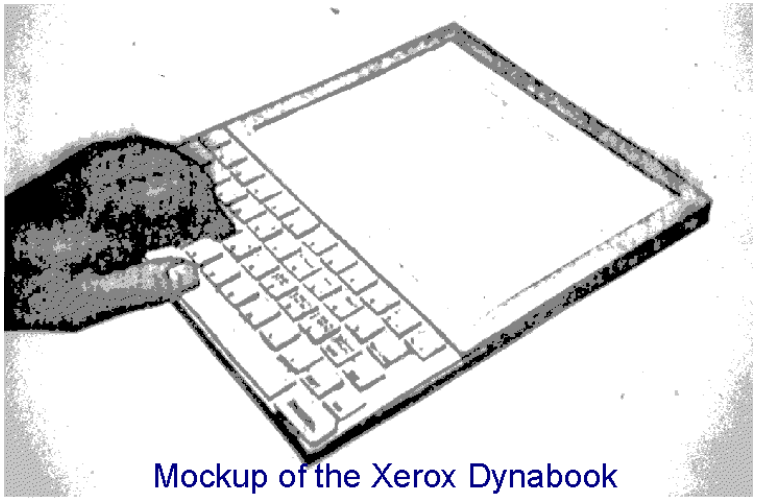


DynaBook


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.

DynaBook

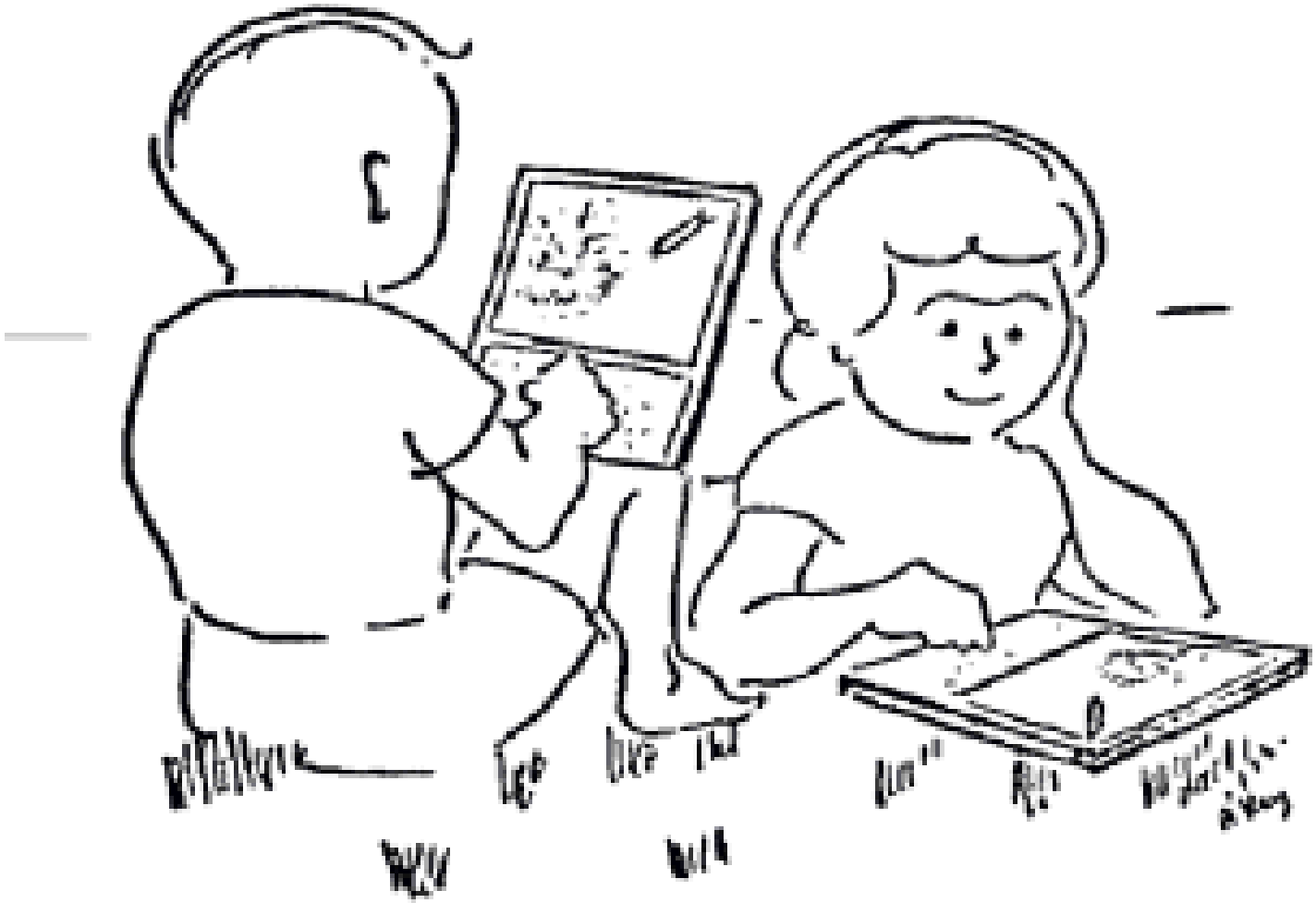


Table of Contents

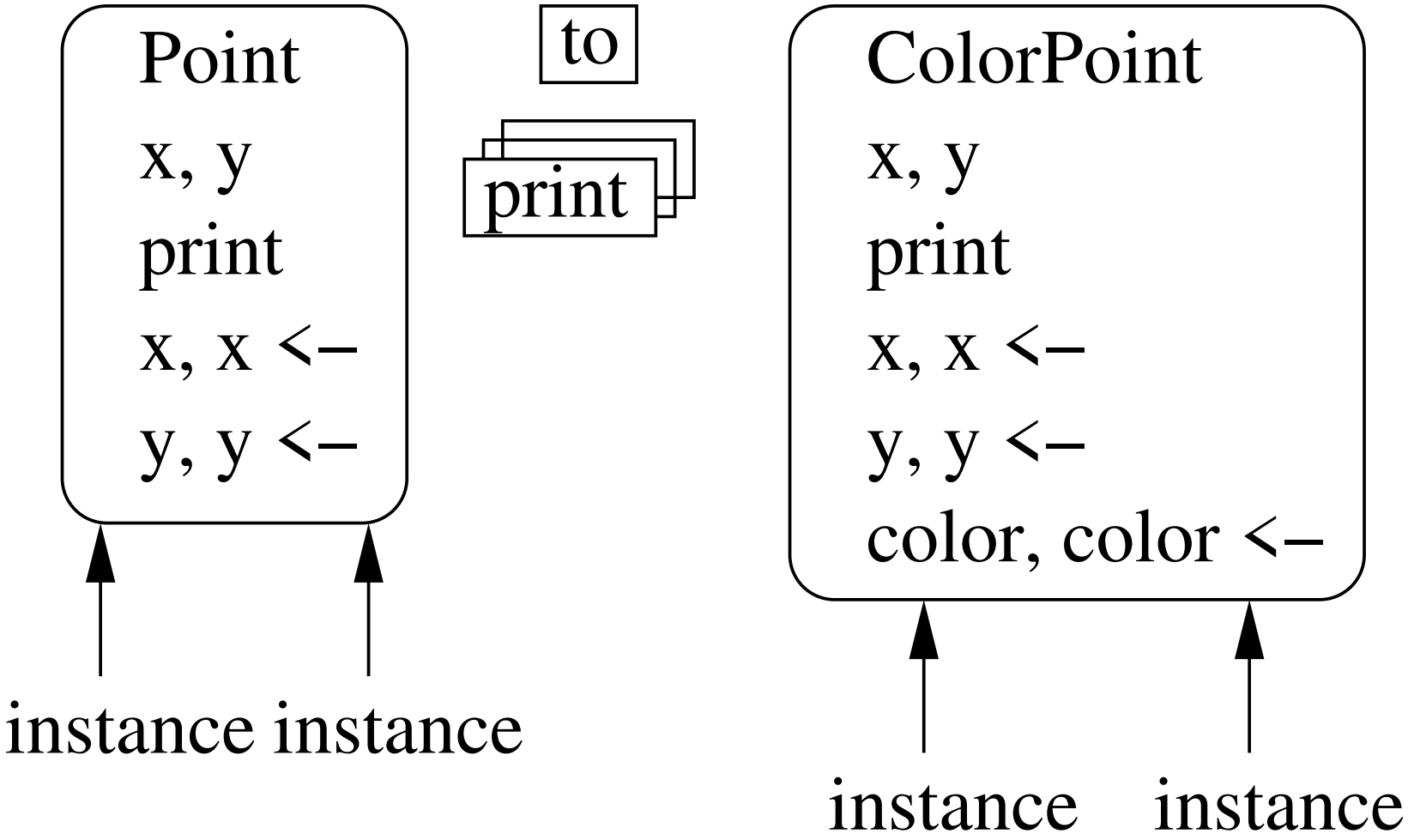
Smalltalk 72

- 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.

$\Rightarrow$ Programmers were using global procedures.
But some successes:

- Pygmalion
  "Programming by examples"
  inspired Star.

Table of Contents

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

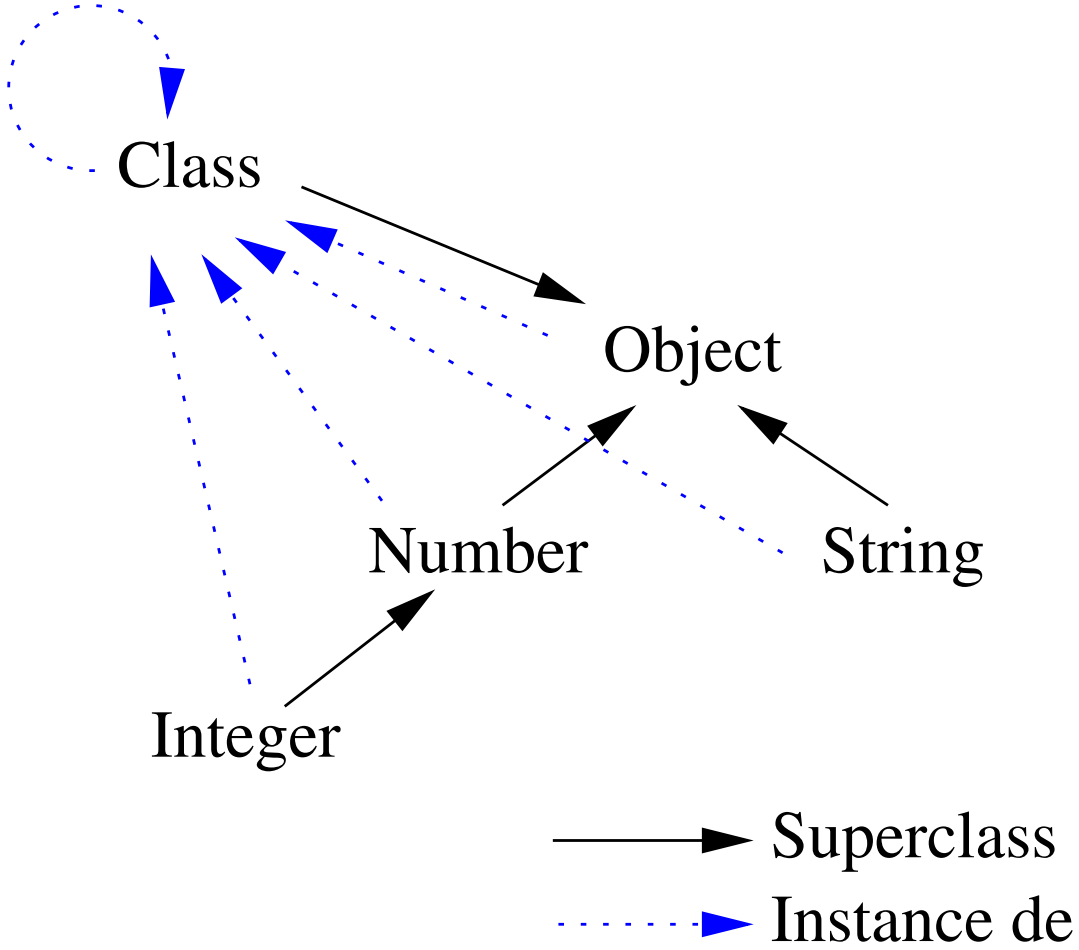
- Introduction of the `Class` class.
  The class of classes. Instance of itself. *Metaclass*. How to print a class, add method, instantiate etc.
- Introduction of the `Object` class.
  Default behavior, shared between all the objects.
- Introduction of dictionaries.
  Message handling is no longer handled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

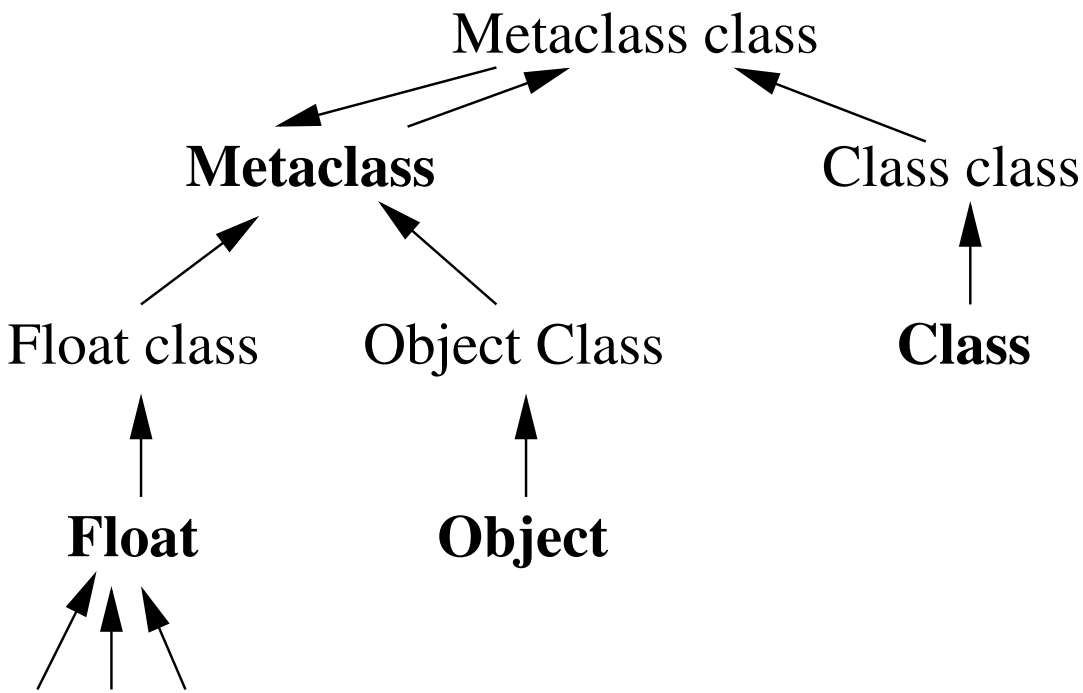
- Significant improvement:
  - Byte-code and a virtual machine provide a 4-100 speedup.
  - *ThingLab*, constraint system experimentation.
  - *PIE*, *Personal Information Environment*.
- But:
  - A single metaclass
    hence a single behavior for classes
    (no specific constructors, etc.).

Table of Contents

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



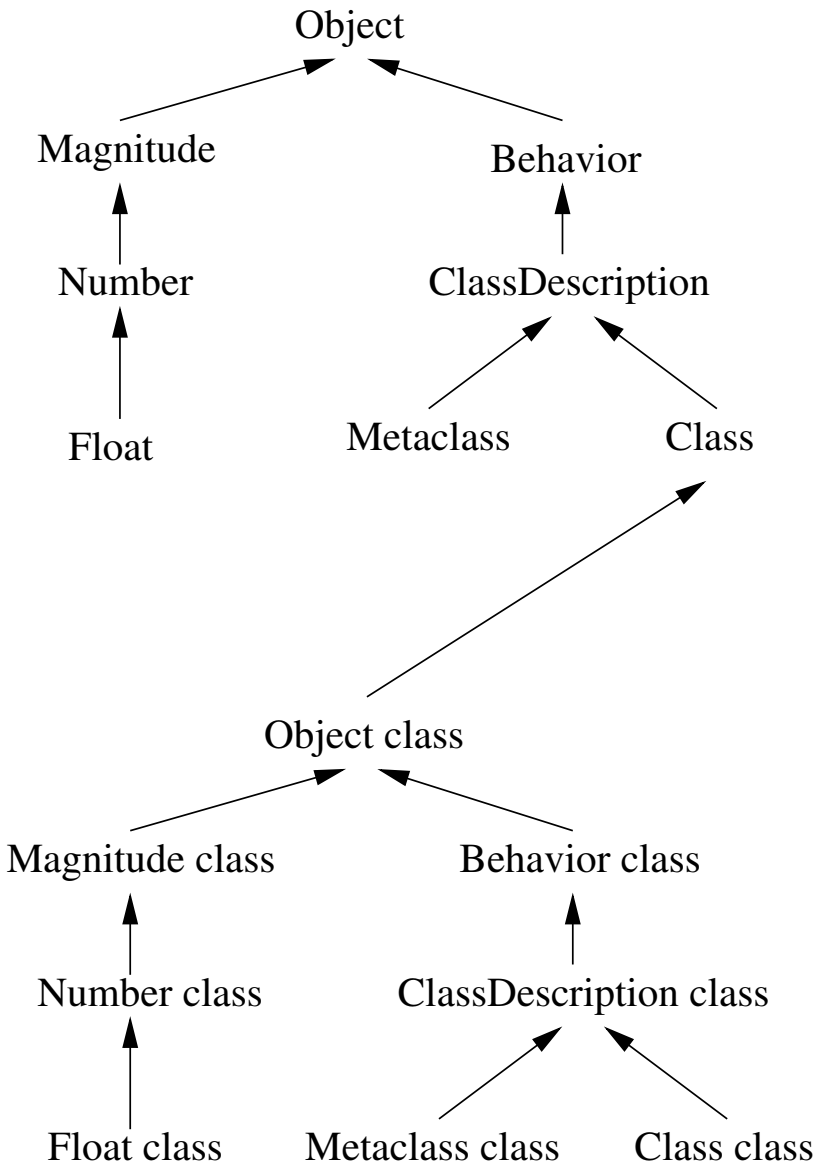
Three layer model:

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

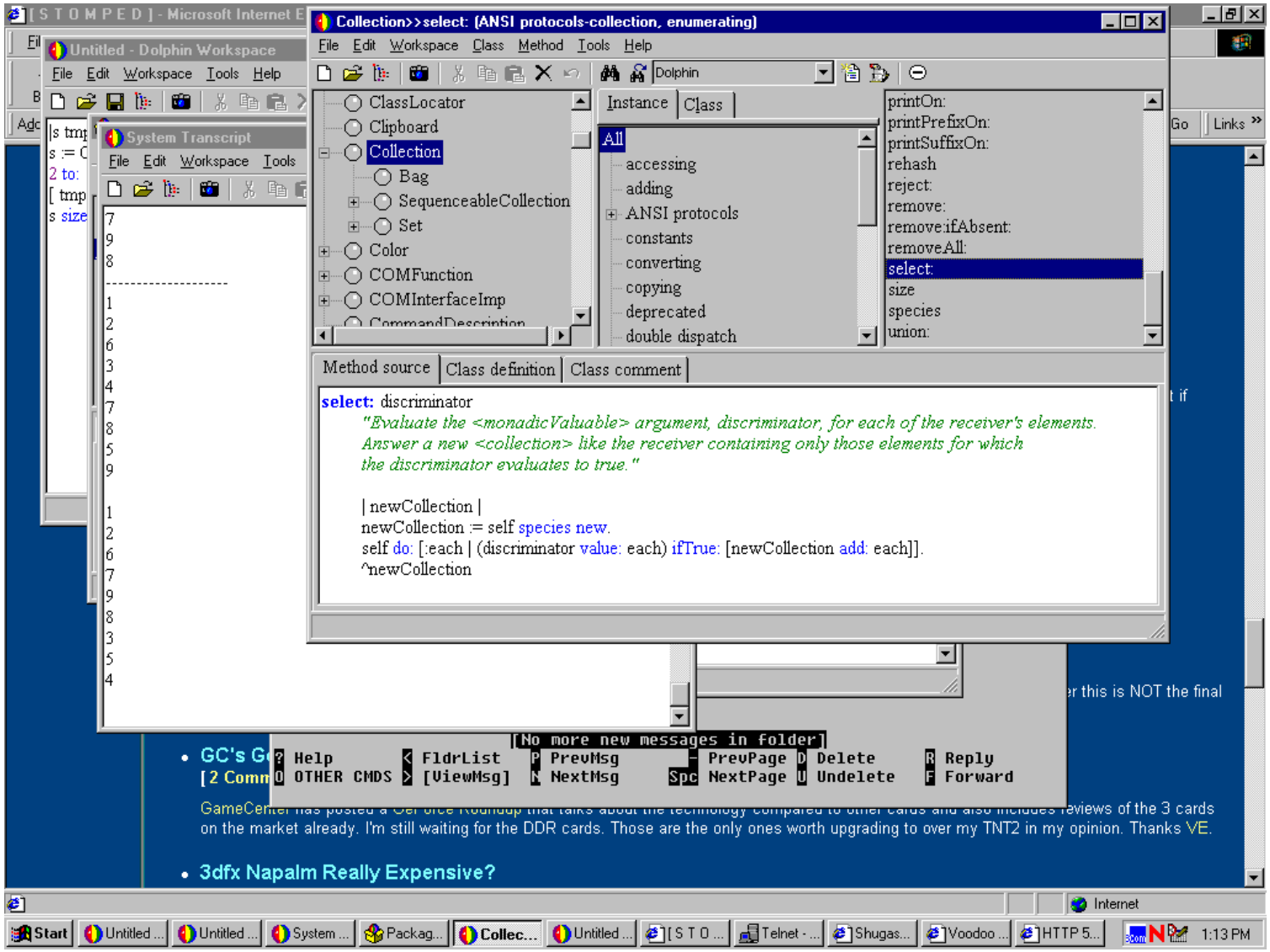
More than a language, a system where *everything* is an object, and the only control structure is message passing.

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

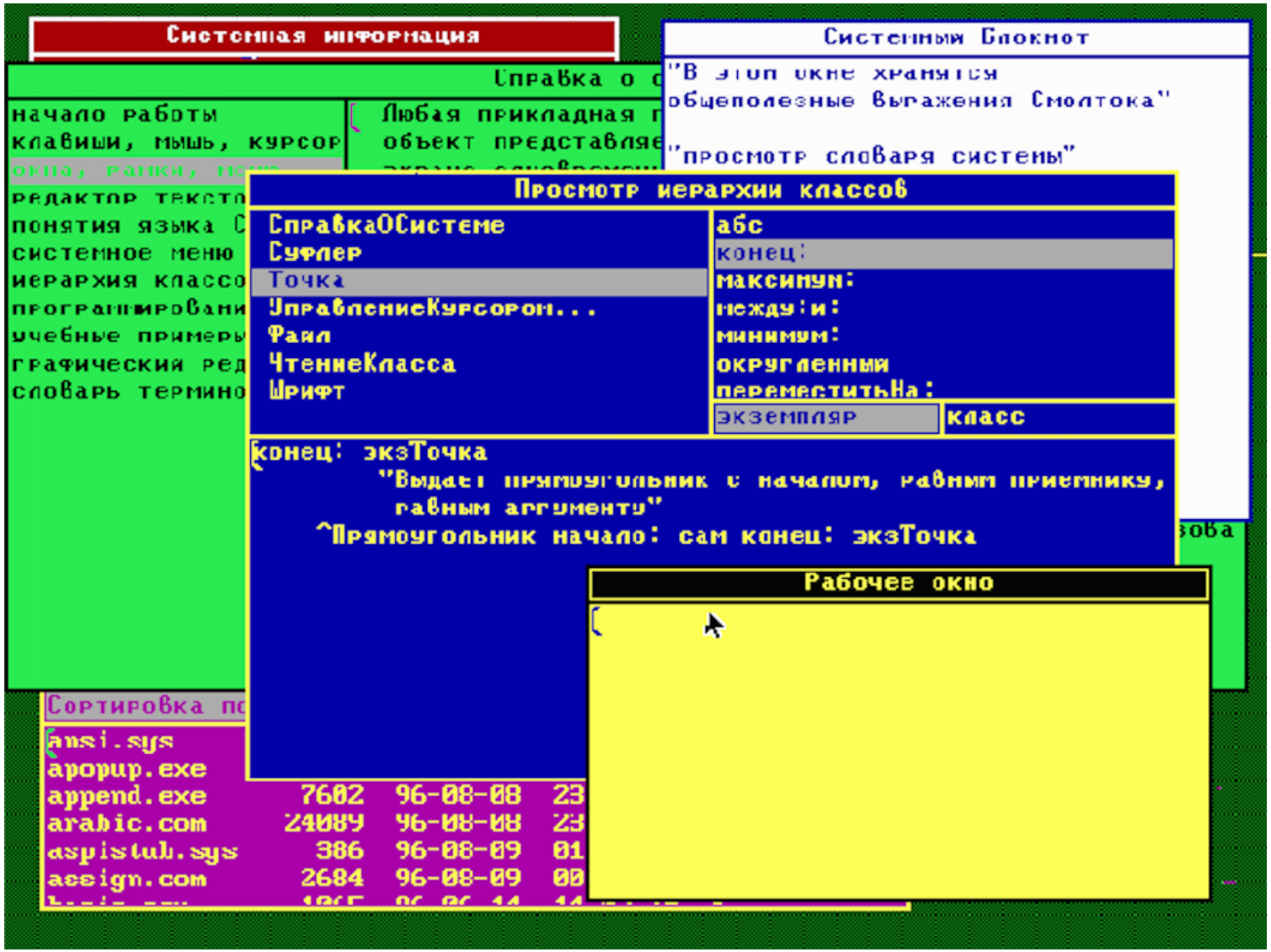
Smalltalk 80 Standard Library

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

Smalltalk 80



Smalltalk 80



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

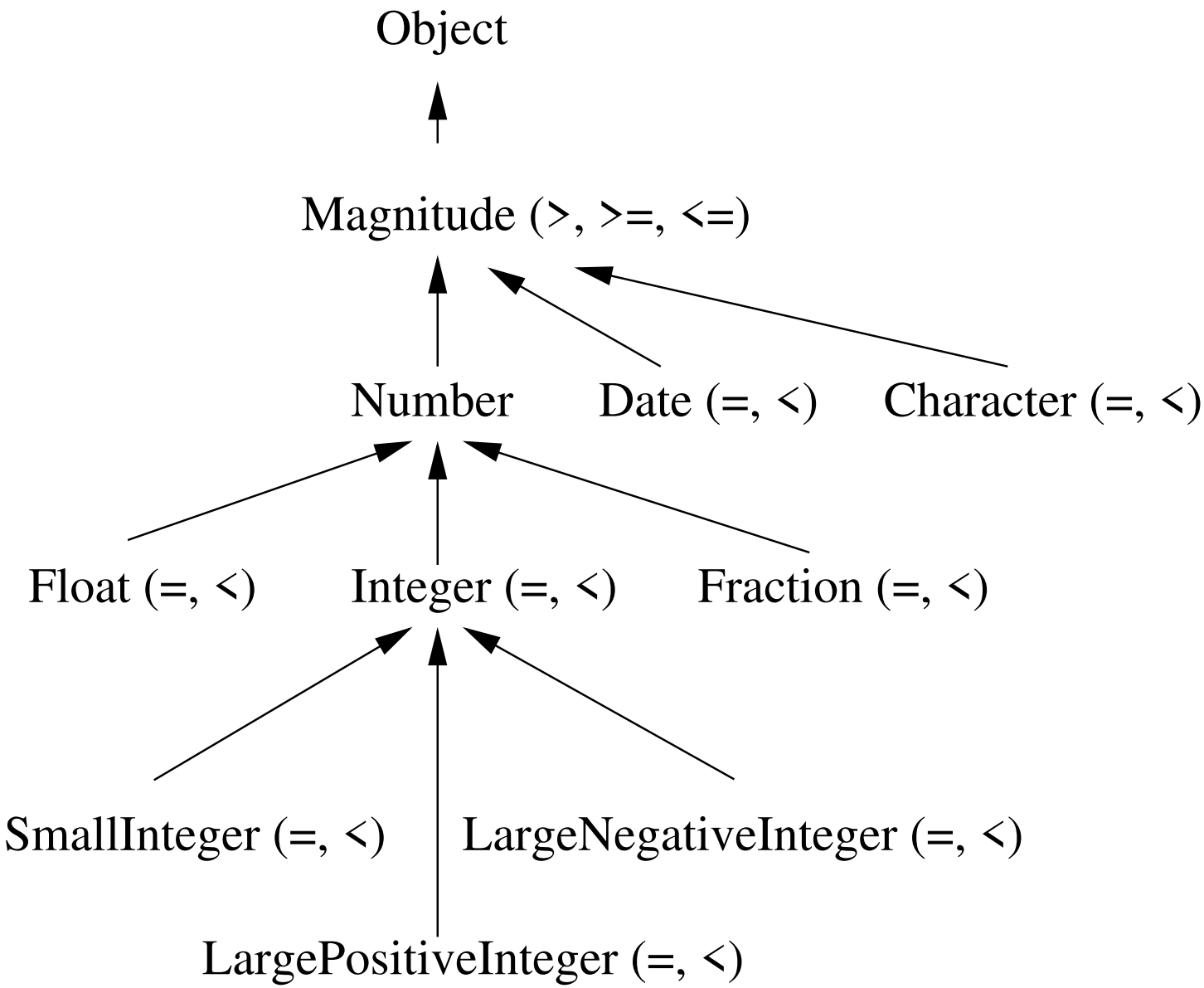
More Boolean methods:

- `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

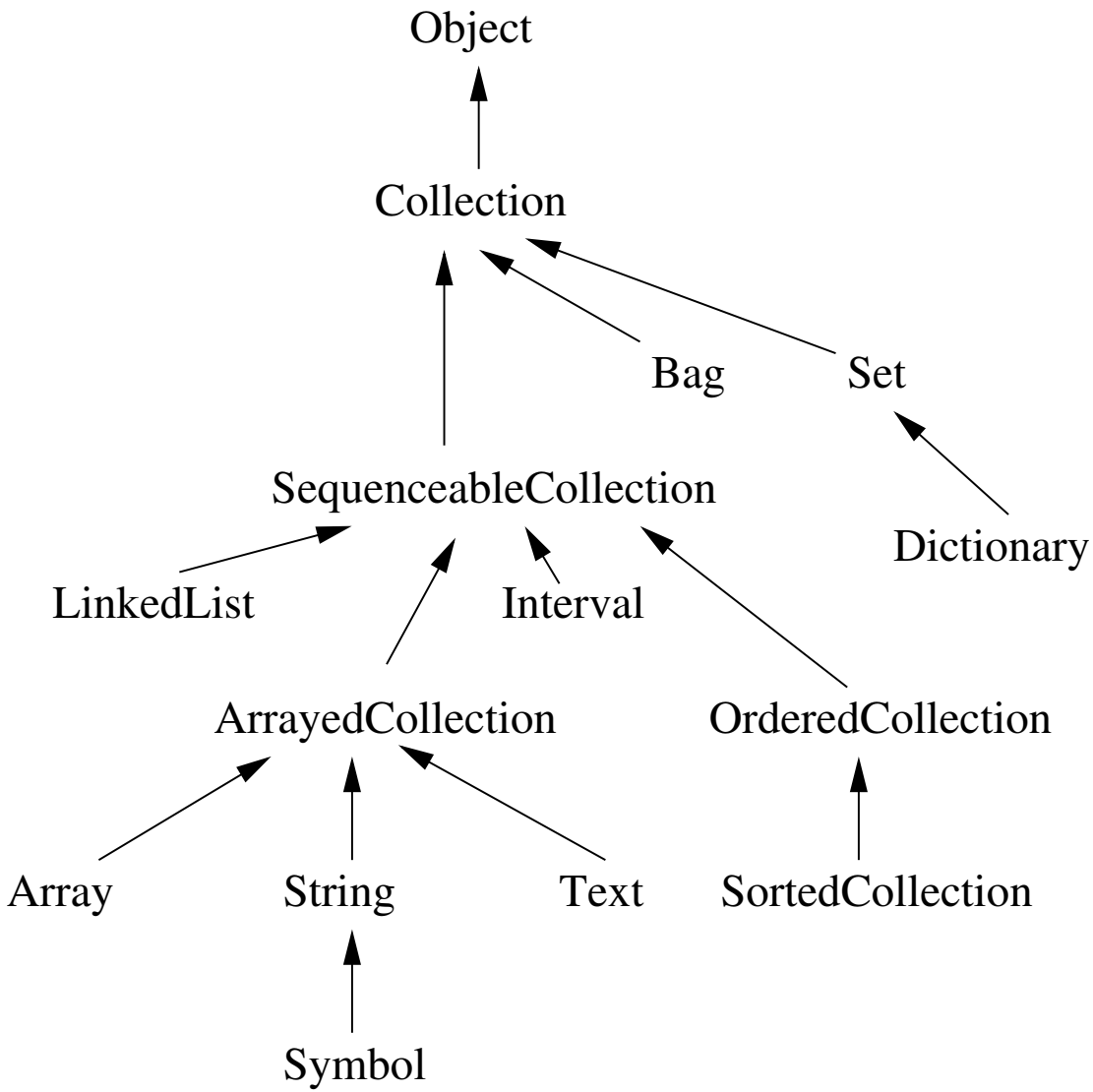
In `Magnitude`

```
>= 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 interpretor,
  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

Table of Contents

Reading...


Technologies objet
Fondateurs
Conception et programmation orientées objet
Bertrand Meyer
Eyrolles

What is this Book ?

Table of Contents

Bertrand Meyer (1950), MIT

Table of Contents

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`)
  - Groupped 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 antonomous 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 have 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 have 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 have 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

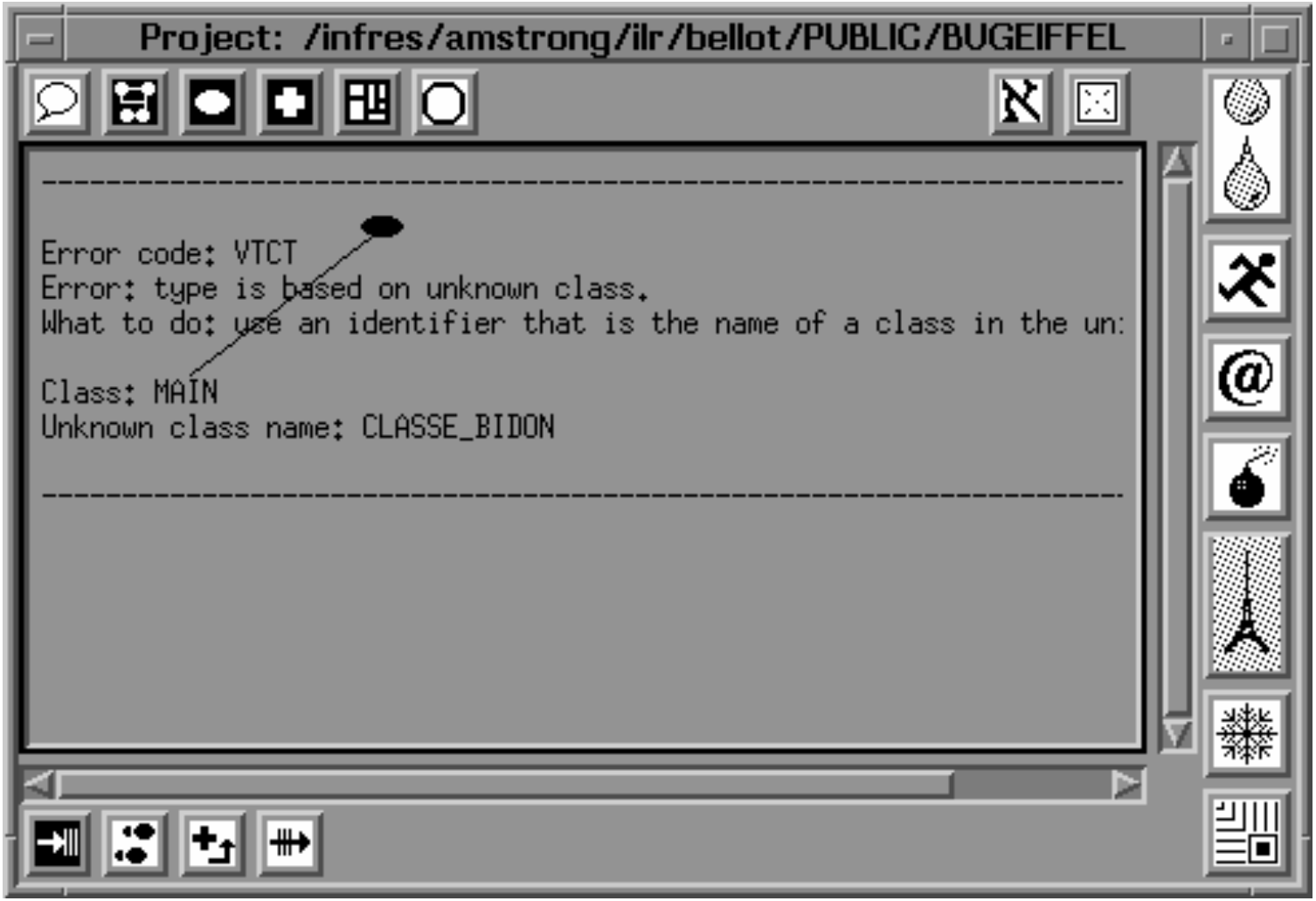
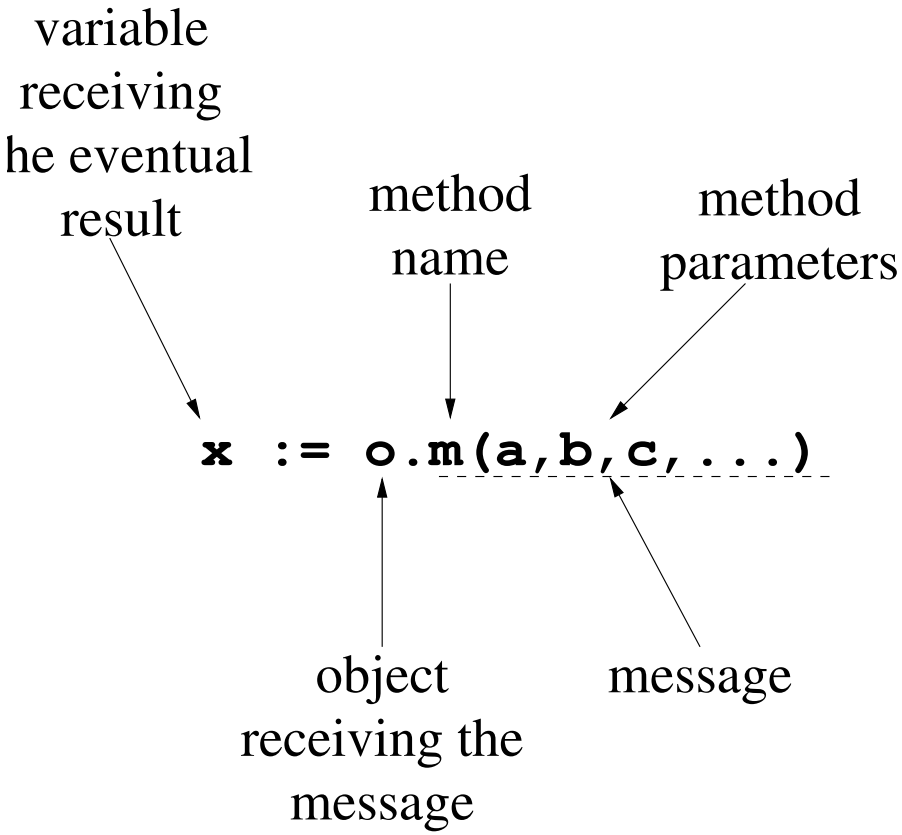

Project: /infres/amstrong/ilr/bellot/PUBLIC/BUGEIFFEL
Error code: VTCT
Error: type is based on unknown class.
What to do: use an identifier that is the name of a class in the un:
Class: MAIN
Unknown class name: CLASSE_BIDON

Table of Contents

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
```

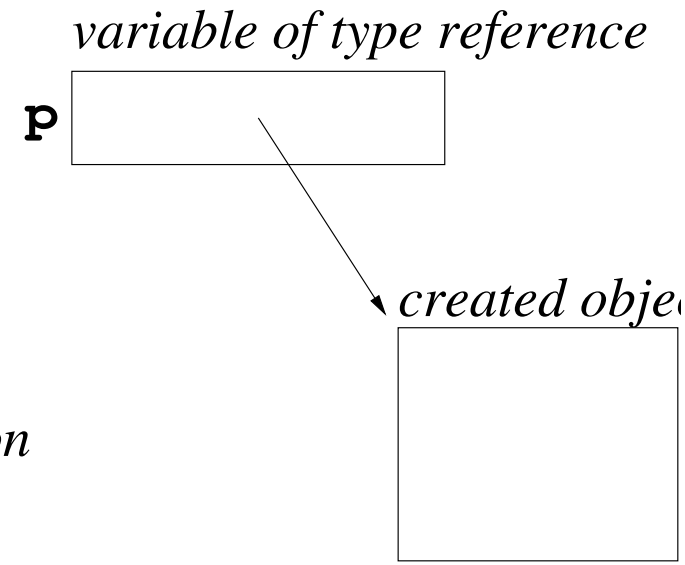
Methods Calls



- 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

```
p : POINT ;

do
    ...
    !! p ; -- instruction
    ...
end ;
```



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 a 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.
  $\Rightarrow$ 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 writtable **except for the current object**. The object mus provide a setter!
`set_x_y(x,y : INTEGER)` de la classe `POINT`.
$\Rightarrow$ 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 writtable **except for the current object**. The object mus provide a setter!
`set_x_y(x,y : INTEGER)` de la classe `POINT`.
$\Rightarrow$ Security

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

Expanded Class

By default, all types are manipulated by reference
(i.e., with indirection)

Performances issues ...

```
expanded class ERECTANGLE
inherit
RECTANGLE ;
end -- class ERECTANGLE
```

Like current

Let us consider the following snippet from class SHAPE:

```
copy_and_move(x,y : INTEGER) : SHAPE is do
    Result := Current.copy() ;
    Result.set_x_y(x,y) ;
end ;
```

And the following code :

```
C1 := C2.copy_and_move(22,23)
```

- No problem when C1 and C2 are SHAPE
- Problem: type is lost when C1 and C2 are of type SQUARE (that inherits from SHAPE)
  $\Longrightarrow$ **Effeil proposes like x or like current to solve this!**

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
- Fine grained (multiple) inheritance handling

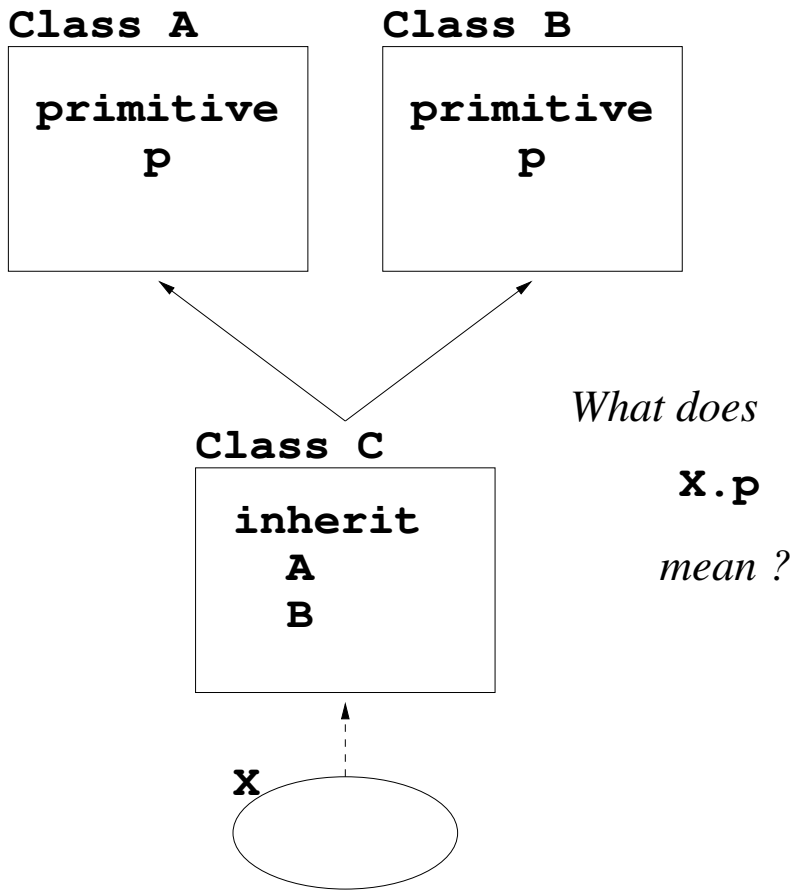
Table of Contents

Problem Statement

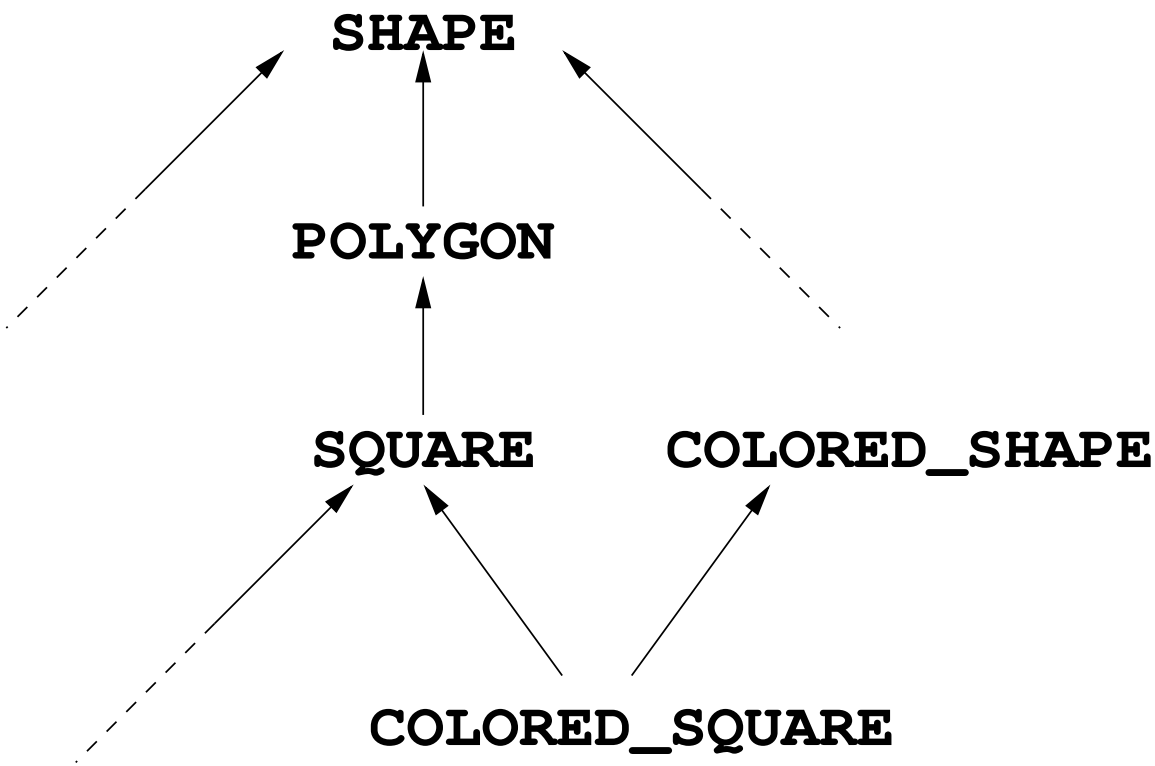
- **Simple Inheritance**: a class may herit at most from only one class
- **Multiple Inheritance**: more powerful than the simple inheritance **but** introduces problems. Eiffel proposes the adaptation clauses to solve these problems.

Problem Statement

- **Simple Inheritance**: a class may herit 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.

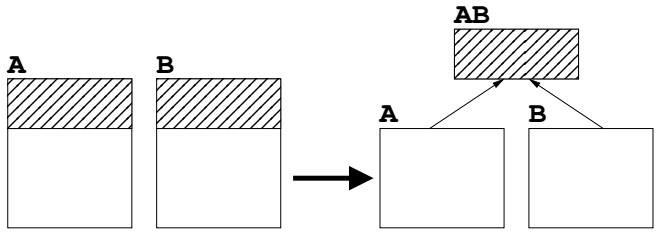


Multiple Inheritance is Sometimes Necessary

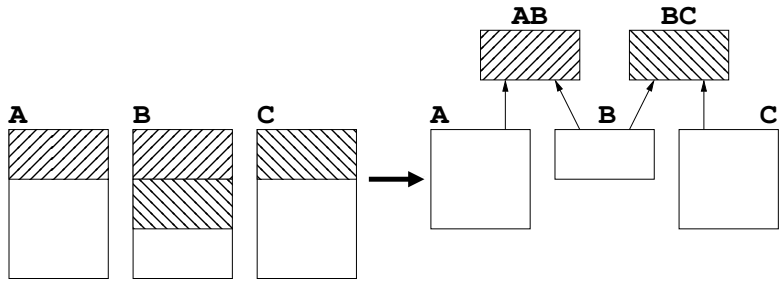


Inheritance for factorization

Simple inheritance helps to factorization:



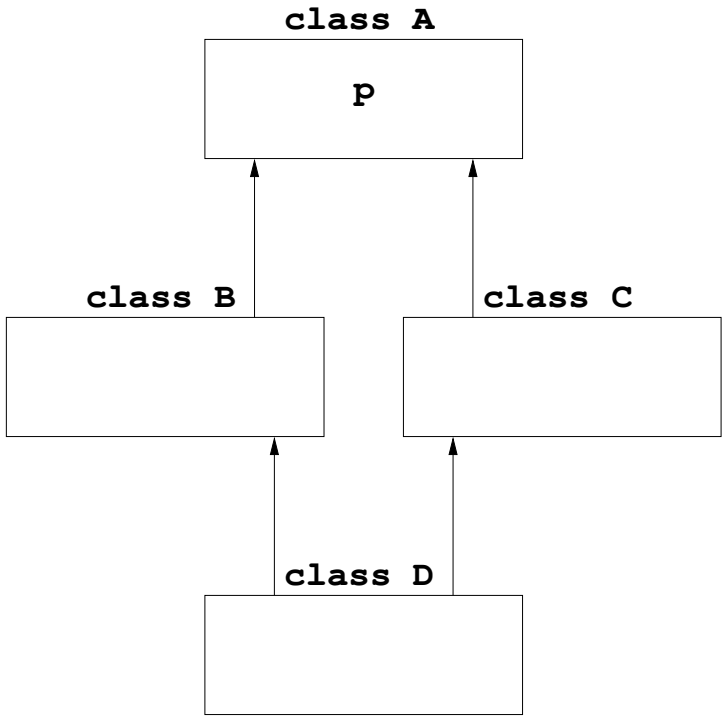
And multiple inheritance is sometimes mandatory

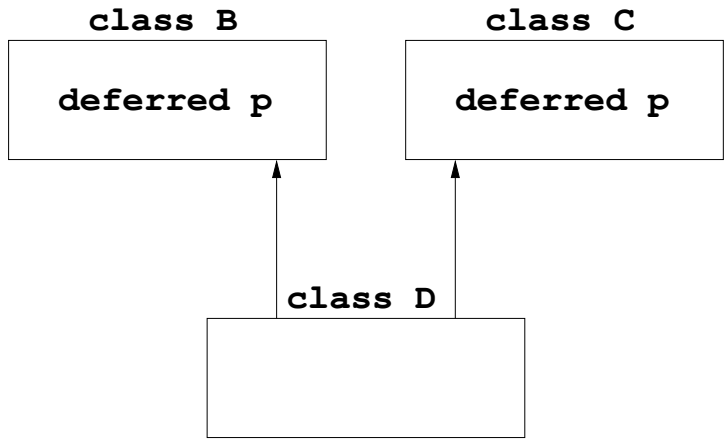


Smalltalk, Java, ... only propose a solution for modelisation while Eiffel also solves the factorization problems.

Jointure of primitives

Two corner cases :





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

Quick Overview of the Other Languages

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

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

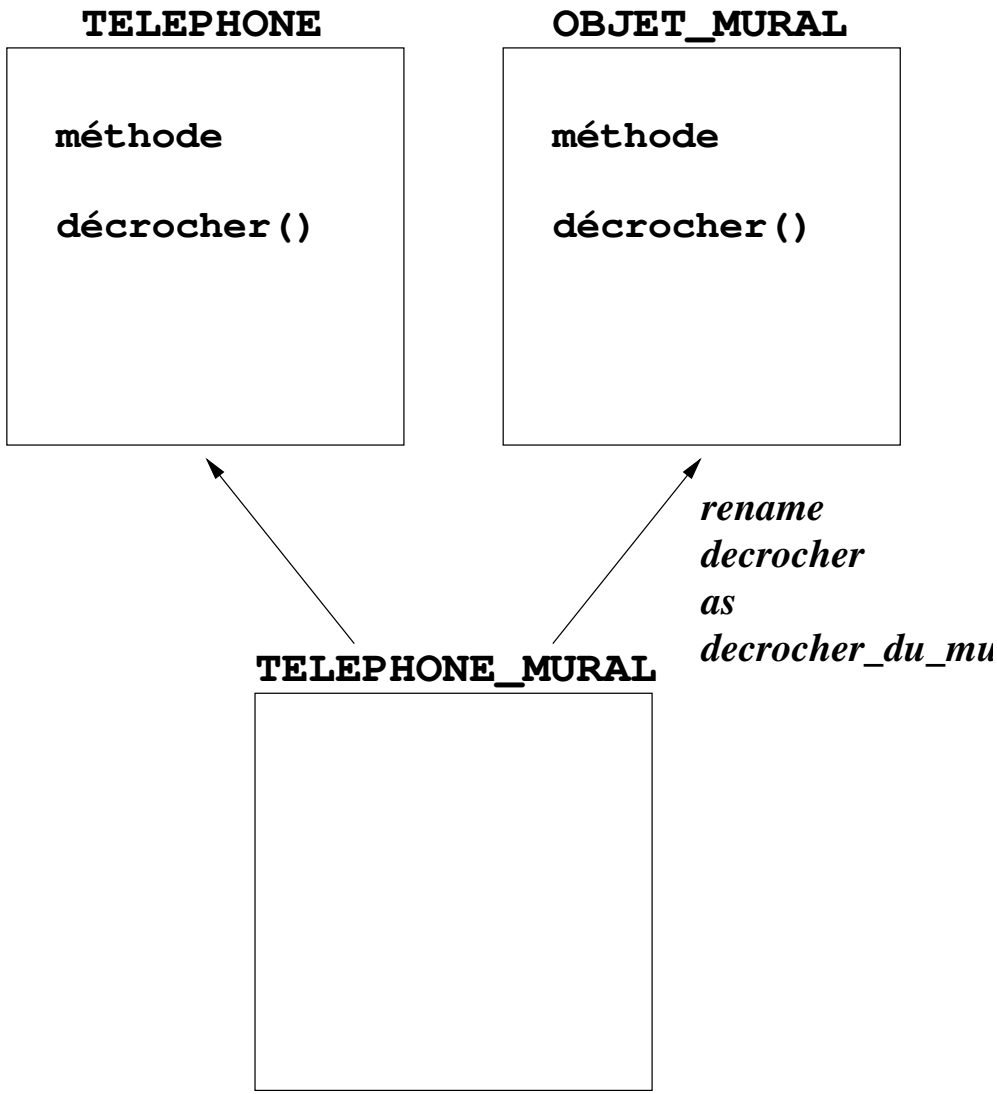
With these operations, we can resolve all problems related to multiple inheritance.

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



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)

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

Keep and redefine

Redefinition to support dynamic lookup

Here, we loose dynamic lookup

```
class B

inherit
  A
    rename p as pa end;

feature

    p ... is ...
```

Keep and redefine

Redefinition to support dynamic lookup

Here, we loose dynamic lookup

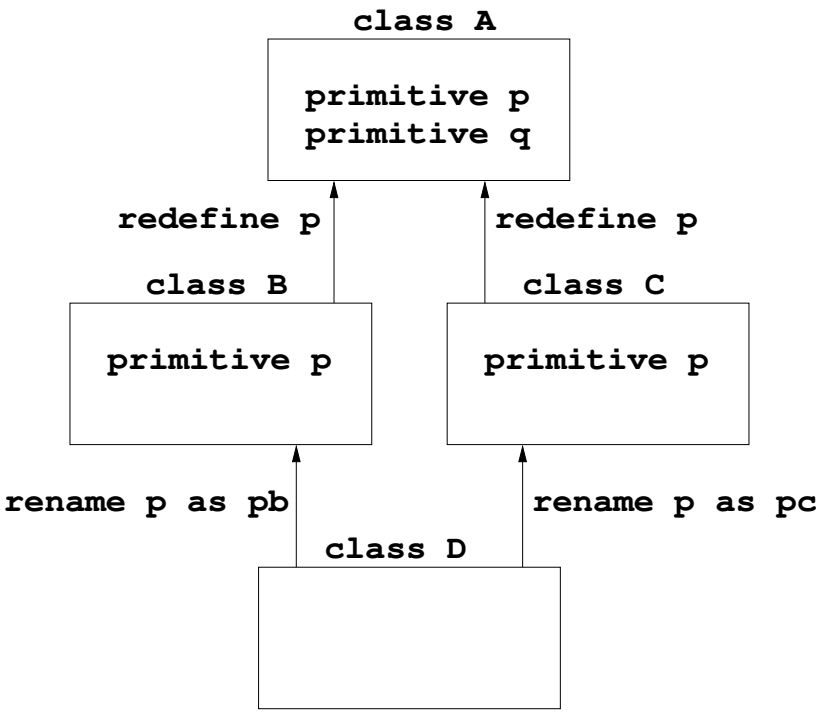
```
class B

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 ...
```

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` ou `C`. What about instances of class `D` ?

Example:

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

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

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
  $\Rightarrow$ Preconditions
- Condition that the provider must meet in order for the service to be acceptable
  $\Rightarrow$ Postconditions

Some History

- Has roots in work on formal verification, formal specification and Hoare logic
- First introducted by Eiffel
- Supported natively by Ada (2012), D, C#
- Librairies 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

Pre-conditions

Pre-conditions must be fulfill 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 fulfill by the provider, i.e. if the client fulfill preconditions, the provider will fulfill postcondiitons.

```
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**)

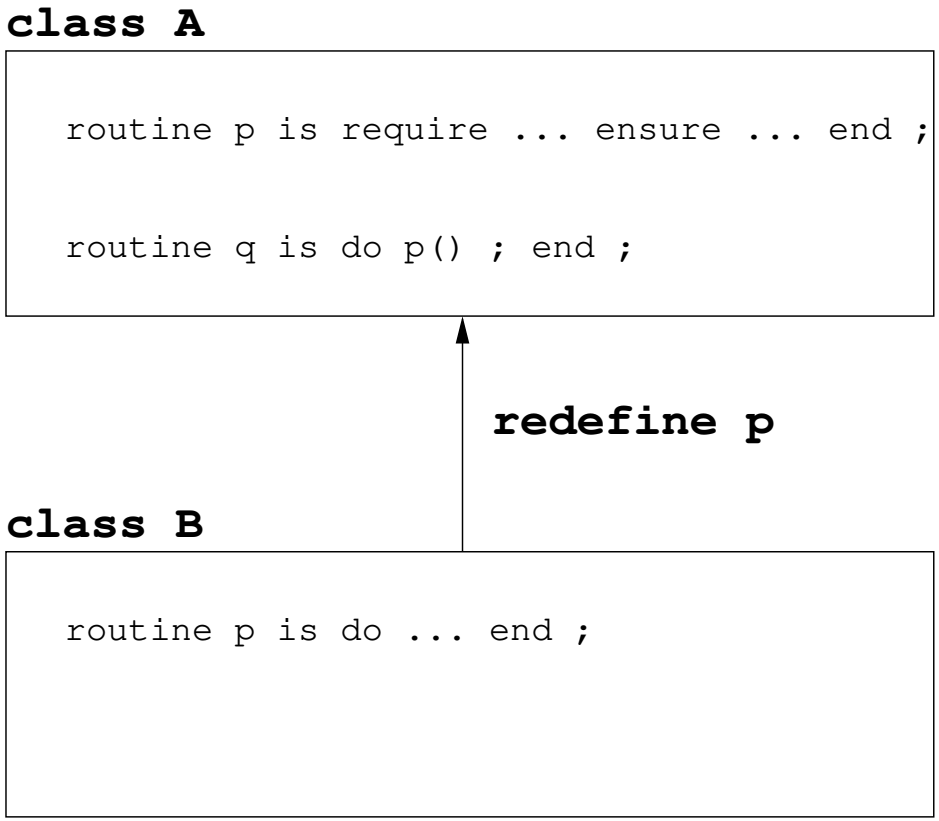
Stripping Objects

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 ;
```

Stripping Object in Postconditions (**Eiffel**)

Redefinition (1/2)



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

$\Rightarrow$ 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 defintion
        ensure then
            ... -- additionnal postc
        end ;
end -- class
```

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
```

Assertions

Can be inserted anywhere in the code.

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

Loop (in)variants

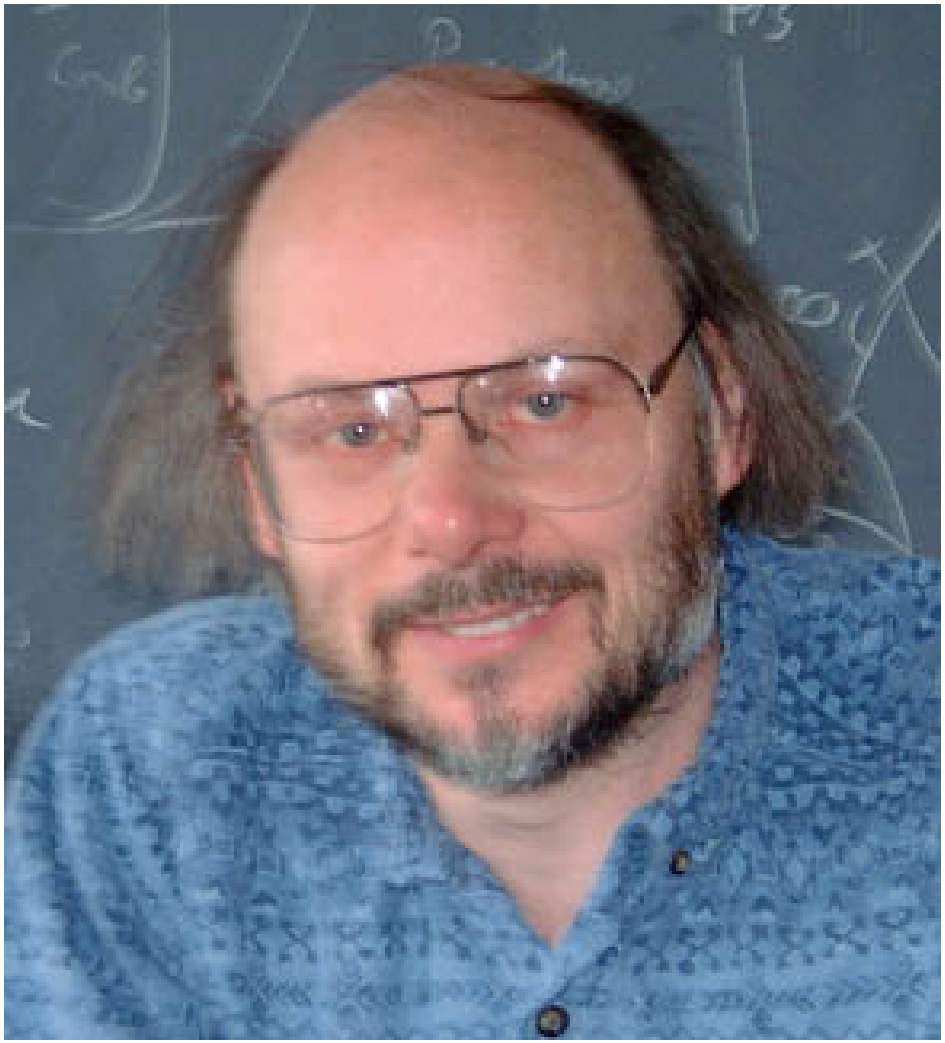
Only one (complex) kind of loop in Eiffel

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

Table of Contents

Bjarne Stroustrup

Bjarne Stroustrup



Bjarne Stroustrup

Bjarne Stroustrup

Bjarne Stroustrup

C++

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

A better and 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.

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
  candidates are:
  bool _List_iterator<int,int &,int *>::operator !=
    (const _List_iterator<int,int &,int *> &) const
```

Table of Contents

CLOS

Developed in mid 80's.

Overview:

- Metaobject Protocol
- Meta-class
- Multiple Inheritance
- Multiple dispatch
- Generic Functions
- Method Qualifier
- Introspection

Small Example

```
(defclass human () (name size birth-year))
(make-instance 'human)
```

```
(defclass Shape () ())
(defclass Rectangle (Shape) ())
(defclass Ellipse (Shape) ())
(defclass Triangle (Shape) ())

(defmethod intersect ((r Rectangle) (e Ellipse))
    ...)

(defmethod intersect ((r1 Rectangle) (r2 Rectangle))
    ...)

(defmethod intersect ((r Rectangle) (s Shape))
    ...)
```

Luca Cardelli

Engineering Properties, 1996

- **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 that 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

Table of Contents

Problem Statement

Traditional class-based OO languages are based on a deep-rooted duality:

- **Classes**: defines behaviours of objects.
- **Object instances**: specific manifestations of a class

Unless one can predict with certainty what qualities a set of objects and classes will have in the distant future, one cannot design a class hierarchy properly

Self

Invented by David Ungar and Randall B. Smith in 1986 at Xerox Park

Overview:

- Neither classes nor meta-classes
- Self objects are a collection of *slots*. Slots are accessor methods that return values.
- Self object is a stand-alone entity
- An object can delegate any message it does not understand itself to the parent object
- Inspired from Smalltalks blocks for flow control
- Generational garbage collector

Example in self

- Copy object lecture and set fill title to TYLA

```
tyla := lecture copy title: 'TYLA'.
```

- add slot to an object

```
tyla _AddSlots: (| remote <- 'true'|).
```

- Modifies at runtime the parent

```
myObject parent: someOtherObject.
```

Impacts

- Javascript
- NewtonScript
- Io
- Rust
- Go

Table of Contents

Rust, Go, ...

Gang of 4 quote

Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called methods or operations.

–

Elements of Reusable Object-Oriented Software
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

Rust's documentation

Even though structs and enums with methods arent called objects, they provide the same functionality, according to the Gang of Fours definition of objects.

Example in Rust

```
trait Foo {
  fn method(&self) -> String;
}

impl Foo for u8 {
  fn method(&self) -> String
      { format!("u8: {}", *self) }
}

impl Foo for String {
  fn method(&self) -> String
      { format!("string: {}", *self) }
}

fn do_something<T: Foo>(x: T) {
    x.method();
}
```

Duck Typing

If it walks like a duck and it quacks like a duck,
then it must be a duck

Deffered to a later lecture (about Genericity)