# Les apports d'un langage de programmation orienté objet.

## A) Les concepts initiaux

### 1) Introduction

On va pouvoir mettre les efforts réalisés en programmation au profit de la réutilisation du code. Il s'agit d'un problème essentiel voire mythique de la programmation. Les avancés sur le sujet sont plutôt mitigées.

D'un côté l'explosion du langage C++ associé à la programmation d'interface utilisateur, le succès de Delphi lui aussi conforté par l'interface utilisateur ainsi que sur l'utilisation des bases de données présentent des avancées certaines.

D'un autre côté, on n'est pas arrivé au stade d'un marché des composants logiciels utilisables et réutilisables à volonté. Non, à la rigueur, on utilise les bibliothèques fournies avec son environnement de programmation préféré, mais jamais du code développé par un tiers. Il n'existe pas à ce jour de solutions toutes prêtes. Un début de solution viendra certainement du côté du langage Ada. Avec Ada, la partie spécification du langage, c'est à dire la partie client, est clairement séparée de la partie réalisation. On peut imaginer, naïvement certes, pouvoir acheter un composant uniquement en regardant sa partie spécification sans s'attacher à connaître le code du composant.

La représentation avec un langage orienté objet (LOO) n'est pas révolutionnaire en elle-même. On peut très bien réaliser des programmes orientés objet (OO) avec un langage qui n'en a pas le nom, voire en assembleur ;-). L'important n'est pas tant le code résultant que le travail d'abstraction ou de modélisation du monde réel qui aura été réalisé auparavant. Déjà N. Wirth, le père du Pascal, prophétisait avec son livre "Algorithms + Data Structure = Programs". Fini le temps où un programme informatique ne se réduisait qu'en une suite d'instructions.

L'appellation "objet" incongrue dans le champ lexical informatique vient de la construction d'une phrase standard : sujet verbe complément d'objet. Voilà introduit le sujet : l'utilisateur étendu au micro-processeur, le verbe : action exécutée par les instructions du micro-processeur et enfin l'objet : ensemble de données modélisant un concept réel, exemple : l'utilisateur déplace le carré (cliquer, glisser) ou l'utilisateur calcul le volume d'un cylindre rond (base * hauteur).

Nous remarquons que l'action est quasi indépendante de la nature de l'objet. Nous pouvons également déplacer un rond ou calculer le volume d'un cylindre à base carré. Pourtant en programmation classique, la procédure de déplacement ou de calcul du volume doit connaitre la nature de l'objet modélisé pour appliquer la bonne action sur les données accessibles globalement. De plus, si par la suite nous ajoutons

un triangle tout est à reprendre alors que comme remarqué auparavant l'action de déplacer (effacer, changer de position, afficher) ou de calcul du volume (aire de la base * la hauteur) se définisse sans préciser la nature de l'objet. Pour se rapprocher de cette abstraction de l'objet, la programmation orientée objet (POO) propose plusieurs concepts que nous allons découvrir.

## 2) L'encapsulation

Dans l'exemple éculé d'un algorithme de tri (voir par ailleurs les arbres binaires sur Blady), ce qui intéresse l'utilisateur n'est pas comment est codé le tri mais bien la possibilité de grouper un ensemble de données puis d'y accéder suivant une relation d'ordre. À contrario, l'intérêt du programmeur de l'algorithme n'est pas de trier tel type de données ou plutôt tel autre mais de coder un algorithme qui fera abstraction du type de données et de sa relation d'ordre. Voilà donc la POO parfaite pour nos deux protagonistes. L'idée est alors de dissocier les différentes parties du programme.

D'un côté, on définira des objets composés de procédures et de données. De l'autre côté on utilisera ces données au travers des procédures, sans jamais entre apercevoir la nature réelle de l'organisation des données. On définit donc l'objet par un type de données abstrait composé d'une partie exportée : les procédures, utilisable par les clients et une partie implémentation : procédures et données, isolée.

Dans le langage des LOO, on dira que ces procédures sont des "méthodes" dont les appels "envoient des messages à l'objet". Les méthodes permettant l'initialisation de l'objet - en général des procédures - sont des "constructeurs". Les méthodes permettant d'obtenir des informations de l'objet - en général des fonctions - sont des "sélecteurs". La définition du type de données abstrait - données + code - est une "classe" et son utilisation est une "instanciation" de l'objet. Vocabulaire à utiliser uniquement en cas de nécessité absolue.

Les données appelées propriétés prennent leur revanche et obtiennent une place égale à l'algorithme dans l'abstraction OO. Le regroupement des propriétés (transcrites dans les données comme longueur, nom) et des actions (transcrites dans les instructions comme déplacer, multiplier) porte le nom d'encapsulation. Chose que l'on pouvait bien faire naturellement jusqu'alors en plaçant les données au plus près des instructions. Avec un LOO on sera en mesure d'avoir un contrôle accru sur la structure du programme. Ces portions de programmes ne sont plus vu comme des sous parties d'un ensemble plus vaste mais comme indépendantes et conçues comme telles.

### 3) L'héritage

La réutilisation est rarement directe mais passe par la modification ou l'ajout de propriétés. L'objectif est alors de réutiliser une partie de l'objet puis soit de modifier, soit d'ajouter d'autres parties pour remplir les fonctions qui nous intéressent. On va donc créer un objet "fils" héritant des propriétés d'un objet "père" en lui modifiant certaines ou en lui en ajoutant d'autres. Il s'agit du concept d'héritage. On a alors la possibilité de créer une hiérarchie complète d'objet héritant les uns des autres. Cela suppose que l'on parte d'un objet relativement simple pour aller à de plus complexes ou de plus spécialisés. Les descendants successifs rendent l'objet de plus en plus spécialisé. Ils ont des propriétés de leurs ancêtres plus de nouvelles propres à eux.

Avec l'héritage multiple un objet peut avoir plusieurs ancêtres indépendants. Là aussi, il s'agit de séparer au mieux les problèmes avec des objets "pères" cohérents hérités par un objet "fils" plus complexe.

Les méthodes de l'objet "fils" peuvent surcharger ou superposer les méthodes héritées des pères et bien sûr en définir de nouvelles.

### 4) La surcharge

La propriété suivante découle des premières. En ramenant ainsi le codage des algorithmes à leur plus simple expression, on se retrouve avec des procédures s'appliquant à des types différents mais ayant le même nom. La "surcharge" permet alors de coder des procédures de même nom ayant paramètres de types différents. Cela s'applique aussi aux opérateurs. Par exemple pour l'opérateur "+", on le retrouve déjà inscrit dans de nombreux langages. Le "+" des entiers est utilisé pour d'autres types de données comme des réels. Avec un LOO, on pourra définir un "+" pour un type vecteur, par exemple.

Ada pratique la surcharge sur tous les types y comprit les types simples.

### 5) La superposition

En spécialisant un objet, une méthode fille peut avoir besoin d'être juste adaptée sans que ses paramètres soient modifiés. La nouvelle méthode spécifique de l'objet fils se superpose à celle de ses parents. Si besoin, elle peut toujours appeler la méthode du père voire celle de n'importe quel parent avec Ada.

## 6) La généricité

La réutilisation se heurte au constat suivant: le programmeur passe la majeure partie de son temps à écrire des procédures similaires mais identiques dans leur fonctionnalité. Reprenons notre exemple du tri. Il nous est évident que l'algorithme du tri est indépendant de la relation d'ordre des données à trier. Il suffit de fournir à l'objet une fonction de comparaison du type de données. Notre objet devient alors "générique" en faisant l'abstraction des propriétés définies hors de l'objet (voir par ailleurs le paragraphe Ada du texte sur les arbres binaires sur Blady).

## 7) Le polymorphisme

Avec un LOO, on peut aller plus loin que la simple réutilisation en anticipant sur le futur. Imaginons un objet avec deux procédures dont l'une polymorphe appelle une plus simple : par exemple "déplace" appelle "affiche".

Le fait d'inclure la procédure "affiche" dans l'objet et non pas comme une procédure classique mais comme procédure abstraite va permettre l'anticipation sur la suite. Mettons qu'alors on ait besoin de créer un objet plus complexe à partir du premier mais seul le code compilé a été fourni. On va quand même pouvoir utiliser l'objet précédent. Il nous faut déclarer un objet fils de l'objet précédent en superposant une procédure "affiche", spécifique à notre objet fils. Et surprise, lors de l'appel à la procédure "déplace", celle-ci appelle maintenant notre nouvelle procédure "affiche". L'investissement antérieur est préservé. Une conception objet adéquate permet des gains de temps et d'énergie incomparables.

Cette propriété, nommée polymorphisme, se conjugue avec l'héritage pour produire des liens entre objet totalement dynamiques. En effet, la décision d'appel de telle ou telle procédure entre les descendants se fera au dernier moment, c'est à dire au moment de l'exécution du programme.

## 8) Conclusion

La POO est non seulement une réponse à la complexité croissante des programmes informatiques, mais aussi elle présente des moyens efficaces de mettre en oeuvre la réutilisation de code.

Pascal Pignard, mars 2001, août-novembre 2005, juillet 2007, mars 2015.