

2020

# Programmation par aspect

**SYNTHESE D'UN ARTICLE SUR LA PROGRAMMATION  
PAR ASPECT**  
LEO SCHIRVANIAN

## **Introduction :**

Depuis les débuts de l'informatique avec Alan Turing, nous cherchons en permanence à trouver la meilleure méthode afin de dialoguer avec un ordinateur. Au fil du temps différentes architectures, paradigmes et langages ont vus le jour, essayant de concilier au mieux un contexte donné avec le langage machine et humain. Mais plus le temps passe, plus les programmes évoluent en se complexifiant et se diversifiant jusqu'à devenir indigeste pour l'Homme.

Ainsi, de nombreux chercheurs ont tenté et tentent toujours de trouver la meilleure architecture, le meilleur langage, la meilleure façon de coder, ... afin de continuer cette évolution de l'informatique tout en simplifiant la création et la maintenance de programmes de plus en plus complexes. En outre, l'apparition de nombreux Frameworks, qui ont permis de réutiliser plus facilement ce qui avait déjà été fait par d'autres de façon optimale, a favorisé l'évolution et la complexification de nos applications. Néanmoins, la solution n'est pas toujours dans des langages ou des Frameworks mais simplement dans la manière dont on code, dont on voit les choses : un paradigme.

Ainsi, la programmation objet par exemple a révolutionné la façon de coder et de réaliser des programmes tout en simplifiant leur création. Mais ce paradigme n'est pas le seul. Il en existe d'autres qui répondent à d'autres contextes et critères, et l'un d'entre eux se nomme la programmation par aspect.

## **Entrée en matière :**

Nous allons donc aujourd'hui réaliser la synthèse d'un article datant de 2001, détaillant le paradigme de programmation par aspect. Cet article ayant pour nom sommaire « ASPECT-ORIENTED PROGRAMMING », n'est pas un article scientifique au sens strict du terme mais plutôt une présentation courte de 5 pages sur les avantages et les inconvénients de ce paradigme de programmation.

Cette courte présentation / article a été écrit par :

- Tzilla Elrad : professeur /chercheur au département Computer Science à l'Illinois Institute of Technology de Chicago.
- Robert E. Filman : scientifique travaillant au Research Institute for Advanced Computer Science, NASA Ames Research Center en Californie.

- Atef Bader : membre de l'équipe technique de Lucent Technologies (grande entreprise de télécommunication) à Chicago.

On remarque donc que cet article est avant tout co-écrit par des personnes expertes dans le domaine vaste de l'informatique. En revanche, s'il y a une telle hétérogénéité au niveau du métier des auteurs, c'est avant tout pour souligner le fait que la programmation par aspect peut être utile dans de nombreux contextes dans un projet informatique. Allant de l'équipe technique d'une entreprise de télécommunication, de scientifiques de la NASA ou encore de chercheur en « computer science », la programmation par aspect représente donc un spectre plutôt large de professionnels qui s'associent pour en décrire l'utilité le temps d'un court article.

Cet article est loin d'être récent, presque 20 ans. Malgré le fait qu'on puisse affirmer sans problème que la programmation par aspect n'a pas explosé en popularité depuis, cet article est toujours d'actualité puisqu'il permet de comprendre ce paradigme, les questions qui se posaient à l'époque quant à son amélioration et la résolution des problèmes existants. Ainsi, nous ne sommes pas sur un article d'actualité mais d'histoire : l'histoire de la programmation par aspect.

Bien que cet article soit court en lui-même et peu récent, il suffit largement à poser les bases d'une réflexion poussée sur ce paradigme de programmation : son utilité, ses forces et ses faiblesses. Ce n'est donc pas un article publié dans une revue spécialisée, ni une thèse mais l'essentiel est là. Et cela nous permet de faire un pas en avant vers une nouvelle façon de programmer toujours plus efficace.

## **Résumé de l'article :**

De l'assembleur à nos programmes actuels, il n'y a pas eu que des améliorations techniques et technologiques mais aussi de nouveaux concepts et paradigmes pour répondre aux nouvelles problématiques posées par ces améliorations. A l'instar du paradigme de la programmation objet qui a révolutionné la façon de coder, de nombreux paradigmes ont vu le jour selon les contextes. Ce dernier n'étant pas parfait, car tout problème ne peut parfaitement se découper en objet à chaque fois, des chercheurs ont mis au point un nouveau paradigme afin de combler le manque d'expressivité de la programmation objet : la programmation par aspect.

La programmation par aspect est basée sur une idée : il est plus efficace de programmer en séparant spécifiquement des propriétés, des préoccupations ou des aires d'intérêt communes (le terme « concerns » est

très utilisés par les auteurs, mais sa traduction en français est délicate, on a donc choisi de refléter cette idée d'objectif premier commun, de centre d'intérêt commun). Ces derniers peuvent être très bien de haut niveau, comme la sécurité ou la qualité du code, ou bien bas niveau, comme le « caching » ou le « buffering ». L'idée principale de la programmation par aspect, c'est d'enfin associer réellement les principes communs entre eux et plus de les forcer à être de multiples objets.

En effet, la programmation objet essaie de trouver des similitudes entre les différentes classes et de les faire rentrer dans un arbre hiérarchique, qui peut s'avérer vite complexe. En revanche, la programmation par aspect, elle, tente de réaliser un découpage des différents centres d'intérêts et de les réunir dans un seul et même programme. Il convient donc pour la programmation par aspect de simplifier le découpage de ces différents points communs.

On peut alors se dire que depuis le langage Fortran, il est possible de découper des bouts de codes dans des sous programmes pour faciliter la programmation et la compréhension. Et tout cela est vrai, la programmation par aspect ne rejette absolument pas les paradigmes ou technologies existantes pour les surpasser, mais elle est là pour corriger les contraintes de ces derniers. En effet, il est extrêmement complexe dans la plupart des cas de définir clairement ce qu'est un « centre d'intérêt ». Si ce découpage est mal fait, et qu'on s'en rend compte trop tard, le code risque de devenir vite un véritable labyrinthe.

Pour pallier à ce problème, la programmation par aspect offre un mécanisme que l'on nomme les aspects. Ces derniers permettent de réaliser des découpages (ou cross-cutting en anglais) afin d'ajouter du code dans le code principal. Ainsi, nous avons un squelette : le code principal, où viennent se tisser plusieurs aspects répondant à diverses propriétés de l'application. Les deux formants alors un tout cohérent et exploitable.

Cela permet notamment d'apporter de multiples avantages comme une meilleure adaptabilité, un code modifiable plus facilement, un code plus compréhensible et réutilisable et un meilleur suivi de la vie du programme. De surcroît, la programmation par aspect ne demande plus de connaître de nombreux algorithmes spécialisés dans différents domaines pour faire que le tout fonctionne en harmonie. On peut à présent grâce à la programmation par aspect, les utiliser proprement sans se concentrer sur ce qui les entoure réellement, ce qui est un avantage indéniable avec l'explosion de nouvelles fonctionnalités sur de multiples logiciels/programmes.

Néanmoins, la programmation par aspect soulève encore de nombreuses questions. Comment spécifier les aspects dans un programme existant ?

Comment l'implémenter ? Les problèmes de découplage entre objet et aspect ? Comment gérer les différents aspects efficacement ? etc

La programmation par aspect est donc un incroyable terrain de recherche et est déjà utile dans différentes applications comme les « middlewares », la sécurité, la tolérance aux fautes, les systèmes d'exploitation, ... Certains langages, Framework basés sur la programmation par aspect ont vu le jour comme le Framework AspectJ. Mais ce paradigme doit encore être amélioré afin que son utilisation se démocratise et se place à sa juste valeur.

## **Réflexion critique théorique :**

Sur le papier, la programmation par aspect a beaucoup de choses à nous apprendre sur notre façon de coder. On a souvent tendance à considérer que coder en objet est la seule et unique façon élégante et propre de programmer. Or, il ne faut jamais avoir coder pour savoir que lorsqu'on ne passe pas préalablement un certain temps à réfléchir sur l'architecture de son code, on devient rapidement perdu et abattu.

C'est pourquoi la programmation par aspect parle à tous ceux ayant déjà programmer en objet. Qui n'a jamais rêver de simplifier ces schémas UML bien trop complexes et flous, ou encore d'arrêter de s'arracher les cheveux pour savoir comment découper un problème complexe en objet qui par définition seront eux aussi complexes à manipuler ? A l'instar du paradigme de programmation fonctionnelle, tout nouveau paradigme éveille en nous, une curiosité et une recherche de nouveauté, de simplicité et d'efficacité toujours plus grande. A l'heure où toutes les applications, logiciels, programmes, machines, systèmes qui nous entourent sont devenus tellement complexes et sont le résultat d'expertise dans tellement de domaines différents, la recherche de simplification est énorme.

De plus, la programmation par aspect est un paradigme qui prend en compte les anciens paradigmes, contrairement par exemple à la programmation fonctionnelle. Ici, la programmation par aspect est tout à fait utilisable avec une programmation objet par exemple. De nombreux Frameworks, bibliothèques, extensions ont été développés depuis la création du paradigme. Pour ne citer que les plus connus, on peut parler de AspectJ et Spring en Java, AspectC++ en C++, ... Il est donc tout à fait possible d'inclure des aspects et des cross-cuttings dans des projets déjà existants.

Cependant, il est important de ne pas se bercer d'illusions, la programmation par aspect ne rend pas par magie les projets complexes plus simples. C'est une nouvelle manière de penser qui peut excellement bien sied dans certains contextes que nous détaillerons dans la prochaine

partie. A l'instar, de la réflexion par exemple, il permet d'insérer du code à haut niveau dans un programme existant. En effet, ces derniers font partie de ce qu'on appelle la programmation post-objet (POP ou Post Object Programming en anglais). Tous les mécanismes, paradigmes de POP ont pour objectif premier d'améliorer l'expressivité d'un projet programmé objet. A l'instar, de la réflexion qui permet de générer du code à haut niveau selon le contexte, la programmation par aspect permet des points d'insertion de code de haut niveau au déclenchement de certains actions.

Nous n'allons pas donner une liste exhaustive de toutes les actions qui permettent de déclencher le cross-cutting et appeler ainsi un aspect (si vous voulez tout savoir je vous recommande la documentation du Framework AspectJ). On peut par exemple appeler ainsi un aspect lors de l'appel ou l'exécution d'une méthode (on peut même filtrer selon son package, sa classe, le nom de la méthode, la signature d'entrée et de sortie), à l'appel du constructeur, à l'initialisation statique d'une classe, quand un attribut de classe est modifié, lors d'un traitement d'exception, ... Le but est donc de permettre d'insérer du code de haut niveau de manière précise selon les besoins. Les possibilités sont assez variées et ne demandent qu'à être exploitées.

En revanche, il existe de nombreux problèmes concernant ce paradigme, notamment sa lisibilité. Ce qui est assez paradoxal puisqu'il souhaite la simplifier. En effet, peu importe la méthode, la façon de coder, un projet complexe restera par définition complexe si on essaie de le découper soit en objet, en « concerns », ... (la définition d'un projet complexe est que chaque sous projet de ce projet est forcément complexe). Ainsi, la programmation par aspect permet de simplifier certaines choses, c'est vrai, mais elle est plutôt complexe à lire. En effet, on imagine un projet rempli de cross-cutting et d'aspects insérés partout dans un code déjà dense et conséquent, ce qui n'aide pas à la lisibilité de ce dernier.

En outre, le débogage de ce genre de projet est aussi compliqué, même si de nouveaux outils apparaissent pour palier à ce problème comme le AspectJ Development Tools d'AspectJ qui permet un débogage transparent et efficace. Il existe de nombreux problèmes concernant la cohabitation avec le code principal. En effet, l'intérêt des aspects est de séparer les différents domaines d'expertise d'un projet. Or, il serait tout à fait absurde d'imaginer un programme où toutes ces expertises ne se rencontrent pas à un moment. Cela ne reste donc qu'une utopie, on sépare les différents domaines, mais il faudra forcément que quelqu'un connaisse plusieurs éléments pour pouvoir les associer. C'est certes plus simple qu'en programmation objet mais à l'instar des micros-services, séparer veut intimement signifier qu'il faudra préparer une réunification ultérieurement.

## **Réflexion critique pratique :**

Pour être totalement franc, ma faible expérience pratique de ce paradigme risque d'influencer grandement mon jugement. Des doutes et des questions légitimes qu'on peut se poser sur ce paradigme sont peut-être solvables par une habitude et une expertise à coder en aspect. J'essaie donc de prendre ce biais en compte pour pouvoir apporter une opinion la plus factuelle possible.

Il est vrai que la programmation par aspect nous parle directement, surtout au niveau de la redondance du code ou du suivi de l'application. Il est facile d'imaginer par exemple un cross-cutting sur chaque méthode pour connaître leur temps d'exécution respective, ce qui serait plutôt intéressant à faire à la main ou en objet. Ici, un simple cross-cutting et un bon filtre suffisent. Les possibilités sont assez impressionnantes, et il faudrait forcément une bonne expertise sur le sujet pour en mesure toutes les applications possibles.

Le plus simple, personnellement, pour mesurer les possibilités de la programmation par aspect, serait de la confronter à mon domaine d'expertise : l'intelligence artificielle, le machine learning et la science des données. On peut rapidement voir des applications assez utiles dans ce domaine. La première application qui me vient à l'esprit serait de faire des cross-cutting pour ce que l'on nomme le « data cleaning » ou nettoyage de données. En effet, en apprentissage machine (machine learning) le résultat de votre modèle dépend énormément de vos données d'apprentissage. Il est donc absolument essentiel de nettoyer ces données et de les arranger de manière à obtenir un bon score. Normalement, on utilise des notebooks pour séparer l'exécution des différentes méthodes, c'est un simple script. Or, plus ce script devient grand, plus il devient confus. De plus, si on fait une erreur, il faut très souvent réexécuté plusieurs cellules du notebook pour la corriger surtout sur les données (il faut les réimporter et les retraitier). Ainsi, avec les aspects, ils seraient possibles d'intégrer le traitement de données directement via des cross-cutting. De surcroît, tous les tests statistiques classiques : moyenne, écart type, p-value, ... pourrait être simplifiés avec des aspects, ce qui améliorerait la lisibilité et la qualité du code de manière significative.

On pourrait même imaginer faire un ensemble d'aspect où on décrit les algorithmes les plus récurrent en intelligence artificielle. Bref, dans mon domaine d'expertise les possibilités sont assez intéressantes. Mais si elles l'étaient vraiment alors cela aurait été démocratisées depuis longtemps. Loin de moi, l'idée de penser que tout ce qui est populaire est forcément efficace et optimal mais si la programmation par aspect se fait si discret

dans mon domaine d'expertise comme dans beaucoup de domaine de l'informatique c'est que quelque chose pose encore problème.

Pour ma part, je pense que nous sommes trop habitués à coder en script ou en objet. J'ai peu d'expériences professionnelles et beaucoup de théorie, il serait donc facile d'infirmer ce que j'avance. Dans tous les cas, il est indéniable que lors de notre scolarité, nous ne sommes formés presque essentiellement au paradigme objet. Bien que le Big Data amène au fur et à mesure la programmation fonctionnelle, celle par aspect se fait toujours discrète.

En effet, je pense qu'à l'heure actuelle, la programmation par aspect est un paradigme de « niche », appliqué dans des cas très spécifiques. De surcroît, il n'est pas essentiel de coder avec des aspects mais plutôt pour simplifier le code et l'optimiser pour une réutilisation future, et nous savons que souvent beaucoup d'entreprises ne recherchent pas la propreté du code mais son fonctionnement quoi qu'il advienne (même s'il est évident que les deux ont une relation de causalité).

Quand on recherche quels sont les applications réelles des aspects, on explique souvent le cas d'un programme de banque, où on pourrait plus simplement traiter des domaines divers comme par exemple l'authentification, le multi-threading, le cryptage, la gestion des utilisateurs, gérer les codes redondants, la sécurité, les logs, l'archivage des données, requêtes aux bases de données, ... Il est donc tout à fait envisageable de réaliser un projet complexe avec des aspects. Les améliorations qu'ils apportent ne sont pas essentiels comme le fut les objets et comme l'est actuellement la programmation fonctionnelle en Big Data avec Apache Spark par exemple, mais néanmoins ils permettent d'améliorer efficacement la qualité, l'architecture et l'utilisation d'un programme de manière significative.

## **Conclusion :**

La programmation par aspect est donc un paradigme aux multiples vertus pour tous ces programmes de plus en plus complexes et nombreux. Même si sa popularité est encore faible, c'est à nous, futurs ingénieurs, de prendre en compte ce paradigme pour nos futurs projets et de l'implémenter pour enfin rendre nos applications, sites, programmes, logiciels, ... les plus claires et propres possibles.

Car en effet, coder en soit, n'est pas la tâche la plus essentielle dans un projet, mais c'est plutôt de savoir comment coder. Avec la programmation par aspect, on peut nettement améliorer les programmes objets à la manière d'une réflexion par exemple. Il est donc important de connaître ce paradigme et de le garder en tête pour répondre à de nouveaux problèmes futurs encore jamais rencontrés.

## **Source :**

- [Mohamed Youssfi, 2015] Aspect Oriented Programming AOP  
AspectJ : <https://youtu.be/xOwPr8tJkrk>
- Documentation AspectJ :  
<https://www.eclipse.org/aspectj/doc/released/>
- Article d'origine : [Tzilla Elrad, Robert E. Filman, Atef Bader, 2001]  
Aspect oriented programming :  
<https://dl.acm.org/doi/pdf/10.1145/383845.383853>
- [David Durand-Christophe Logé, 2004] Spécifications et gestion d'informations de QoS dans les applications réparties par les approches modèles et programmation orientée aspect  
[https://www.researchgate.net/profile/David\\_Durand2/publication/242087516\\_Specifications\\_et\\_gestion\\_d'informations\\_de\\_QoS\\_da\\_ns\\_les\\_applications\\_reparties\\_par\\_les\\_approches\\_modeles\\_et\\_pro grammation\\_orientee\\_aspect/links/56dde10d08ae46f1e99f8eb3.pdf](https://www.researchgate.net/profile/David_Durand2/publication/242087516_Specifications_et_gestion_d'informations_de_QoS_da_ns_les_applications_reparties_par_les_approches_modeles_et_pro grammation_orientee_aspect/links/56dde10d08ae46f1e99f8eb3.pdf)
- [Jean Baltus, 2002] La Programmation Orientée Aspect et AspectJ: Présentation et Application dans un Système Distribué :  
<https://staff.info.unamur.be/ven/CISma/FILES/2002-baltus.pdf>