Cours Introductif

Cours : Gustavo Petri

TD : Aldric Degorre,
Adrien Pommellet

Slides : Yann Régis-Gianas

Université Denis Diderot – Paris 7

Plan

Plan

POCA : un cours de programmation objet **avancée** ?

- Approfondissement et critique des outils de la programmation objet.

POCA : un cours de programmation objet **avancée** ?

- Approfondissement et critique des outils de la programmation objet.

Approfondissement

- Mieux comprendre les mécanismes connus.
- Découvrir de nouveaux mécanismes
  (*via* le langage SCALA).
- Améliorer ses méthodes de raisonnement sur les programmes objet.
- Concevoir des composants logiciels réutilisables.

POCA : un cours de programmation objet **avancée** ?

- Approfondissement et critique des outils de la programmation objet.

Approfondissement

- Mieux comprendre les mécanismes connus.
- Découvrir de nouveaux mécanismes
  (*via* le langage SCALA).
- Améliorer ses méthodes de raisonnement sur les programmes objet.
- Concevoir des composants logiciels réutilisables.

Critique

- Connaître les limites de la programmation objet.
- Savoir contourner ces problèmes à l'aide :
  - de mécanismes modernes (alternatifs) de programmation ;
  - ou *via* des patrons de conception.

Des experts en JAVA dans la salle ?

Qu'affiche ce programme ?

```
01 int j = 5;
02 int i = 0;
03 i = j << 32;
04 println (i);
05 Integer a = new Integer (5);
06 Integer b = new Integer (5);
07 if (a == b) println ("1");
08 a++;
09 b++;
10 if (a == b) println ("2");
11 a = 317;
12 b = 317;
13 if (a == b) println ("3");
```

Des experts en JAVA dans la salle ?

Qu'affiche ce programme ?

```
01 int j = 5;
02 int i = 0;
03 i = j << 32;
04 println (i); // i = 5
05 Integer a = new Integer (5);
06 Integer b = new Integer (5);
07 if (a == b) println ("1");
08 a++;
09 b++;
10 if (a == b) println ("2");
11 a = 317;
12 b = 317;
13 if (a == b) println ("3");
```

Des experts en JAVA dans la salle ?

Qu'affiche ce programme ?

```
01 int j = 5;
02 int i = 0;
03 i = j << 32;
04 println (i); // i = 5
05 Integer a = new Integer (5);
06 Integer b = new Integer (5);
07 if (a == b) println ("1"); // a != b
08 a++;
09 b++;
10 if (a == b) println ("2");
11 a = 317;
12 b = 317;
13 if (a == b) println ("3");
```

Des experts en JAVA dans la salle ?

Qu'affiche ce programme ?

```
01 int j = 5;
02 int i = 0;
03 i = j << 32;
04 println (i); // i = 5
05 Integer a = new Integer (5);
06 Integer b = new Integer (5);
07 if (a == b) println ("1"); // a != b
08 a++;
09 b++;
10 if (a == b) println ("2"); // a == b
11 a = 317;
12 b = 317;
13 if (a == b) println ("3");
```

Des experts en JAVA dans la salle ?

Qu'affiche ce programme ?

```
01 int j = 5;
02 int i = 0;
03 i = j << 32;
04 println (i); // i = 5
05 Integer a = new Integer (5);
06 Integer b = new Integer (5);
07 if (a == b) println ("1"); // a != b
08 a++;
09 b++;
10 if (a == b) println ("2"); // a == b
11 a = 317;
12 b = 317;
13 if (a == b) println ("3"); // a != b
```

Des experts en JAVA dans la salle ?

ÉTONNANT, NON ?

```
01 int j = 5;
02 int i = 0;
03 i = j << 32;
04 println (i); // i = 5
05 Integer a = new Integer (5);
06 Integer b = new Integer (5);
07 if (a == b) println ("1"); // a != b
08 a++;
09 b++;
10 if (a == b) println ("2"); // a == b
11 a = 317;
12 b = 317;
13 if (a == b) println ("3"); // a != b
```

⇒ La bonne réponse, c'est d'utiliser la méthode ".equals".

Des experts en JAVA dans la salle ?

Qu'affiche ce programme ?

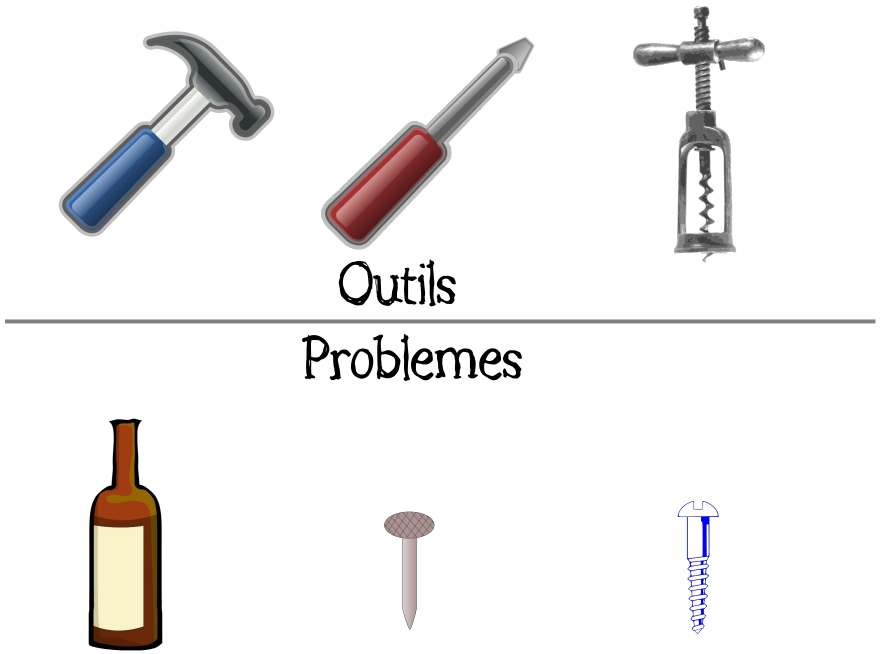
```
01 class Point {
02       protected final int x, y;
03       private final String name;
04       Point(int x, int y) {
05            this.x = x;
06            this.y = y;
07            name = makeName();
08       }
09
10       protected String makeName() {
11            return "[" + x + "," + y + "]";
12       }
13
14       public final String toString() {
15            return name;
16       }
17 }
```

```
01 public class ColorPoint extends Point {
02       private final String color;
03       ColorPoint(int x, int y, String color) {
04            super(x, y);
05            this.color = color;
06       }
07
08       protected String makeName() {
09           return super.makeName()
10                + ": " + color;
11       }
12
13       public static void main(String[] args) {
14            System.out.println
15            (new ColorPoint(4, 2, "purple"));
16       }
17 }
```

Question

Est-ce qu'il faut et il suffit de connaître un langage de programmation dans ses moindres recoins pour devenir un bon programmeur ?

Qu'est-ce que la programmation ?



(Une définition plus sérieuse vient un peu plus loin.)

Il faut bien comprendre l'outil en lui-même **et** son utilité.

Comment progresser en programmation ?

Apprentissage centré sur les mécanismes des langages (objets)

Apprentissage linguistique

1. Comprendre un mécanisme de programmation
2. Raisonner sur des programmes écrits à l'aide de ces mécanismes
3. L'intégrer dans sa boîte à outils

Apprentissage méta-linguistique

- Didactique : Apprendre à apprendre un nouveau langage
- Dialectique : Comprendre ce qui se passe (en comprenant ce qui s'est passé)

Essayons d'appliquer cette méthode.

Plan

Comprendre un mécanisme de programmation

Mécanisme objet 1 :

La liaison tardive

(Un mécanisme que vous connaissez déjà.)

1. Compréhension à l'aide d'exemples
2. Formalisation
3. Pourquoi ce mécanisme a été introduit ? À quoi sert-il ?

Exemple 1 : un afficheur d'arbre

Écrire une fonction `printTree` *qui attend un arbre et un document, et affiche l'arbre dans le document en utilisant la capacité du document à afficher des énumérations.*

(Un premier programme SCALA)

Exemple 1 : un afficheur d'arbre

En SCALA, la syntaxe pour définir une fonction est :

```
01 def printTree (tree : Tree, doc : Document) : Unit =
02   ...
```

Exemple 1 : un afficheur d'arbre

```
01 def printTree (tree : Tree, doc : Document) : Unit =
02   tree match {
03     case Leaf (x) =>
04       doc.printLine (x.toString)
05     case Node (x, children) => {
06       doc.printLine (x.toString)
07       doc.startEnumeration ();
08       children.foreach (child => printTree (child, doc));
09       doc.stopEnumeration ();
10     }
11   }
```

Exemple 1 : un afficheur d'arbre

La fonction `printTree` est définie par cas sur la forme de l'arbre :

```
01 def printTree (tree : Tree, doc : Document) : Unit =
02   tree match {
03     case Leaf (x) => ...
04     case Node (x, children) => ..
05   }
```

(Nous reviendrons sur cette construction "`match`" un peu plus tard.)

Exemple 1 : un afficheur d'arbre

On utilise la capacité de l'objet "`doc`" à afficher une énumération :

```
01 def printTree (tree : Tree, doc : Document) : Unit =
02   tree match {
03     case Leaf (x) =>
04       /* Si "t" est une feuille, on l'affiche. */
05       doc.printItem (x.toString)
06     case Node (x, children) => {
07       /* Si "t" est un nœud, */
08       /* on affiche l'entier sur le nœud. */
09       doc.printItem (x.toString)
10       /* On débute une énumération de ses fils */
11       doc.startEnumeration () ;
12       /* que l'on affiche à l'aide d'un appel récursif. */
13       children.foreach (child => printTree (child, doc)) ;
14       /* et on ferme l'énumération. */
15       doc.stopEnumeration () ;
16     }
17   }
```

Exemple 1 : un afficheur d'arbres

- "`Document`" est un type d'objet que l'on peut définir ainsi :

```
01 type Document = {
02     def printItem (s: String) : Unit
03     def startEnumeration () : Unit
04     def stopEnumeration () : Unit
05 }
```

- On reconnaît les méthodes utilisées par la fonction précédente et leurs signatures respectives.

Exemple 1 : un afficheur d'arbres

- On peut définir deux objets de ce type :

```
01 object ConsoleDocument {
02   var offset = 0
03   def printItem (s: String) = println (mkIndent (offset) + s)
04   def startEnumeration () = offset += 2
05   def stopEnumeration () = offset -= 2
06 }
07
08 object HTMLDocument {
09   def printItem (s: String) = println ("<li>" + s + "</li>")
10   def startEnumeration () = println ("<ul>")
11   def stopEnumeration () = println ("</ul>")
12 }
```

Exemple 1 : un afficheur d'arbres

- Quand on exécute "`printTree(t, ConsoleDocument)`", on obtient un comportement différent de l'appel "`printTree(t, HTMLDocument)`".
- Pourtant, le code de "`printTree`" n'est pas modifié entre ces deux appels.
- C'est l'évaluation des appels de méthode à l'intérieur du code de "`printTree`" qui varie.
- En effet, par exemple, le sens de l'expression "`doc.startEnumeration()`" dépend de l'objet "`doc`". Lorsque "`doc`" vaut "`HTMLDocument`", la fonction appelée est la méthode de l'objet "`HTMLDocument`". Lorsque "`doc`" vaut "`ConsoleDocument`", la fonction appelée est la méthode de l'objet "`ConsoleDocument`".

$\Rightarrow$ La résolution de ces appels de fonctions est donc **dynamique**.

- Autrement dit, le code de `printTree` *est paramétré* par le code des méthodes de l'objet "`doc`".

Quelques définitions

Ordre supérieur

Un fragment de code paramétré par un (ou plusieurs) fragment(s) de code est dit d'ordre supérieur.

(La programmation fonctionnelle et la programmation objet sont d'ordre supérieur.)

Liaison tardive

Le mécanisme de résolution qui utilise à la fois un objet et le nom d'une méthode pour effectuer un appel de fonction s'appelle liaison tardive.

(On dit aussi liaison dynamique, retardée, résolue dynamiquement, ...)

Un mécanisme présent dans de nombreux langages objets

Exercice

1. Donnez l'implémentation de printTree en JAVA, C++ et O'CAML.
2. Quels mécanismes avez-vous utilisé pour simuler l'analyse de motifs ?

Formalisation

- Aujourd'hui, nous n'allons pas rentrer dans les détails de fonctionnement de ce mécanisme important.
- Ce sera le sujet d'un prochain cours.

  (et d'un cours de compilation avancée)

Pourquoi a-t-on introduit un tel mécanisme ?

- Essayons de programmer "`printTree`" en C.
- Les premières définitions de type qui viennent à l'idée sont :

```
01 typedef struct document {
02     void (*printItem) (int);
03     void (*startEnumeration) (void);
04     void (*stopEnumeration) (void);
05 } document_t;
06
07 typedef struct tree {
08     int value;
09     int nb_children;
10     struct tree** child;
11 } tree_t;
```

Pourquoi a-t-on introduit un tel mécanisme ?

- Un état est implicitement partagé par les méthodes d'un objet.
- Une représentation correcte d'un objet en C est donc plutôt :

```
01 typedef struct document {
02     void *state; // État partagé
03     void (*printItem) (int);
04     void (*startEnumeration) (void);
05     void (*stopEnumeration) (void);
06 } document_t;
```

Pourquoi ne pas utiliser ce codage ?

(ou pourquoi être Turing-complet n'est pas suffisant.)

- Ce codage requiert une discipline de programmation importante.
- Pas d'aide du langage : pas de syntaxe, pas de typage, ...
- Il n'est pas compatible avec d'autres mécanismes objets classiques.
- Avec des implémentations moins naïves, on peut s'en sortir (un peu). (cf. GObject)

Un langage d'ordre supérieur, cela sert à quoi ?

Un langage d'ordre supérieur, cela sert à quoi ?

À écrire des programmes **modulaires** !

Un langage d'ordre supérieur, cela sert à quoi ?

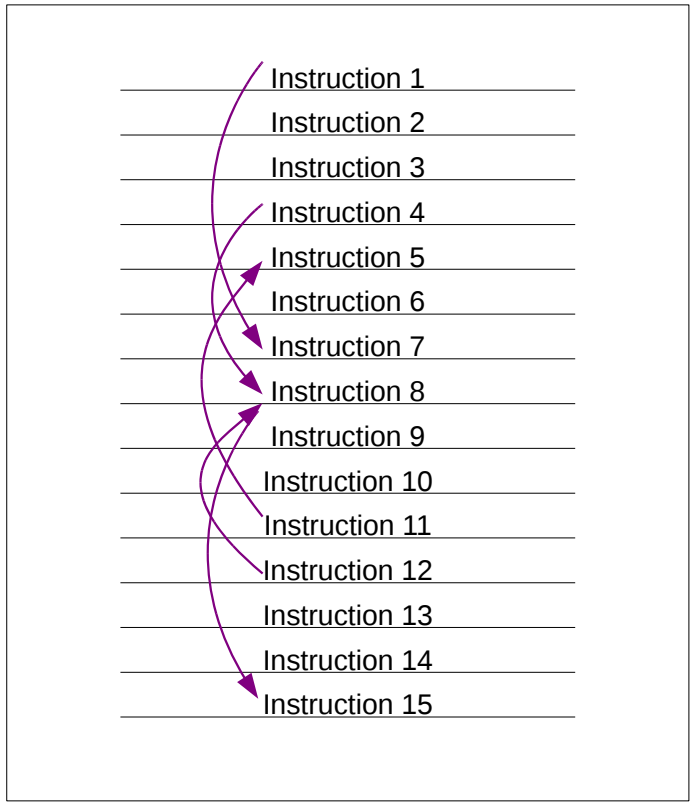
À écrire des programmes **modulaires** !

Modularité

Un programme est construit de façon modulaire si les composants logiciels qui le constituent :

- sont faiblement couplés, *i.e.* leurs interdépendances sont minimales ;
- ont une forte cohésion interne.

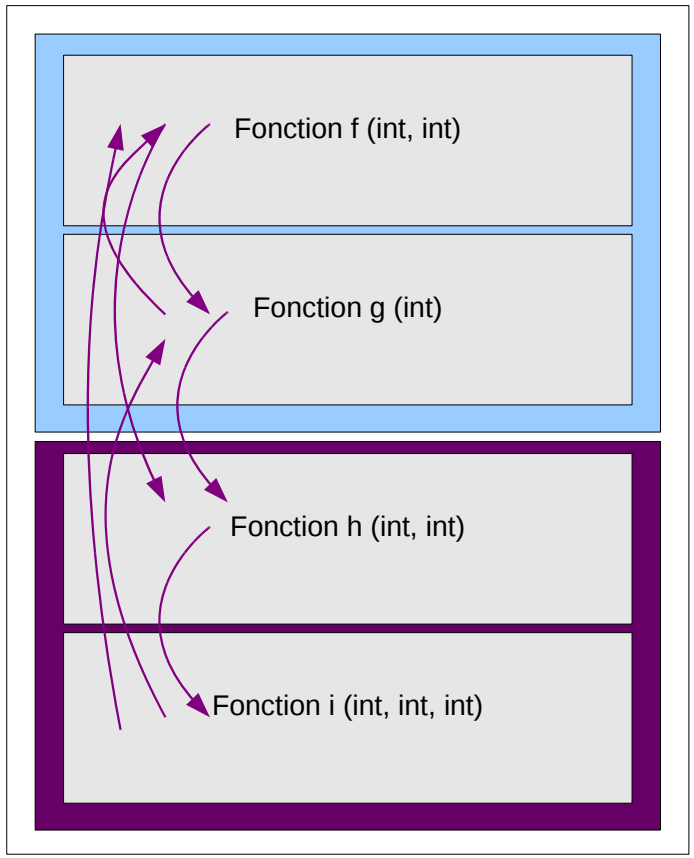
Modularité : Niveau 0 – Programmation non structurée



(Une flèche de A vers B signifie « A fait référence à B »

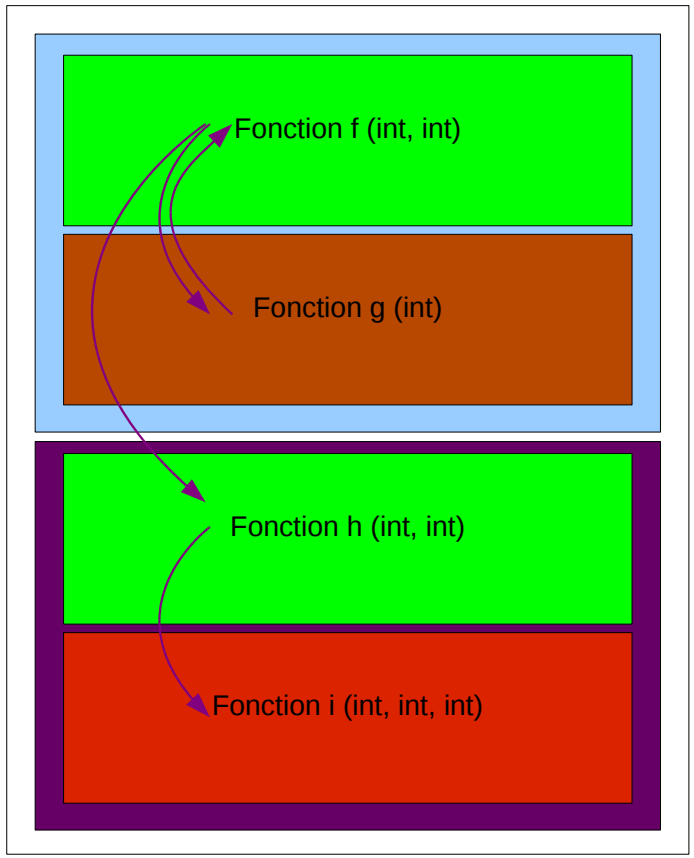
- Évaluer les interdépendances :
  « Quelle quantité de modifications si j'enlève l'instruction 7 ? »
- Évaluer la cohésion :
  « Peut-on expliciter le rôle de chaque instruction indépendante ? »

Modularité : Niveau 1 – Programmation d'ordre 1



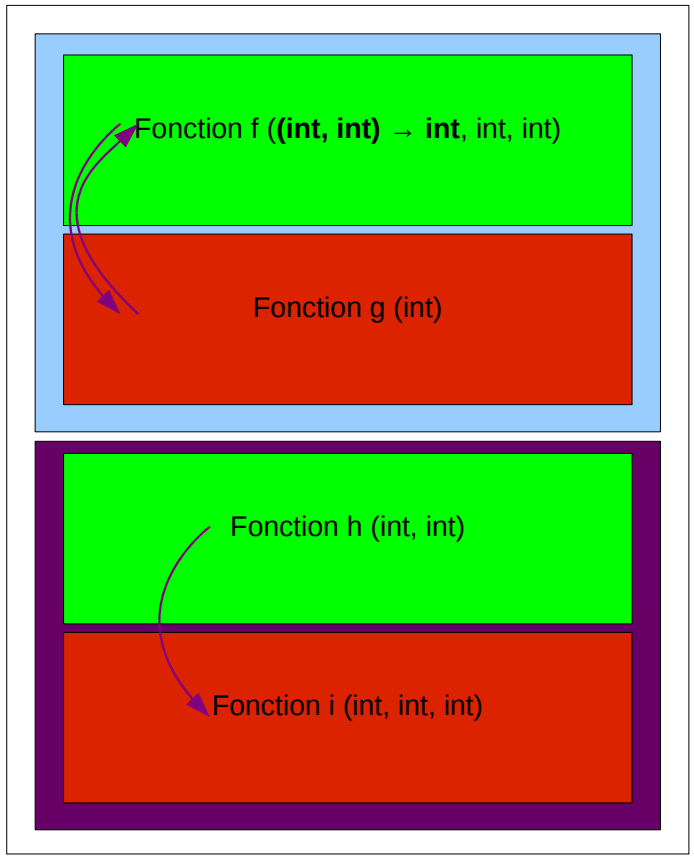
- Évaluer les interdépendances :
  « Quelle quantité de modifications si je change le type de g ? »
- Évaluer la cohésion :
  « Peut-on expliciter le rôle de chaque fonction ? module ? »

Modularité : Niveau 2 – Programmation d'ordre 1



- Évaluer les interdépendances :
  « Quelle quantité de modifications si je change le type de g ? »
- Évaluer la cohésion :
  « Peut-on expliciter le rôle de chaque fonction ? module ? »

Modularité : Niveau 3 – Programmation d'ordre supérieur



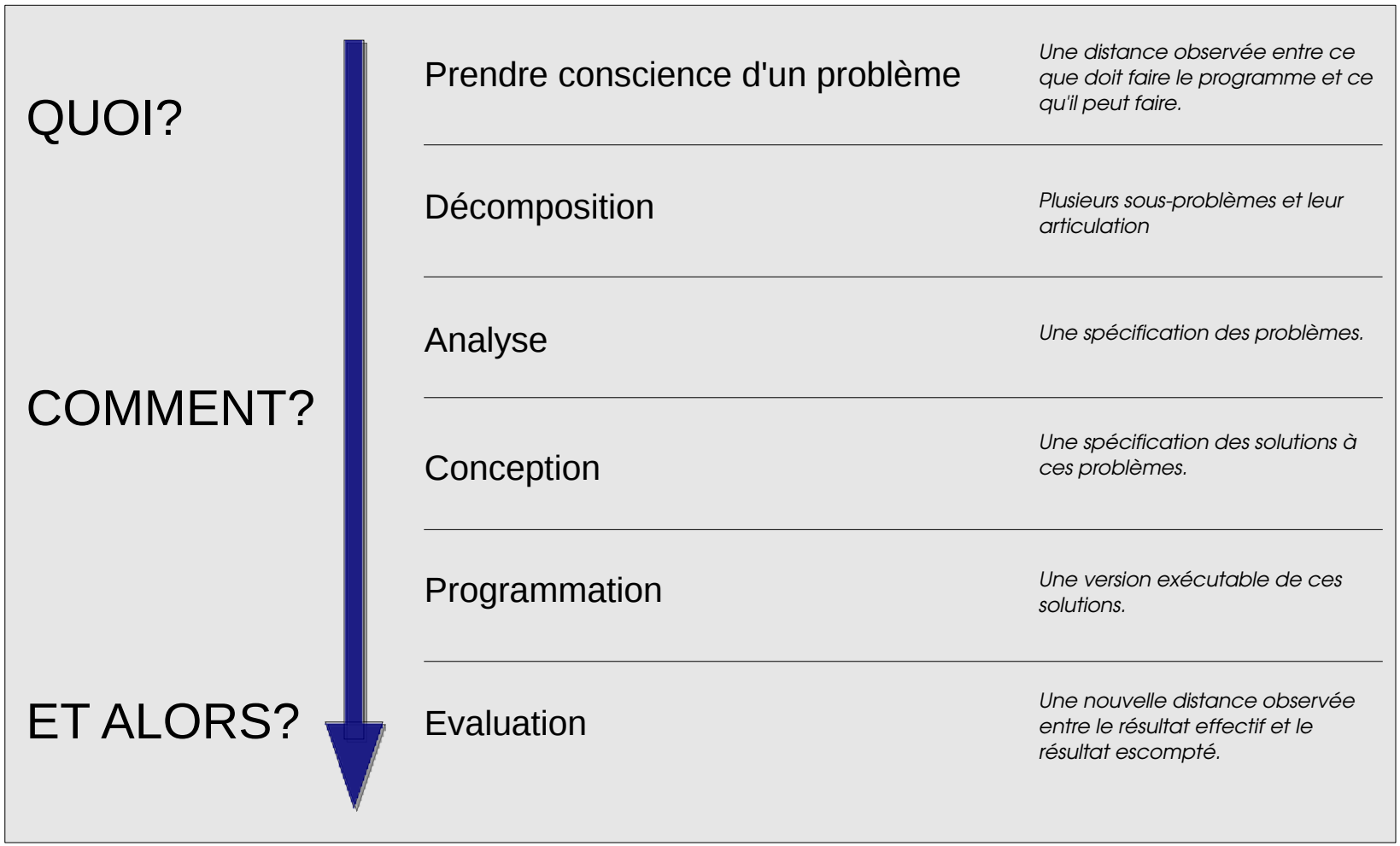
- On dissocie la définition d'un composant de son utilisation.
- Les composants sont indépendants et donc réutilisables.

Quelles sont les différences entre programmation fonctionnelle et programmation objet ?

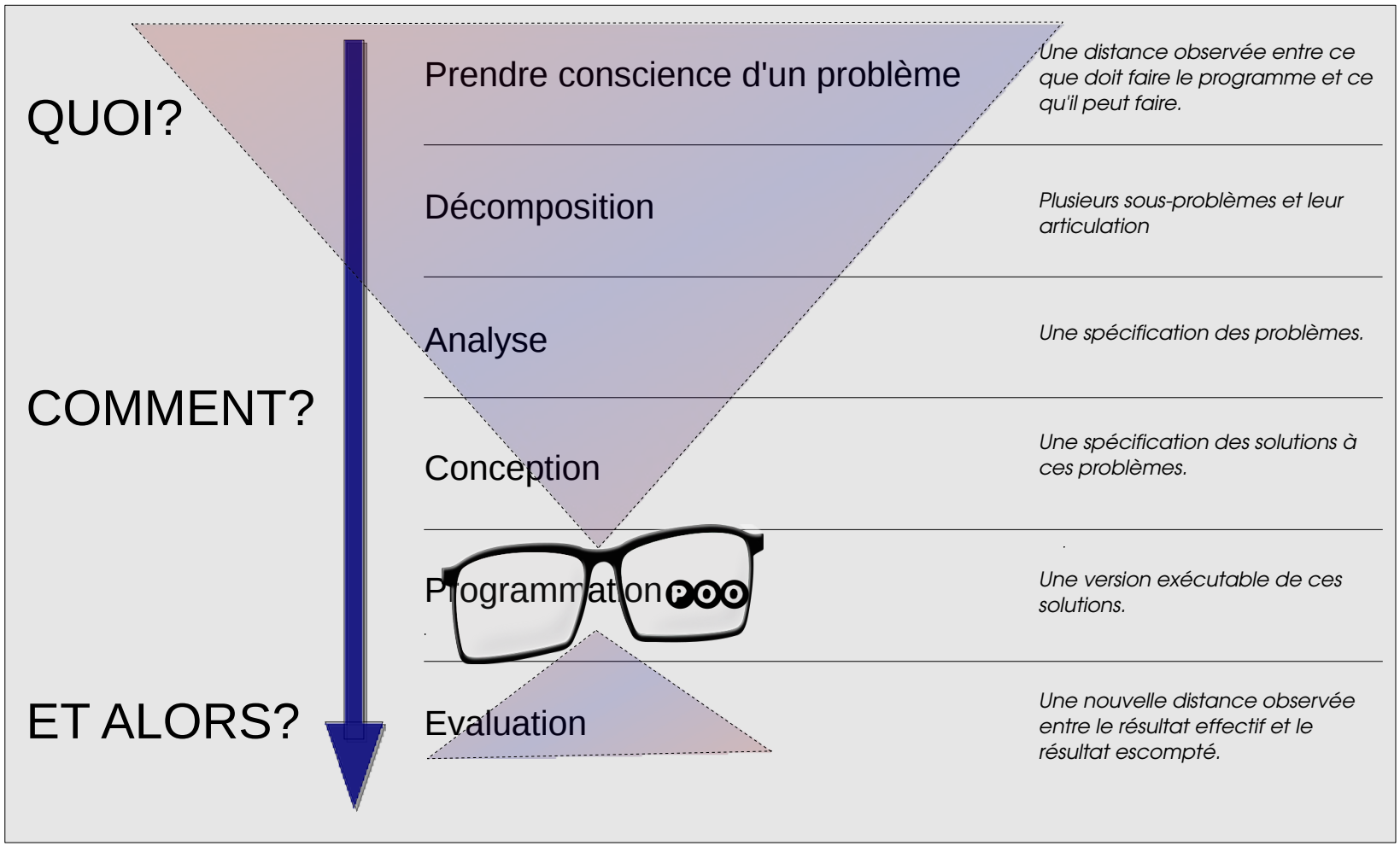
- Étant d'ordre supérieur, elles favorisent toutes deux la programmation modulaire.
- Des façons de développer différentes (mais complémentaires !).

$\Rightarrow$ Sujet du cours de programmation comparée.

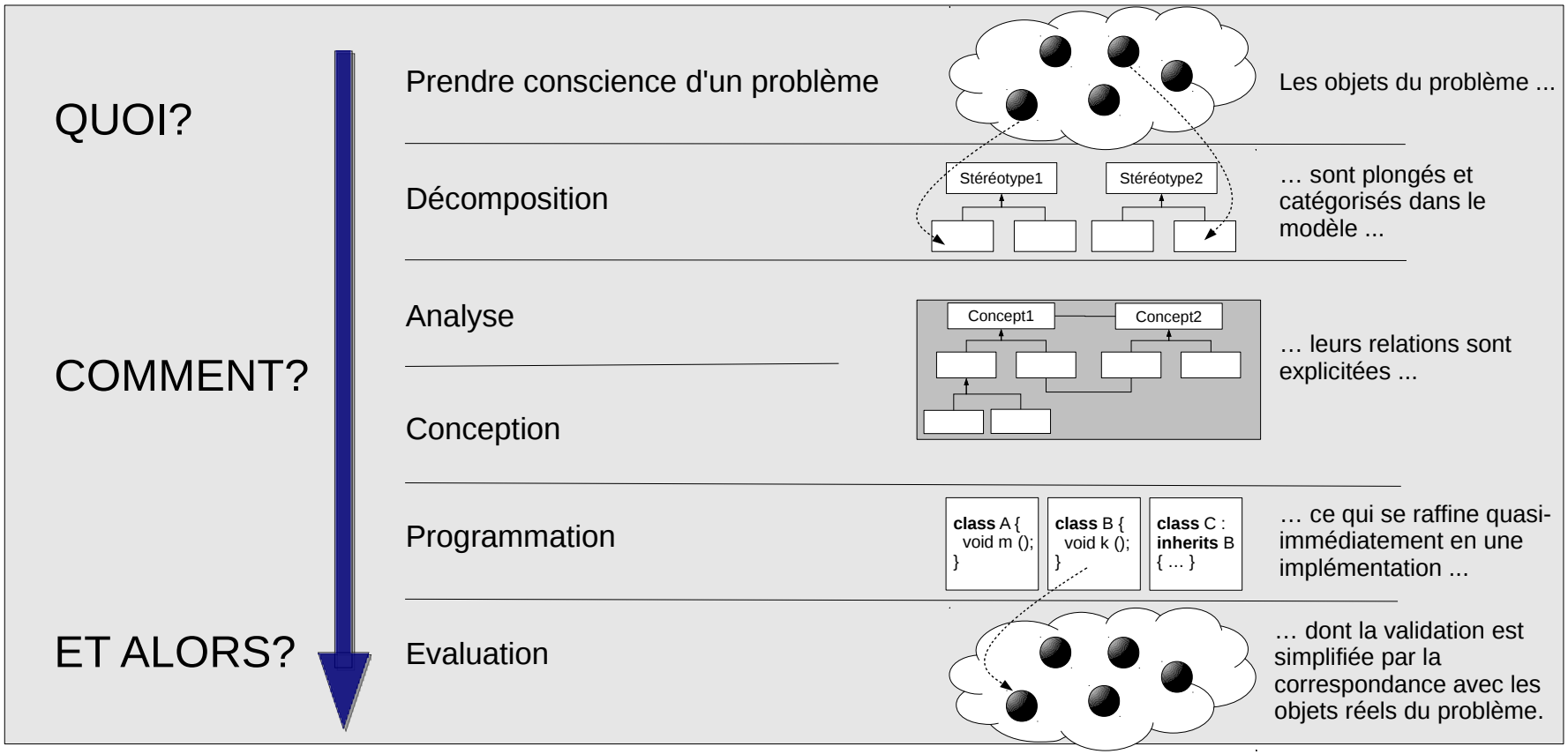
Le développement logiciel



La programmation orientée objet



Le développement logiciel orienté objet



La décomposition en objets

- Un problème est modélisé comme une interaction entre des objets.
- L'état interne d'un objet est opaque pour les autres objets.
- Les interactions s'opèrent *via* des envois de messages.

L'analyse et la conception à l'aide d'objets

- Pour mieux comprendre un problème :
  - On **explicite** des relations d'instance/généralisation entre objets.

  $\Rightarrow$ C'est le principe d'abstraction,
  on cache les détails pour hiérarchiser l'analyse.

  - On **explicite** le sens des relations et des interactions entre objets.

  $\Rightarrow$ C'est le principe d'orthogonalité (*Separation of concerns*),
  on affecte un et un seul rôle (simple) à chaque objet.
- Méthode/croyances objet :
  - « C'est en explicitant le problème que sa solution apparaît. »
  - « Utiliser le vocabulaire du problème à l'intérieur de sa solution informatique permet d'y injecter le savoir-faire du domaine. »

Programmation avec un langage à objets

- Le modèle logique et le modèle d'implémentation sont très proches :
  - Prototyper aide la conception du modèle logique.
  - La phase d'implémentation est courte.

Validation

- Comme les objets du problème sont matérialisés dans le système, écrire des scénarios de tests est facile.
- La faible distance entre les objets du problème et ceux du système nous conforte dans l'idée que l'**on construit le bon logiciel**.

Critique de la programmation objet

Points bénéfiques

- Un système d'objets peut intégrer la complexité d'un problème.
  (En la mimant dans le logiciel.)
- Grâce aux processus de généralisation, la complexité est décomposée graduellement sans perdre de vue le problème concret.
- La programmation objet passe à l'échelle grâce à la modularité.

Défauts et dangers

- La solution au problème global est fragmentée entre les objets suivant un partitionnement aussi complexe que le problème lui-même.

$\Rightarrow$ Il est parfois difficile de raisonner localement sur un objet pour comprendre sa contribution propre à la résolution du problème.

- Il y a des problèmes techniques intrinsèques à la P.O.O.

POCA : un cours de programmation objet **avancée** ?

- Approfondissement et critique des outils de la programmation objet.

Approfondissement

- Mieux comprendre les mécanismes connus.
- Découvrir de nouveaux mécanismes (*via* le langage SCALA).
- Améliorer ses méthodes de raisonnement sur les programmes objet.
- Concevoir des composants logiciels réutilisables.

Critique

- Connaître les limites de la programmation objet.
- Savoir contourner ces problèmes à l'aide :
  - de mécanismes modernes de programmation ;
  - ou *via* des patrons de conception.

Plan

Motivations

De la liaison tardive à la POO

Contenu et fonctionnement du cours

Le programme de POCA

1. Programmation d'ordre supérieur
   - Modularité
   - Construire du code à l'aide d'objets
   - L'objet comme abstraction
2. Programmation générique
   - Généricité fondée sur les types :
     Comment programmer « en toute généralité » ?
   - Méta-programmation :
     Comment automatiser la construction du code source ?

Le langage Scala

- Nous allons étudier et utiliser le langage de programmation Scala.
- Voici ces caractéristiques :
  - Impératif, fonctionnel et objet.

  $\Rightarrow$ Programmation d'ordre supérieur.

  - Statiquement typé.

  $\Rightarrow$ Aide au raisonnement.

  - Système de types (très) riches.

  $\Rightarrow$ Donne un langage pour l'abstraction.
- Nous étudierons aussi certaines particularités des mécanismes objets des langages C#, Python, O'Caml, CLOS...

Pourquoi Scala ?

- Très schématiquement : Java est une synthèse (minimaliste) des résultats la Recherche sur les langages objets de la fin des années 80, et ses “generics” implémentent des résultats du début des années 90 (C# est à peu près dans le même cas, C++ est définitivement des années 80). Scala s'appuie sur des résultats des années 2000 et le langage évolue encore aujourd'hui.
- Scala est compilé vers la JVM et peut utiliser des composants Java de façon transparente (comme si ils étaient des composants Scala natifs).
- Ces arguments en font un successeur naturel à Java et les entreprises suivantes l'intègrent désormais dans leurs langages de développement officiels : Twitter, EDF, Xebia, Xerox, Sony, Siemens, GridGain, AppJet, ...

Fonctionnement du cours

- En cours, prenez des notes et posez des questions !
- Travaux dirigés :

  mardi en salle 2001/2003 de 10h30 à 13h30.
- Le site du cours :

  https://piazza.com/univ-paris-diderot.fr/fall2017/poca/home
- Enregistrement :

  https://piazza.com/univ-paris-diderot.fr/fall2017/poca
- Évaluation :
  - Projet : $\sim$60%
  - Examen : $\sim$40%

Principe du projet

- Le projet est fourni en **deux** temps.
- La version 1 doit être développée en pensant à sa **réutilisabilité**.
- La version 2 doit être développée sans modifier (ou très peu) le code source de la version 1.
- Pour ce projet, la phase d'analyse et de conception est **essentielle**.

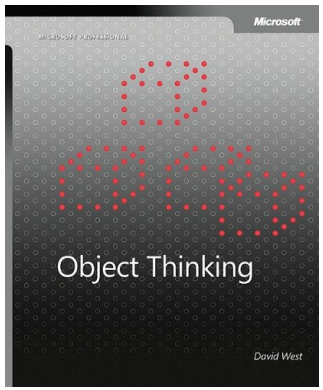
Devoir à la maison

- Avant la prochaine séance de travaux dirigés, vous devrez avoir lu :

  https://www.scala-exercises.org/scala_tutorial/

- C'est un tutoriel **interactif** qui offre un tour d'horizon de Scala.

Bibliographie



- *Object Thinking*,
  David West, Microsoft Professional
  Une exposition synthéthique, mais pas très critique, des idées de la POO.
- *Programming-in-the large versus programming-in-the-small*,
  DeRemer et Kron
  Une explication des problématiques de conception des gros logiciels.
- *Object-Oriented Software Construction*,
  Bertrand Meyer
  Une référence sur la POO.