Programmation Objet : Concepts Avancés
Cours introductif

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

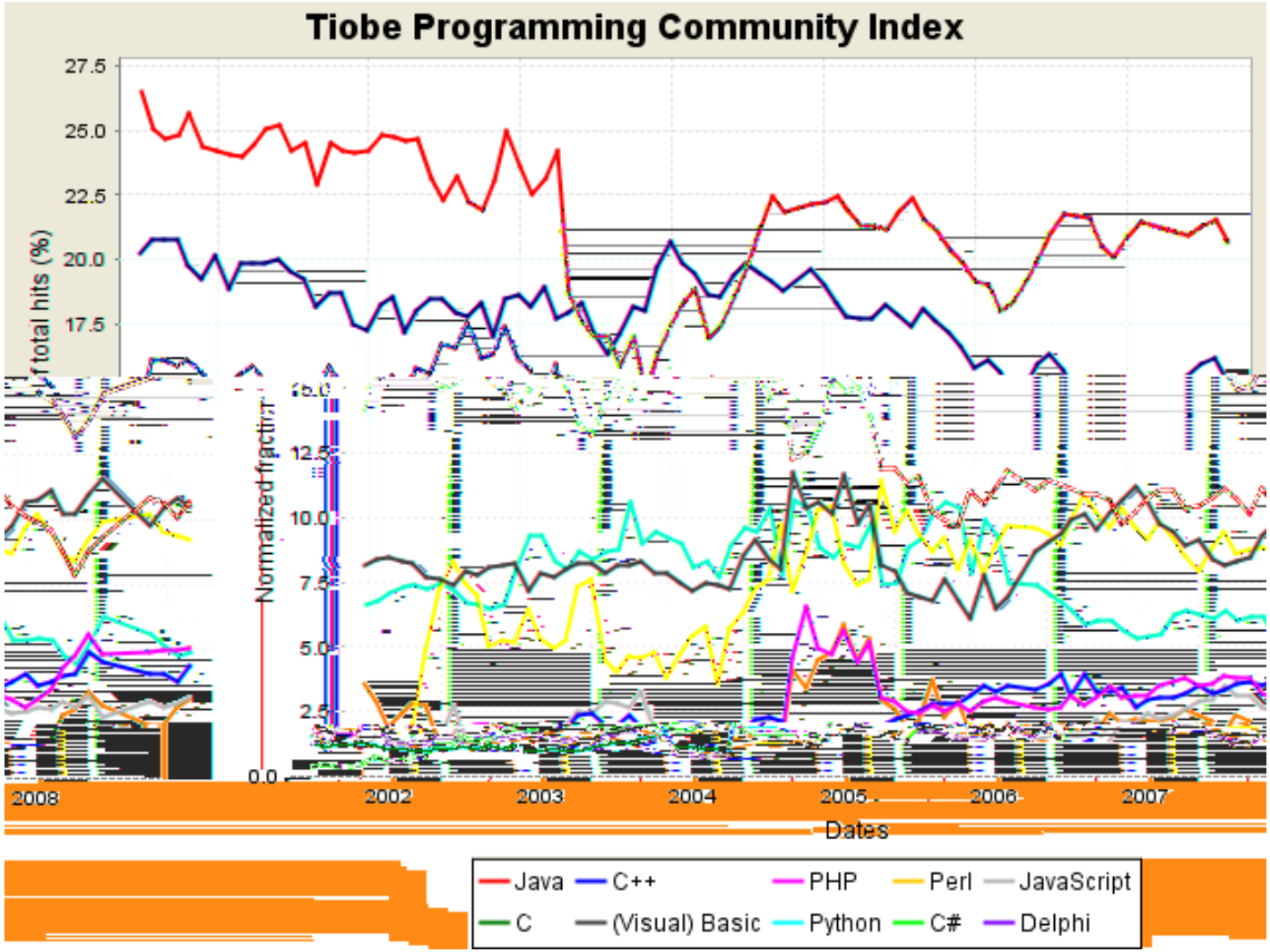
3 octobre 2008

Programmation Objet ?

Position 09/08	Position 09/07	Langage	Score	Evolution	Classe
1	1	Java	20.715%	-0.99%	A
2	2	C	15.379%	+0.47%	A
3	5	C++	10.716%	+0.78%	A
4	3	(Visual) Basic	10.490%	-0.26%	A
5	4	PHP	9.243%	-0.96%	A
6	8	Python	5.012%	+1.99%	A
7	6	Perl	4.841%	-0.58%	A
8	7	C#	4.334%	+0.75%	A
9	9	JavaScript	3.130%	+0.41%	A
10	14	Delphi	3.055%	+1.83%	A
11	10	Ruby	2.762%	+0.70%	A
12	13	D	1.265%	-0.11%	A
13	11	PL/SQL	0.700%	-1.16%	A–
14	12	SAS	0.640%	-0.76%	B
15	23	ActionScript	0.472%	+0.07%	B
16	16	Lisp/Scheme	0.419%	-0.21%	B
17	18	Lua	0.415%	-0.16%	B
18	22	Pascal	0.400%	-0.03%	B
19	-	PowerShell	0.384%	0.00%	B
20	17	COBOL	0.360%	-0.27%	B

source : `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

Programmation Objet : des pronostics ?



source : `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

Programmation Objet : Concepts Avancés

Objectifs :

- Fournir les outils conceptuels pour appréhender :
  - la P.O.O. dans sa globalité (indépendamment du langage) ;
  - les problèmes intrinsèques à la programmation par objets ;
  - les outils linguistiques et de génie logiciel pour les contourner.
- "Apprendre à apprendre" un nouveau langage de programmation.
- S'initier au développement de composants logiciels réutilisables.

Connaissez-vous JAVA ?

Qu'a che ce programme ?

```
int j = 5;
int i = 0;
i = j << 32;
println (i);
Integer a = new Integer (5);
Integer b = new Integer (5);
if (a == b) println ("1");
a++;
b++;
if (a == b) println ("2");
a = 317;
b = 317;
if (a == b) println ("3");
```

Connaissez-vous JAVA ?

Qu'a che ce programme ?

```
int j = 5;
int i = 0;
i = j << 32;
println (i); // i = 5
Integer a = new Integer (5);
Integer b = new Integer (5);
if (a == b) println ("1");
a++;
b++;
if (a == b) println ("2");
a = 317;
b = 317;
if (a == b) println ("3");
```

Connaissez-vous JAVA ?

Qu'a che ce programme ?

```
int j = 5;
int i = 0;
i = j << 32;
println (i); // i = 5
Integer a = new Integer (5);
Integer b = new Integer (5);
if (a == b) println ("1"); // a != b
a++;
b++;
if (a == b) println ("2");
a = 317;
b = 317;
if (a == b) println ("3");
```

Connaissez-vous JAVA ?

Qu'a che ce programme ?

```
int j = 5;
int i = 0;
i = j << 32;
println (i); // i = 5
Integer a = new Integer (5);
Integer b = new Integer (5);
if (a == b) println ("1"); // a != b
a++;
b++;
if (a == b) println ("2"); // a == b
a = 317;
b = 317;
if (a == b) println ("3");
```

Connaissez-vous JAVA ?

Qu'a che ce programme ?

```
int j = 5;
int i = 0;
i = j << 32;
println (i); // i = 5
Integer a = new Integer (5);
Integer b = new Integer (5);
if (a == b) println ("1"); // a != b
a++;
b++;
if (a == b) println ("2"); // a == b
a = 317;
b = 317;
if (a == b) println ("3"); // a != b
```

Connaissez-vous Java ?

Étonnant, non ?

```
int j = 5;
int i = 0;
i = j << 32;
println (i); // i = 5
Integer a = new Integer (5);
Integer b = new Integer (5);
if (a == b) println ("1"); // a != b
a++;
b++;
if (a == b) println ("2"); // a == b
a = 317;
b = 317;
if (a == b) println ("3"); // a != b
```

Pourtant . . .

Qu'a che ce programme ?

```
abstract class Shape {
   void show () {
      System.out.println ("Hello, I am " + this.what ()) ;
   }
   abstract String what () ;
}

class Square extends Shape {
   String what () { return ("a square.") ; }
}

class Circle extends Shape {
   String what () { return ("a circle.") ; }
}

class TestShape {
   public static void main (String[] args) {
      Shape c = new Circle () ;
      Shape s = new Square () ;
      c.show () ;
      s.show () ;
   }
}
```

Pourtant . . .

*Et celui-ci (*PYTHON*) ?*

```
class shape:
    def show (self):
        print 'Hello, I am ' + self.what ()
    def what (self):
        print 'Error! I am an abstract class!'
        exit (1)

class square (shape):
    def what (self): return 'a square.'

class circle (shape):
    def what (self): return 'a circle.'

c = circle ()
s = square ()
c.show ()
s.show ()
```

Pourtant . . .

Et celui-là (C++) ?

```
#include <iostream>
#include <string>

class Shape {
public:
   void show () { std::cout << "Hello, I am " + this->what () << std::endl; }
   virtual std::string what () = 0;
};

class Square : public Shape {
public: virtual std::string what () { return ("a square."); }
};

class Circle : public Shape {
public: virtual std::string what () { return ("a circle."); }
};

int main (int argc, const char** argv) {
   Shape* c = new Circle ();
   Shape* s = new Square ();
   c->show ();
   s->show ();
}
```

Pourtant . . .

Et encore celui-ci (SCALA) ?

```
abstract class Shape {
   def show = println ("Hello, I am " + this.what)
   def what : String
}

class Circle extends Shape {
   def what = "a circle."
}

class Square extends Shape {
   def what = "a square."
}

object testShape {
   def main(args: Array[String]) {
      val c = new Circle
      val s = new Square
      c.show
      s.show
   }
}
```

Pourtant . . .

Et puis celui-là (C#) ?

```
using System ;

abstract class Shape {
   public void show() { Console.WriteLine ("Hello, I am " + this.what ()) ; }
   public abstract string what() ;
}

class Square : Shape {
   public override string what() { return "a square." ; }
}

class Circle : Shape {
   public override string what() { return "a circle." ; }
}

class TestShape {
   static void Main() {
      Shape c = new Circle () ;
      Shape s = new Square () ;
      c.show () ;
      s.show () ;
   }
}
```

Pourtant . . .

Et ce dernier (O'Caml) ?

```
class virtual shape = object (self)
  method show = printendline ("Hello, I am " ^ self#what)
  method virtual what : string
end
class square = object
  inherit shape
  method what = "a square."
end
class circle = object
  inherit shape
  method what = "a circle."
end;;
begin
  let c = new circle in
  let s = new square in
    c#show;
    s#show
end
```

La théorie des langages de programmation

Objectifs

- Classifier ces langages : leurs points communs, leurs différences.
- Se doter d'une connaissance des mécanismes fondamentaux liés à la programmation par objets.

La théorie des langages de programmation

Objectifs

- Classifier ces langages : leurs points communs, leurs différences.
- Se doter d'une connaissance des mécanismes fondamentaux liés à la programmation par objets.

Moyens

- Idéalement : des systèmes formels (des définitions mathématiques).
- Dans ce cours, nous raisonnerons à l'aide :
  - d'exemples de problèmes, exprimés dans un des langages précédents, dans un langage fictif ou dans la notation semi-formelle UML.
  - de solutions propres à un langage donné ou exprimées sous la forme de *design patterns*.
  - en programmant en travaux dirigés et dans votre projet.

Organisation du cours

- 11 séances de cours.
- 4 séances de TD sur machine.
- Évaluation du module :
  - 2/3 de la note finale : un projet
    - fait en binôme ;
    - commencé lors du TP2 ;
    - en deux temps, pour vous faire travailler l'extensibilité.
  - 1/3 de la note finale : le TP 4 noté.
- Pré-requis du cours :
  - Programmation objet en JAVA.
  - Savoir utiliser les outils suivants :
    - EMACS (ou un éditeur équivalent) ;
    - MAKE ;
    - SUBVERSION.

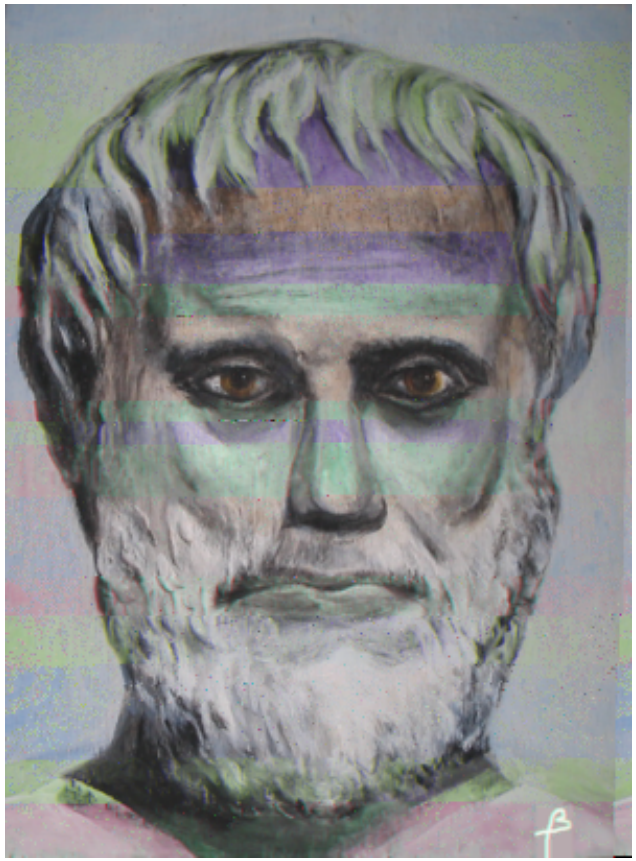
Bibliographie

- *Object-Oriented Software Construction (Broché)* de Bertrand Meyer
- *Langages et modèles à objets*. État des recherches et perspectives Ouvrage coordonné par Roland Ducournau, Jérôme Euzenat, Gérald Masini et Amedeo Napoli issu des cours du CIMPA Collection didactique #19 de l'INRIA
- *A theory of objects* (Springer) de Luca Cardelli et Martin Abadi
- *Object Oriented Programming a Unified Foundation* (Birkhaüser) de Giuseppe Castagna.

I. Concepts fondamentaux

Qu'est-ce qu'un objet ?

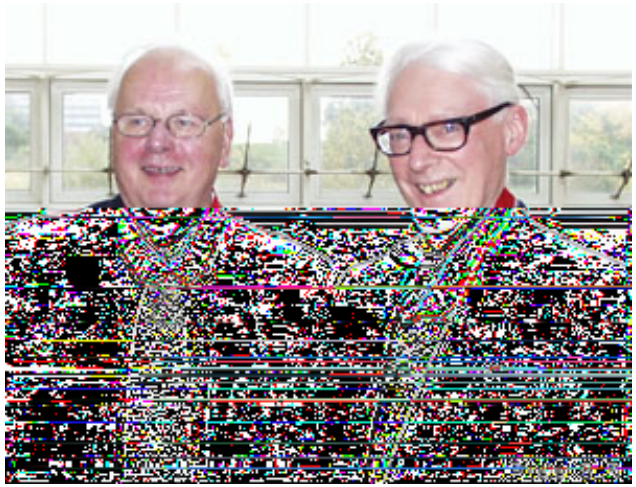
« Objet $\neq$ Concept »



Aristote

Qu'est-ce qu'un objet ?

« Entité autonome réagissant à des messages »



Ole-Johan Dahl
et
Kristen Nygaard

Qu'est-ce qu'un objet ?

Une définition vague . . .

Programmons !

```
(* Un objet représentant un entier borné. *)
class bounded_integer =
   attribute bound : integer
   attribute x : integer
   message get = x
   message set x' = if x' ≥ bound then x ← bound else x ← x'
end

(* Une fonction incrémentant un entier. *)
function incr i = i.set (i.get + 1)

(* L'objet suivant ne pourra être incrémenté que 10 fois. *)
let i_10 = new bounded_integer { bound = 10 ; x = 0 }
```

Quelle différence avec les enregistrements ?

```
(* Un enregistrement stockant un couple d'entiers. *)
record integer pair =
  field x : integer
  field y : integer
end

(* Accéder à l'entier sous-jacent. *)
function get p = p.x

(* Modifier l'entier sous-jacent, en respectant la borne. *)
function set p x' = if x' ≥ p.y then p.x ← p.y else p.x ← x'

(* Création d'un entier borné. *)
function mk_bounded_integer bound init = { x = init ; y = bound }
```

Des enregistrements trop permissifs ?

Que pensez-vous de ce programme ?

```
function cheater client p my_bound =
   p.y ← my_bound
```

Des enregistrements trop permissifs ?

Que pensez-vous de ce programme ?

- Il casse l'abstraction ! Le résultat de la fonction mk_bounded_integer ne reste pas toujours un entier borné.
- Il faut vérifier en tout point du programme que le champ "y" n'est pas modifié.
- Si le programme est complexe, cela peut demander une énergie considérable.
- Au contraire, on veut circonscrire le maintien des invariants du programme à un niveau local (une unité de programmation).

Des enregistrements trop permissifs ?

Que pensez-vous de ce programme ?

- Dans ce but, l'approche objet tend à proscrire les modifications directes de l'état. (À l'exception des langages de la famille de CLOS sur laquelle nous reviendrons.)
- L'état ne peut être modifié qu'à la suite de réceptions de messages.
- On peut ainsi vérifier qu'avant et après la réception du message, l'objet est dans un état cohérent.
- Cette propriété s'appelle l'encapsulation.
- On peut aussi l'obtenir à l'aide de types abstraits (en Ada ou O'Caml par exemple).

Des enregistrements trop permissifs ?

En O'CAML

```
(* En O'Caml *)
module BoundedInteger : sig
  type t
  val mk_bounded_integer : int → int → t
  val get : t → int
  val set : t → int → unit
end = struct
  type t = int ref × int
  let mk_bounded_integer bound init = (ref init, bound)
  let get (x, _) = !x
  let set (x, bound) x' = if x' ≥ bound then x := bound else x := x'
end
```

Des enregistrements trop permissifs ?

En ADA

```
– En ADA
package BoundedInteger is
    type T is private;
    function Get (I : T) return Integer;
    procedure Set (I : out T; Y : Integer);
private
    type T is
        record
            Bound : Integer;
            X : Integer;
        end record;
end BoundedInteger;
```

```
package body BoundedInteger is
    function Get (I : T) return Integer is
    begin
      return I.X;
    end Get;
    procedure Set (I : out T; Y : Integer) is
    begin
        I.X := Y;
    end Set;
end BoundedInteger;
```

Premier ra□nement de la définition (encapsulation)

« Un objet est une entité possédant un état modifiable par envoi de messages. »

Un choix cornélien ?

Comment doit-on spécifier le domaine de cette fonction ?

```
(* Une fonction incrémentant un entier. *)
function incr i = i.set (i.get + 1)
```

Proposition :

L'argument "i" doit être exactement bounded_integer.
(domaine réduit à un type d'objet)

Un choix cornélien ?

Comment doit-on spécifier le domaine de cette fonction ?

```
(* Une fonction incrémentant un entier. *)
function incr i = i.set (i.get + 1)
```

Proposition :

L'argument "i" doit accepter les messages get et set
(domaine étendu à un ensemble infini d'objets)

Un choix cornélien ?

Comment doit-on spécifier le domaine de cette fonction ?

```
(* Une fonction incrémentant un entier et vérifiant le non-dépassement. *)
function checked_incr i =
  let previous_x = i.get in
    i.set (previous x + 1) ;
    if i.get = previous x then error ("overflow")
```

Proposition :

L'argument "i" doit être exactement bounded_integer.
(domaine réduit à un type d'objet)

Un choix cornélien ?

Comment doit-on spécifier le domaine de cette fonction ?

```
(* Une fonction incrémentant un entier et vérifiant le non-dépassement. *)
function checked_incr i =
  let previous_x = i.get in
    i.set (previous x + 1) ;
    if i.get = previous x then error ("overflow")
```

Proposition :

L'argument "i" doit se comporter comme bounded_integer.
(domaine étendu à un ensemble infini d'objets)

Un choix cornélien ?

Comment doit-on spécifier le domaine de cette fonction ?

```
(* Une fonction incrémentant un entier et vérifiant le non-dépassement. *)
function checked_incr i =
  let previous_x = i.get in
    i.set (previous x + 1);
    if i.get = previous x then error ("overflow")
```

Proposition :

C'est trop compliqué ! Donnons une spécification informelle (commentaires) et faisons confiance au programmeur . . .

Un choix cornélien ?

Comment doit-on spécifier le domaine de cette fonction ?

Proposition :

> *C'est trop compliqué ! Donnons une spécification informelle (commentaires) et faisons confiance au programmeur . . .*

- Approche non typée (aucune vérification).
- Approche typée dynamiquement (vérification à l'exécution).

Un choix cornélien ?

Comment doit-on spécifier le domaine de cette fonction ?

Proposition :

C'est trop compliqué ! Le programmeur va faire des erreurs !

Un choix cornélien ?

Comment doit-on spécifier le domaine de cette fonction ?

Proposition :

C'est trop compliqué ! Le programmeur va faire des erreurs !

- Approche typée ou par analyse statique.
- S'appuie sur une description calculable de la forme des objets, un type.
- Un algorithme décide si un programme est respectueux des spécifications.

$\Rightarrow$ Comment déterminer si deux spécifications sont compatibles ?

Le principe de subsomption

- « La spécification S est compatible avec la spécification S'. », si . . . ?

Le principe de subsomption

- « La spécification S est compatible avec la spécification S'. », si . . . ?
- Idéalement : dans tout contexte où un objet de spécification S' est attendu, un objet de spécification S se comporte « aussi bien ».

$\Rightarrow$ Réutilisabilité

Le principe de subsomption

- « La spécification S est compatible avec la spécification S'. », si . . . ?
- Idéalement : dans tout contexte où un objet de spécification S' est attendu, un objet de spécification S se comporte « aussi bien ».

$\Rightarrow$ Réutilisabilité

- Dans l'exemple de la fonction checked_incr, idéalement, on écrirait :

> *« l'objet accepte les messages `get` et `set`. Le message `get` répond l'entier représenté par l'objet. Le message `set` x (où x est un entier) modifie l'état de l'objet : après réception du message, l'objet représente l'entier x sauf si x est plus grand que la borne de l'objet. Dans ce cas, l'objet représente alors cette borne. »*

Le principe de subsomption

- « La spécification S est compatible avec la spécification S'. », si . . . ?
- Idéalement : dans tout contexte où un objet de spécification S' est attendu, un objet de spécification S se comporte « aussi bien ».

⇒ Réutilisabilité

- Dans l'exemple de la fonction checked_incr, idéalement, on écrirait :

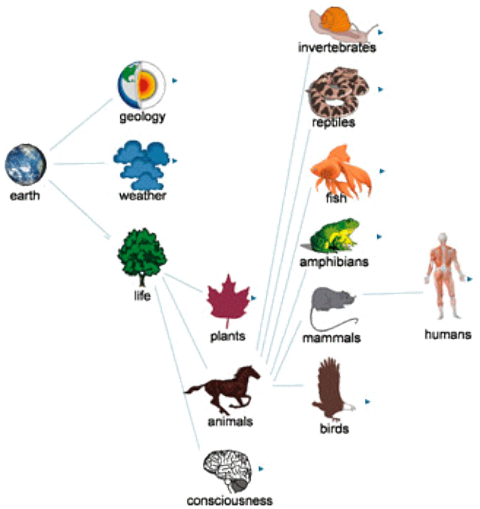
> *« l'objet accepte les messages `get` et `set`. Le message `get` répond l'entier représenté par l'objet. Le message `set`*

Dernier ra□nement de la définition

Un objet est une entité logicielle réutilisable composée d'un état (le plus souvent opaque) et ayant un comportement propre.

- Cette définition n'est toujours pas formelle.
- Elle s'approche cependant de la réalité du paradigme objet.

Taxinomie des langages à objets



Comment définir les objets, leur compatibilité et leur comportement ?

- Avec des classes :
  - Vérifiés par un algorithme de typage statique :
    - basé sur du sous-typage nominal (avec héritage multiple ou non) (C#, C++, Java, ...) ;
    - basé sur du sous-typage structurel (O'Caml, ...).
  - Non vérifiés ou vérifiés dynamiquement.
- Sans classes :
  - à base de prototypes (JavaScript, Objective-C, ...) ;
  - à base de multi-méthodes (CLOS, Nice, ...).

Conclusion de cette partie

- Une définition informelle de la notion d'objet en informatique.

  Un objet est une entité logicielle réutilisable composée d'un état (le plus souvent opaque) et ayant un comportement propre.

- Une classification des langages à explorer.

II. Critique de la P.O.O.

Trois grands problèmes des langages à objet

Nous nous plaçons dans un langage à objet orienté classe (type JAVA).

1. L'extensibilité fonctionnelle.
2. Les opérations n-aires.
3. Raisonner sur les objets.

L'extensibilité

- L'héritage est un mécanisme permettant :
  - la réutilisation de code ;
  - la classification en hiérarchies ;
  - l'extensibilité de l'existant.
- Intéressons-nous de plus près à l'extensibilité d'un système objet.

L'extensibilité des données dans un monde à objets

- Dans un monde à objets, il est très simple d'intégrer au système un nouveau type de données, sans être intrusif, c'est-à-dire sans modification du code existant.

```
class shape =
   message what : string
   message show : string =
      "Hello, I am " + this.what
end
class square =
   inherits shape
   message what : string = "a square."
end
class circle =
   inherits shape
   message what : string = "a circle."
end
```

Bibliothèque

```
class rectangle =
   message what : string = "a rectangle."
end
```

Code client

L'extensibilité des données dans un monde à modules

- Dans un monde à modules, intégrer un nouveau type de données à du code existant est compliqué.

```
module A = struct

  type t = Square | Circle

  let show x =
    ¨Hello, I am ¨ ^ what x

  let what = function
    | Square → ¨a square.¨
    | Circle → ¨a circle.¨

end
```

Bibliothèque

Code client

L'extensibilité des données dans un monde à modules

- Dans un monde à modules, intégrer un nouveau type de données à du code existant est compliqué.

```
module A = struct

  type t = [ `Square | `Circle ]

  let what = function
    | `Square → "a square."
    | `Circle → "a circle."

  let opened_show what x = "Hello, I am "
^ what x

  let show x = opened_show what x

end
```

Bibliothèque

```
module B = struct

  type t = [ `Square | `Circle | `Rectangle ]

  let what = function
    | #A.t as x → A.what x
    | `Rectangle → "a rectangle."

  let show x = A.opened_show what x

end
```

Code client

L'extensibilité des fonctions dans un monde à objets

- Si on veut étendre une hiérarchie d'objets pour lui faire accepter un nouveau message, un gros travail est nécessaire.
- Comment rajouter un message « details : string » commun à tous les objets de la hiérarchie suivante ?

```
class shape =
   message what : string
   message show : string =
      "Hello, I am " + this.what
end
class square =
   inherits shape
   message what : string = "a square."
end
class circle =
   inherits shape
   message what : string = "a circle."
end
```

Bibliothèque

Code client

L'extensibilité des fonctions dans un monde à objets

- Si on veut étendre une hiérarchie d'objets pour lui faire accepter un nouveau message, un gros travail est nécessaire.
- Comment rajouter un message « details : string » commun à tous les objets de la hiérarchie suivante ?

```
class shape =
  message what : string
  message show : string =
    "Hello, I am " + this.what
end
class square =
  inherits shape
  message what : string = "a square."
end
class circle =
  inherits shape
  message what : string = "a circle."
end
```

Bibliothèque

```
class shape_bis =
  inherits shape
  message details : string
end
class square_bis =
  inherits shape_bis, square
  message details =
    "I am a very nice square."
end
class circle_bis =
  inherits shape_bis, square
  message details =
    "I am an ugly circle."
end
```

Code client

Un premier *design pattern* à la rescousse

- Pour les langages ne fournissant pas d'héritage multiple, il faut trouver une autre solution pour contourner le problème.
- Voici une solution qui utilise un objet appelé visitor.

```
predeclared class visitor
class shape =
  ...
  message accept : visitor → unit
end
class square =
  ...
  message accept (v : visitor) = v.visit_square this
end
class circle =
  ...
  message accept (v : visitor) = v.visit_circle this
end
class visitor =
  message visit (s : shape) : unit = s.accept this
  message visit_square : square → unit
  message visit_circle : circle → unit
end
```

Bibliothèque

```
class details_visitor =
  inherits visitor
  message visit_square (s : square) =
    ¨I am a very nice square¨
  message visit_circle (s : circle) =
    ¨I am an ugly circle¨
end
class with_details =
  attribute shape : shape
  message what = shape.what
  message show = shape.show
  message details =
    shape.accept (new details_visitor)
end
```

Code client

Un premier *design pattern* à la rescousse

- Cette méthode pour permettre l'extensibilité fonctionnelle d'une hiérarchie d'objets peut être réutilisée.
- C'est une sorte de « recette de cuisine », qu'on appelle *design pattern* ou patrons de conception.
- Nous verrons d'autres *design patterns* tout au long de ce cours.

La solution en SCALA

- Le langage SCALA intègre un filtrage par motifs sur les classes qui permet de traiter encore plus élégamment le problème précédent.

```
case class WithDetails (shape : Shape) {
  def what = shape.what
  def show = shape.show
  def details =
    shape match {
      case Square () => "I am a very nice square."
      case Circle () => "I am an ugly square."
    }
}

object testShape {
  def main(args: Array[String]) {
    val c = WithDetails (Circle ())
    val s = WithDetails (Square ())
    println (c.details)
    println (s.details)
  }
}
```

Un message à destinataires multiples

- Certains messages concernent plusieurs objets simultanément.
- Exemple : un message servant à savoir si deux objets sont égaux.
- Dans un cadre typé, c'est le type de l'objet qui détermine la réaction à un message n'ayant qu'un seul destinataire.
- Plus généralement, c'est le n-uplet des types des destinataires d'un message qui doit déterminer la réaction du système.

Un message à destinataires multiples

- Comment définir un message à destinataires multiples ?
- Par exemple, un message servant à savoir si deux objets sont égaux ?

```
class integer =
  attribute x : int
  message get : int = x
  message set (x' : int) = x ← x'
end
class bounded_integer =
  inherits integer
  attribute bound : int
  message set (x' : int) = if x' ≥ bound then x ← bound else x ← x'
end
```

Une première tentative

```
class integer =
  attribute x : int
  message get : int = x
  message set (x' : int) = x ← x'
  message equal (i : integer) = (x = i.x)
end
class bounded_integer =
  inherits integer
  attribute bound : int
  message set (x' : int) =
    if x' ≥ bound then x ← bound else x ← x'
  message equal (i : bounded_integer) =
    (x = i.x ∧ bound = i.bound)
end
```

- Que pensez-vous de ces objets ?

Une première tentative

```
class integer =
  attribute x : int
  message get : int = x
  message set (x' : int) = x ← x'
  message equal (i : integer) = (x = i.x)
end
class bounded_integer =
  inherits integer
  attribute bound : int
  message set (x' : int) =
    if x' ≥ bound then x ← bound else x ← x'
  message equal (i : bounded_integer) =
    (x = i.x ∧ bound = i.bound)
end
```

```
function will_explode (i : integer) =
  let i' = new integer { x = 1 } in
    i.equal (i')
```

- Que pensez-vous de ces objets ?

Une seconde tentative

```
class integer =
  attribute x : int
  message get : int = x
  message set (x' : int) = x ← x'
  message equal (i : integer) = (x = i.x)
end
class bounded_integer =
  inherits integer
  attribute bound : int
  message set (x': int) =
    if x' ≥ bound then x ← bound else x ← x'
  message equal (i : integer) =
    if i instanceOf bounded_integer then
      let bi = cast<bounded_integer>(i) in
        (x = i.x ∧ bound = i.bound)
    else
      false
end
```

```
function will_explode (i : integer) =
  let i' = new integer { x = 1 } in
    i'.equal (i)
```

```
function explosion () =
  let i = new bounded_integer
            { x = 42, bounded = 51 }
  in
   will_explode i
```

- Vérifiez que nous avons réparé nos objets !
- Ceci explique les restrictions sur le type des arguments des méthodes de certains langages (JAVA, ...). Nous reviendrons plus en détails sur ces problèmes techniques.

Les multi-méthodes

- Les messages à destinataires multiples, comme le message equal précédent, pose un problème fondamental : si on modélise la réception des messages par des méthodes encapsulées dans une classe donnée, on fixe un destinataire privilégié et on peut ainsi casser la symétrie du message.
- Certains langages de programmation orienté objet ont décidé de faire un pas en arrière vis-à-vis de la notion d'encapsulation (NICE, CLOS, ...). Dans ces langages, on peut définir les méthodes en dehors des classes.
- Le choix de la méthode à appeler dépend de tous les arguments, symétriquement.

```
class Integer { int x; }
class BoundedInteger extends Integer { int bound; }
equals (Integer lhs, Integer rhs) { return lhs.x == rhs.x; }
equals (BoundedInteger lhs, BoundedInteger rhs) { return lhs.x == rhs.x; }
```

Exemple en NICE

Raisonner dans un monde qui change . . .

- En présence d'un état modifiable, l'encapsulation est parfois un leurre.
- En effet, si des données sont partagées par deux objets, un message envoyé au premier peut changer l'état de l'autre.
- Cette situation complique le raisonnement sur le système, censé être facilité par l'encapsulation !

```
Integer[] array = new Integer[3];
array[0] = 42;
MyInteger a1 = new MyInteger (array);
MyInteger a2 = new MyInteger (array);
a1.incr ();
a2.incr ();
System.out.println (a1.get()[0]);
System.out.println (a2.get()[0]);
```

Exemple en JAVA

Raisonner dans un monde qui change . . .

- En présence d'un état modifiable, l'encapsulation est parfois un leurre.
- En effet, si des données sont partagées par deux objets, un message envoyé au premier peut changer l'état de l'autre.
- Cette situation complique le raisonnement sur le système, censé être facilité par l'encapsulation !

```
Integer[] array = new Integer[3];
array[0] = 42;
MyInteger a1 = new MyInteger (array);
MyInteger a2 = new MyInteger (array);
a1.incr ();
a2.incr ();
System.out.println (a1.get()[0]);
System.out.println (a2.get()[0]);
```

```
public class MyInteger {
  private Integer[] x;
  MyInteger (Integer[] y) { x = y; }
  public void incr () { x[0] = x[0] + 1; }
  public Integer[] get () { return x; }
}
// Les deux lignes a□chent 44 !
```

Exemple en JAVA

Contre-mesures !

- Il faut avoir une idée des partages de données : nous verrons des *design patterns* permettant de s'imposer une discipline de propriété (*ownership*) des objets.
- Certains langages, comme C++, introduisent deux contextes d'utilisation des objets à l'aide du mot-clé const. On écrirait alors :

```
class MyInteger {
  private: std::vector<int> _x;
  public: MyInteger (const std::vector<int>& i): _x(i) {}
  public: void incr () { _x[0]++; }
  public: const std::vector<int>& get () const { return _x; }
};

std::vector<int> array(3);
array[0] = 42;
MyInteger a1 (array);
MyInteger a2 (array);
a1.incr ();
a2.incr ();
std::cout << (a1.get ())[0] << << (a2.get ())[0] << std::endl; // a□che 43
```

- . . . ce qui force le tableau _x à être copié au moment de la création de l'instance du type MyInteger.

Aides au raisonnement

- Malheuresement, le mot-clé const est peu expressif.
- O'CAML propose des objets fonctionnels, non modifiables, qui facilitent grandement le raisonnement.

```
#class functional_point y =
    object
      val x = y
      method get_x = x
      method move d = {< x = x + d >}
    end;;
class functional_point :
  int →
  object (α) val x : int method get_x : int
method move : int → α end
```

```
#let p = new functional_point 7;;
val p : functional_point = <obj>

#p#get_x;;
- : int = 7

#(p#move 3)#get_x;;
- : int = 10

#p#get_x;;
- : int = 7
```

*Exemple tiré du manuel d'*O'CAML

Conclusion de cette partie

- Dans cette partie, nous avons introduit 3 problèmes des langages à objet :
  - L'extensibilité fonctionnelle.
  - Les messages à destinataires multiples.
  - Le partage de données.
- Dans la suite de ce cours, nous reviendrons sur ces problèmes en les regardant à la lumière des mécanismes que nous allons explorés.

III. Le langage C++

Le langage C avec des objets ? (source : Wikipedia)

- Le langage C++ a été conçu par Bjarne Stroustrub dans les années 80.
- Le langage C++ est normalisé par l'ISO. Sa première normalisation date de 1998 (ISO/CEI 14882 :1998), sa dernière de 2003 (ISO/CEI 14882 :2003). La normalisation de 1998 standardise la base du langage (Core Language) ainsi que la bibliothèque standard du C++ (C++ Standard Library).

Le langage C avec des objets ? (source : Wikipedia)

- les déclarations reconnues comme instructions (repris dans C99) ;
- les opérateurs new et delete pour la gestion d'allocation mémoire ;
- le type de données bool (booléen) ;
- les références ;
- le mot clé const pour définir des constantes (repris par C à la fin des années 1980) ;
- les fonctions inline (repris dans C99) ;
- les paramètres par défaut dans les fonctions ;
- les référentiels lexicaux (Espace de noms) et l'opérateur de résolution : : ;
- les classes, ainsi que tout ce qui y est lié : l'héritage, les fonctions membres, les fonctions membres virtuelles, les constructeurs et le destructeur ;
- la surcharge des opérateurs ;
- les templates ;
- la gestion d'exceptions ;
- l'identification de type pendant l'exécution (RTTI : run-time type identification) ;
- le commentaire de fin de ligne introduit par « // » (existant dans BCPL, repris dans C99).

Reprenons ! Plus méthodiquement

- C++ est un langage à objet :
  - Typé statiquement,
  - Dont le langage noyau est très proche de C,
  - Utilisant des classes construites par héritage, éventuellement multiple.
- La notion d'encapsulation est implémentée par un système de protection public, protected, private.
- La réutilisabilité du code existant est implémentée par :
  - un système de classes classiques avec liaison tardive (les méthodes dites virtuelles dans le jargon C++) ;
  - la possibilité d'e□ectuer des appels statiques à des méthodes ;
  - un mécanisme de patrons de classes et de fonctions (les *templates*).

Un exemple de programme en C++, très C

```
#include <cstdio>
int main (int argc, char **argv) {
   printf (¨Hello World !\n¨) ;
}
```

- C++est 99% compatible avec C.

Un exemple de programme en C++, moins C

```
#include <iostream>
#include <string>

int main (int argc, char** argv) {
   std::string msg = "Hello";
   msg += " World !";
   std::cout << msg << std::endl;
}
```

- Les espaces de noms, un outil de structuration.
- std::string est une classe fournie par la bibliothèque standard.
- C++ introduit la surcharge des opérateurs. msg += ¨ World !¨ est un appel de méthode.

Notre propre classe pour les chaînes de caractères

```
#include <cstring>
class mystring {
   private: const char* _str;
   public: mystring (const char* str) : _str (strdup (str)) {}
   public:
   void append (const char* str) { // ... }
   void operator+= (const char* str) {
      this→append (str);
   }
};
```

- Une définition de classe en C++se termine par un point-virgule.
- La spécification de protection (private, public, protected) s'exprime par blocs.
- L'initialisation des attributs se fait à l'aide d'une syntaxe spéciale.
- this est de type mystring*.

Notre classe, entre une interface et une implémentation

```
#ifndef MYSTRING_HH
# define MYSTRING_HH
#include <cstring>

class mystring {
   private: const char* _str;
   public: mystring (const char* str);
   public:
   void append (const char* str);
   void operator+= (const char* str);
};
#endif // MYSTRING_HH
```

mystring.hh

```
#include ¨mystring.hh¨
mystring::mystring (const char* str) :
   _str (strdup (str))
{}

void mystring::append (const char* str) {
   // ...
}

void mystring::operator+= (const char* str)
{
      this→append (str);
}
```

mystring.cc