# Programmation orientée objet

La **programmation orientée objet (POO)**, ou **programmation par objet**, est un paradigme de programmation informatique élaboré par les Norvégiens Ole-Johan Dahl et Kristen Nygaard au début des années 1960 et poursuivi par les travaux de l'Américain Alan Kay dans les années 1970. Il consiste en la définition et l'interaction de briques logicielles appelées *objets* ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes. Dès lors, l'étape de modélisation revêt une importance majeure et nécessaire pour la POO. C'est elle qui permet de transcrire les éléments du réel sous forme virtuelle.

Orthogonalement à la programmation par objet, afin de faciliter le processus d'élaboration d'un programme, existent des méthodologies de développement logiciel objet, dont la plus connue est le processus unifié (« *Unified Software Development Process* » en anglais), qui utilisent des langages de modélisation tels que le *Unified Modeling Language* (UML).

Même s'il est possible de concevoir par objets une application informatique sans utiliser des outils logiciels dédiés, ces derniers facilitent de beaucoup la conception, la maintenance, et la productivité. On en distingue plusieurs sortes :

- les langages de programmation (Java, C#, VB.NET, Vala, Objective C, Eiffel, Python, Ruby, C++, Ada, PHP, Smalltalk, LOGO, AS3, Haxe…) ;
- les outils de modélisation qui permettent de concevoir sous forme de schémas semi-formels la structure d'un programme (Objecteering, UMLDraw, Rhapsody, DBDesigner…) ;
- les bus distribués (DCOM, CORBA, RMI, Pyro…) ;
- les ateliers de génie logiciel ou AGL (Visual Studio pour des langages Dotnet, NetBeans ou Eclipse pour le langage Java).

Il existe actuellement deux grandes catégories de langages à objets :

- les langages à classes, que ceux-ci soient sous forme fonctionnelle (Common Lisp Object System), impérative (C++, Java) ou les deux (Python, OCaml) ;
- les langages à prototypes (JavaScript).

## Sommaire



## Origines

Le langage Simula 67, en implantant les Record Class de Hoare, pose les constructions qui seront celles des langages orientés objet à classes : classe, polymorphisme, héritage, etc. Mais c'est réellement par et avec Smalltalk 71 puis Smalltalk 80, inspiré en grande partie par Simula 67 et Lisp, que les principes de la programmation par objets, résultat des travaux d'Alan Kay, sont véhiculés : objet, encapsulation, messages, typage et polymorphisme (via la sous-classification) ; les autres principes, comme l'héritage, sont soit dérivés de ceux-ci ou une implantation. Dans Smalltalk, tout est objet, même les classes. Il est aussi plus qu'un langage à objets, c'est un environnement graphique interactif complet.

À partir des années 1980, commence l'effervescence des langages à objets : C++ (1983), Objective-C (1984), Eiffel (1986), Common Lisp Object System (1988), etc. Les années 1990 voient l'âge d'or de l'extension de la programmation par objets dans les différents secteurs du développement logiciel.

Depuis, la programmation par objets n'a cessé d'évoluer aussi bien dans son aspect théorique que pratique et différents métiers et discours mercatiques à son sujet ont vu le jour :

- l'analyse objet (AOO ou OOA en anglais) ;
- la conception objet (COO ou OOD en anglais) ;
- les bases de données objet (SGBDOO) ;
- les langages objets avec les langages à prototypes ;
- ou encore la méthodologie avec MDA (Model Driven Architecture).

Aujourd'hui, la programmation par objets est vue davantage comme un paradigme, le **paradigme objet**, que comme une simple technique de programmation. C'est pourquoi, lorsque l'on parle de nos jours de programmation par objets, on désigne avant tout la partie codage d'un modèle à objets obtenu par AOO et COO.

## Principes

La programmation orientée objet a été introduite par Alan Kay avec Smalltalk. Toutefois, ses principes n'ont été formalisés que pendant les années 1980 et, surtout, 1990. Par exemple le typage de second ordre, qui qualifie le typage de la programmation orienté objet (appelé aussi *duck typing*), n'a été formulé qu'en 1995 par Cook.

### Objet (attributs et méthodes)

Concrètement, un objet est une structure de données valuées et cachées qui répond à un ensemble de messages. Cette structure de données définit son *état* tandis que l'ensemble des messages qu'il comprend décrit son *comportement* :

- Les données, ou champs, qui décrivent sa structure interne sont appelées ses *attributs* ;
- L'ensemble des messages forme ce que l'on appelle *l'interface* de l'objet ; c'est seulement au travers de celle-ci que les objets interagissent entre eux. La réponse à la réception d'un message par un objet est appelée une *méthode* (méthode de mise en œuvre du message) ; elle décrit quelle réponse doit être donnée au message.

Certains attributs et/ou méthodes (ou plus exactement leur représentation informatique) sont cachés : c'est le principe d'encapsulation. Ainsi, le programme peut modifier la structure interne des objets ou leurs méthodes associées sans avoir d'impact sur les utilisateurs de l'objet.

Un exemple avec un objet représentant un nombre complexe : celui-ci peut être représenté sous différentes formes (cartésienne *(réel, imaginaire)*, trigonométrique, exponentielle *(module, angle)*). Cette représentation reste cachée et est interne à l'objet. L'objet propose des messages permettant de lire une représentation différente du nombre complexe. En utilisant les seuls messages que comprend notre nombre complexe, les objets appelants sont assurés de ne pas être affectés lors d'un changement de sa structure interne. Cette dernière n'est accessible que par les méthodes des messages.

### Typage et polymorphisme

Dans la programmation par objets, chaque objet est typé. Le *type* définit la syntaxe (« Comment l'appeler ? ») et la sémantique (« Que fait-il ? ») des messages auxquels peut répondre un objet. Il correspond donc, à peu de chose près, à l'interface de l'objet. Toutefois, la plupart des langages objets ne proposent que la définition syntaxique d'un type (C++, Java, C#, ...) et rares sont ceux qui fournissent aussi la possibilité de définir aussi sa sémantique (Eiffel avec sa *conception par contrats*).

Un objet peut appartenir à plus d'un type : c'est le *polymorphisme* ; cela permet d'utiliser des objets de types différents là où est attendu un objet d'un certain type. Une façon de réaliser le polymorphisme est le sous-typage (appelé aussi *héritage de type*) : on raffine un type-père en un autre type (le sous-type) par des restrictions sur les valeurs possibles des attributs. Ainsi, les objets de ce sous-type sont conformes avec le type père. De ceci découle le *principe de substitution de Liskov*. Toutefois, le sous-typage est limité et ne permet pas de résoudre le problème des types récursifs (un message qui prend comme paramètre un objet du type de l'appelant). Pour résoudre ce problème, Cook définit en 1995 la sous-classification et le typage du second ordre qui régit la programmation orientée objet : le type est membre d'une famille polymorphique à point fixe de types (appelée *classe*). Les traits sont une façon de représenter explicitement les classes de types. (La représentation peut aussi être implicite comme avec Smalltalk, Ruby, etc.).

On distingue dans les langages objets deux mécanismes du typage :

- le typage dynamique : le type des objets est déterminé à l'exécution lors de la création desdits objets (Smalltalk, Common Lisp, Python, PHP…) ;
- le typage statique : le type des objets est vérifié à la compilation et est soit explicitement indiqué par le développeur lors de leur déclaration (C++, Java, C#, Pascal…), soit déterminé par le compilateur à partir du contexte (Scala, OCaml…).

De même, deux mécanismes de sous-typage existent : l'héritage simple (Smalltalk, Java, C#) et l'héritage multiple (C++, Python, Common Lisp, Eiffel, WLangage).

Le polymorphisme ne doit pas être confondu avec le sous-typage ou avec l'attachement dynamique (*dynamic binding* en anglais).

### Redéfinition

La programmation objet permet à un objet de raffiner la mise en œuvre d'un message défini pour des objets d'un type parent, autrement dit de redéfinir la méthode associée au message : c'est le principe de *redéfinition* des messages (ou *overriding* en anglais).

Or, dans une définition stricte du typage (typage du premier ordre), l'opération résultant d'un appel de message doit être la même quel que soit le type exact de l'objet référé. Ceci signifie donc que, dans le cas où l'objet référé est de type exact un sous-type du type considéré dans l'appel, seule la méthode du type père est exécutée :

Soit un type `Reel` contenant une méthode `*` faisant la multiplication de deux nombre réels,
Soient `Entier` un sous-type de `Reel`, `i` un `Entier` et `r` un `Reel`,
Alors l'instruction `i * r` va exécuter la méthode `*` de `Reel`. On pourrait appeler celle de `Entier` grâce à une redéfinition.

Pour réaliser alors la redéfinition, deux solutions existent :

- le typage du premier ordre associé à l'attachement dynamique (c'est le cas de C++, Java, C#, …). Cette solution induit une faiblesse dans le typage et peut conduire à des erreurs. Les relations entre type sont définies par le sous-typage (théorie de Liskov) ;
- le typage du second ordre (duquel découlent naturellement le polymorphisme et l'appel de la bonne méthode en fonction du type exact de l'objet). Ceci est possible avec Smalltalk et Eiffel. Les relations entre types sont définies par la sous-classification (théorie F-Bound de Cook).

### Classe et prototype

La structure interne des objets et les messages auxquels ils répondent sont définis par des modules logiciels. Ces mêmes modules créent les objets via des opérations dédiées. Deux représentations existent de ces modules : la classe et le prototype.

La *classe* est une structure informatique particulière dans le langage objet. Elle décrit la structure interne des données et elle définit les méthodes qui s'appliqueront aux objets de même famille (même classe) ou type. Elle propose des méthodes de création des objets dont la représentation sera donc celle donnée par la classe génératrice. Les objets sont dits alors *instances de la classe*. C'est pourquoi les attributs d'un objet sont aussi appelés *variables d'instance* et les messages *opérations d'instance* ou encore *méthodes d'instance*. L'interface de la classe (l'ensemble des opérations visibles) forme les types des objets. Selon le langage de programmation, une classe est soit considérée comme une structure particulière du langage, soit elle-même comme un objet (objet non-terminal). Dans le premier cas, la classe est définie dans le runtime ; dans l'autre, la classe a besoin elle aussi d'être créée et définie par une classe : ce sont les *méta-classes*. L'introspection des objets (ou « méta-programmation ») est définie dans ces méta-classes.

La classe peut être décrite par des attributs et des messages. Ces derniers sont alors appelés, par opposition aux attributs et messages d'un objet, *variables de classe* et *opérations de classe* ou *méthodes de classe*. Parmi les langages à classes on retrouve Smalltalk, C++, C#, Java, etc.

Le *prototype* est un objet à part entière qui sert de prototype de définition de la structure interne et des messages. Les autres objets de mêmes types sont créés par clonage. Dans le prototype, il n'y a plus de distinction entre attributs et messages : ce sont tous des *slots*. Un slot est un label de l'objet, privé ou public, auquel est attachée une définition (ce peut être une valeur ou une opération). Cet attachement peut être modifié à l'exécution. Chaque ajout d'un slot influence l'objet et l'ensemble de ses clones. Chaque modification d'un slot est locale à l'objet concerné et n'affecte pas ses clones.

Le concept de *trait* permet de modifier un slot sur un ensemble de clones. Un *trait* est un ensemble d'opérations de même catégorie (clonage, persistance, etc.) transverse aux objets. Il peut être représenté soit comme une structure particulière du langage, comme un slot dédié ou encore comme un prototype. L'association d'un objet à un trait fait que l'objet et ses clones sont capables de répondre à toutes les opérations du trait. Un objet est toujours associé à au moins un trait, et les traits sont les parents des objets (selon une relation d'héritage). Un trait est donc un mixin doté d'une parenté. Parmi les langages à prototype on trouve Javascript, Self, Io, Slater, Lisaac, etc.

## Différents usages de la POO

Différents langages utilisent la programmation orienté objet, par exemple PHP, Python, etc.

En PHP la programmation orientée objet est souvent utilisée pour mettre en place une architecture MVC (Modèle Vue Contrôleur), où les modèles représentent des objets[1].

## Modélisation objet

La modélisation objet consiste à créer un modèle du système informatique à réaliser. Ce modèle représente aussi bien des objets du monde réel que des concepts abstraits propres au métier ou au domaine dans lequel le système sera utilisé.

La modélisation objet commence par la qualification de ces objets sous forme de types ou de classes sous l'angle de la compréhension des besoins et indépendamment de la manière dont ces classes seront mises en œuvre. C'est ce que l'on appelle l'analyse orientée objet ou OOA (acronyme de « *Object-Oriented Analysis* »). Ces éléments sont alors enrichis et adaptés pour représenter les éléments de la solutions technique nécessaires à la réalisation du système informatique. C'est ce que l'on appelle la conception orientée objet ou OOD (acronyme de « *Object-Oriented Design* »). À un modèle d'analyse peuvent correspondre plusieurs modèles de conception. L'analyse et la conception étant fortement interdépendants, on parle également d'analyse et de conception orientée objet (OOAD). Une fois un modèle de conception établi, il est possible aux développeurs de lui donner corps dans un langage de programmation. C'est ce que l'on appelle la programmation orientée objet ou OOP (en anglais « *Object-Oriented Programming* »).

Pour écrire ces différents modèles, plusieurs langages et méthodes ont été mis au point. Ces langages sont pour la plupart graphiques. Les trois principaux à s'imposer sont OMT de James Rumbaugh, la méthode Booch de Grady Booch et OOSE de Ivar Jacobson. Toutefois, ces méthodes ont des sémantiques différentes et ont chacune des particularités qui les rendent particulièrement aptes à certains types de problèmes. OMT offre ainsi une modélisation de la structure de classes très élaborée. Booch a des facilités pour la représentation des interactions entre les objects. OOSE innove avec les cas d'utilisation pour représenter le système dans son environnement. La méthode OMT prévaut sur l'ensemble des autres méthodes au cours de la première partie de la décennie 1990.

À partir de 1994, Booch et Jacobson, rapidement rejoint par Rumbaugh, décident d'unifier leurs approches au sein d'une nouvelle méthode qui soit suffisamment générique pour pouvoir s'appliquer à la plupart des contextes applicatifs. Ils commencent par définir le langage de modélisation UML (*Unified Modeling Language*) appelé à devenir un standard de l'industrie[2]. Le processus de normalisation est confié à l'Object Management Group (OMG), un organisme destiné à standardiser des technologies orientées objet comme CORBA (acronyme de « *Common Object Request Broker Architecture* »), un intergiciel (« *middleware* » en anglais) objet réparti. Rumbaugh, Booch et Jacobson s'affairent également à mettre au point une méthode permettant d'une manière systématique et répétable d'analyser les exigences et de concevoir et mettre en œuvre une solution logicielle à l'aide de modèles UML. Cette méthode générique de développement orienté objet devient le processus unifié (également connu sous l'appellation anglo-saxonne de « *Unified Software Development Process* »)[3]. Elle est itérative et incrémentale, centrée sur l'architecture et guidée par les cas d'utilisation et la réduction des risques. Le processus unifié est de plus adaptable par les équipes de développement pour prendre en compte au mieux les particularités du contexte.

Néanmoins pour un certain nombre de concepteurs objet, dont Bertrand Meyer, l'inventeur du langage orienté objet Eiffel, guider une modélisation objet par des cas d'utilisations est une erreur de méthode qui n'a rien d'objet et qui est plus proche d'une méthode fonctionnelle. Pour eux, les cas d'utilisations sont relégués à des utilisations plutôt annexes comme la validation d'un modèle par exemple [réf. nécessaire].

## Notes et références


1. « Adoptez une architecture MVC en PHP » (https://openclassrooms.com/courses/adoptez-une-architecture-mvc-en-php), sur *OpenClassrooms* (consulté le 30 janvier 2018).
2. **(en)** Booch, Grady. et Jacobson, Ivar., *Le guide de l'utilisateur UML*, Eyrolles, 2000 (ISBN 2212091036 et 9782212091038, OCLC 43833417 (https://worldcat.org/oclc/43833417&lang=fr), lire en ligne (https://www.worldcat.org/oclc/43833417))
3. **(en)** Jacobson, Ivar., Booch, Grady. et Rumbaugh, Jim., *The unified software development process*, Addison-Wesley, 1999 (ISBN 0201571692, 9780201571691 et 9780321822000, OCLC 636807532 (https://worldcat.org/oclc/636807532&lang=fr), lire en ligne (https://www.worldcat.org/oclc/636807532))


## Voir aussi

### Bibliographie


- Brad J. Cox, Andrew J. Novobilski (1986). *Object-Oriented Programming: An Evolutionary Approach*, (ISBN 0-201-54834-8).
- Grady Booch, James Rumbaugh, Ivar Jacobson (2000). *Le guide de l'utilisateur UML*, (ISBN 2-212-09103-6).
- De Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1999). *Design Patterns*, (ISBN 2-7117-8644-7).
- Bertrand Meyer (2000). *Conception et programmation orientées objet*, (ISBN 2-212-09111-7).
- De Hugues Bersini (2007). *L'Orienté Objet*, (ISBN 978-2-212-12084-4).
- Francisco Bonito (2000). *La programmation : l'orienté objet*.


### Articles connexes

- Modèles de conception
- GObject
- Simula
- Encapsulation
- Polymorphisme
- Héritage
- Analyse décisionnelle des systèmes complexes

### Liens externes

- Introduction à la POO (http://hdd34.developpez.com/cours/artpoo/) Apprendre simplement la Programmation Orientée Objet
- Des paradigmes "classiques" à l'orienté objet (http://rb.ec-lille.fr/recherche/Article_objet/httoc.html)
- Analyse et conception orientée objet avec UML et RUP (http://www.beroux.com/france/documents/oop_uml_et_rup.php), un survol rapide.
- **(en)** *The Theory of Classification* de Anthony J.H. Simons sur le JOT (Journal of Object Technology) (http://www.jot.fm/)