DU PARADIGME IMPERATIF AU PARADIGME ORIENTE OBJET EVOLUTION OU REVOLUTION ?

André DELACHARLERIE
CeFIS - FUNDP - Namur
Rue Lambin, 25
B-5100 JAMBES (BELGIQUE)
Tél/Fax : +32 81 30 40 22

1. INTRODUCTION

Près de trente années se sont écoulées depuis que O.J. DAHL et K. NYGAARD ont proposé les fondements du paradigme orienté objet dans le langage SIMULA en 1966. Or cette nouvelle vision des systèmes informatiques reste encore trop largement méconnue aujourd'hui, tant auprès des professionnels de l'informatique que dans la communauté éducative. Pourtant, les ouvrages et les articles qui présentent le paradigme orienté objet comme le remède quasi miracle à tous les maux de l'informatique moderne se multiplient de plus en plus et il n'est pas d'éditeur de logiciel ou de langage de programmation qui ne se réclame de l'orientation objet (entre autres OS/2, Turbo Pascal...).

L'enseignant d'informatique de l'école secondaire est donc naturellement perplexe : le paradigme orienté objet est-il donc bien la panacée que l'on décrit et y a-t-il lieu de jeter aux orties les "vieux" paradigmes et en particulier le paradigme impératif ou paradigme de von Neumann [CLOUTIER 88] qui naquit avec l'informatique elle-même et qui fut le compagnon d'apprentissage de pratiquement tout informaticien professionnel ou amateur.

Dans cette communication, nous voudrions donc jeter un regard aussi objectif que possible pour resituer tout d'abord les concepts fondamentaux du paradigme orienté objet et ensuite argumenter quelques comparaisons entre les deux paradigmes qui nous intéressent pour les situer mutuellement. Nous terminerons par quelques réflexions sur la place que pourrait occuper l' "orienté objet" dans notre enseignement.

2. CONCEPTS ET PRINCIPES FONDATEURS DU PARADIGME ORIENTE OBJET

On vient de le rappeler, les premières bases du paradigme orienté objet[1] sont dues à des recherches visant à simuler sur ordinateur les processus du monde réel tels que le trafic d'un

[1] Suivant les auteurs, il est tantôt question de paradigme orienté objet ou de paradigme objet sans que ces deux expressions aient manifestement des significations différentes. Par analogie avec les paradigmes impératif, cartésien, systémique, ... il aurait peut-être été judicieux de choisir un adjectif qui signifie "relatif aux objets". Malheureusement l'adjectif "objectal" semble réservé, selon Larousse à la psychanalyse. Aussi, nous continuerons de parler indifféremment du paradigme objet ou orienté objet.

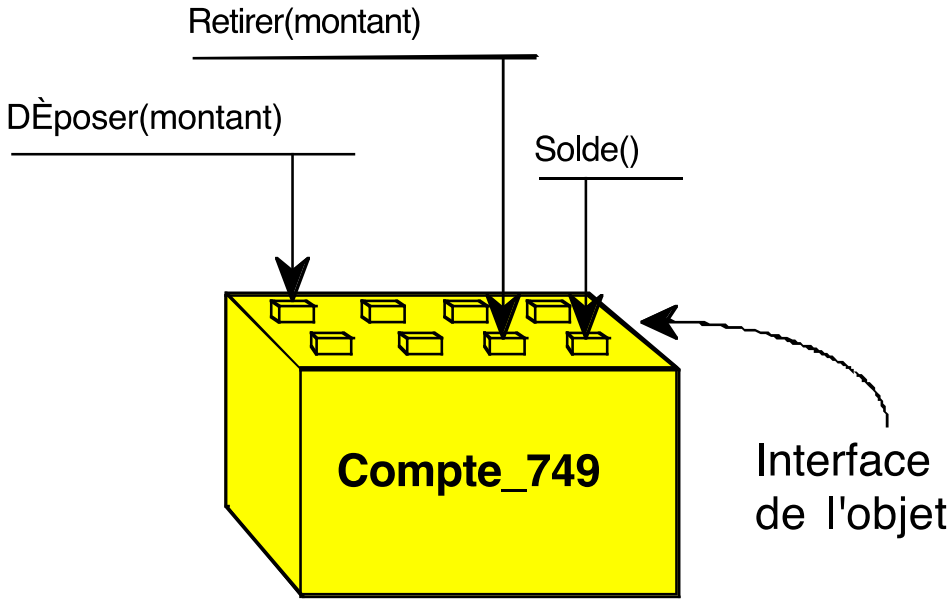
port ou l'utilisation des services d'une gare [FERBER 91]. Globalement l'hypothèse de départ consiste à postuler que tout système est constitué d'entités qui interagissent les unes avec les autres en s'envoyant des messages permettant ainsi le déclenchement chez l'entité destinataire de certains comportements bien précis. Ces comportements sont quant à eux bien spécifiques à chaque entité et sont produits par référence à un ensemble d'informations que l'entité mémorise sur son état ainsi que par des mécanismes d'action qui lui sont propres.

De tels systèmes sont en fait très nombreux dans le monde qui nous entoure. Comme le fait remarquer D.A. Taylor, les organismes vivants sont eux-mêmes structurés de cette façon. "La cellule est la brique de base dont est composé tout corps vivant. Les cellules sont des "paquets" organiques qui, comme les objets, combinent des informations et des comportements. (...) Les cellules sont entourées d'une membrane qui ne permet que certains types d'échanges chimiques avec les autres cellules. Cette membrane (...) dissimule aussi la complexité de la cellule et présente une interface relativement simple avec le reste de l'organisme. Toutes les interactions entre cellules ont lieu par le biais de messages chimiques reconnus par la membrane et transmises à l'intérieur de la cellule." [TAYLOR 93]

On peut observer que l'ensemble des institutions et des personnes d'un pays, ou l'ensemble des services d'une entreprise offrent les mêmes particularités d'organisation. Aussi, il semble donc tout naturel d'appliquer au domaine du logiciel des principes d'organisation que l'on peut voir fonctionner avec une (certaine !) efficacité dans le monde de la réalité, car un logiciel n'est finalement qu'un modèle qui simule un système réel ou même imaginaire.

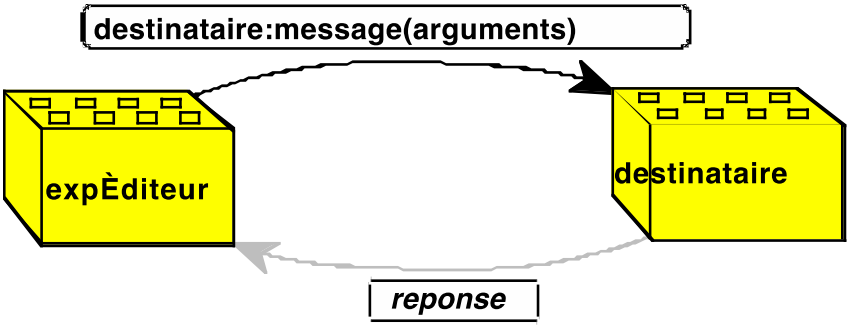
Analysons donc plus en détail ces différents principes qui structurent le paradigme objet.

a. Principe d'encapsulation

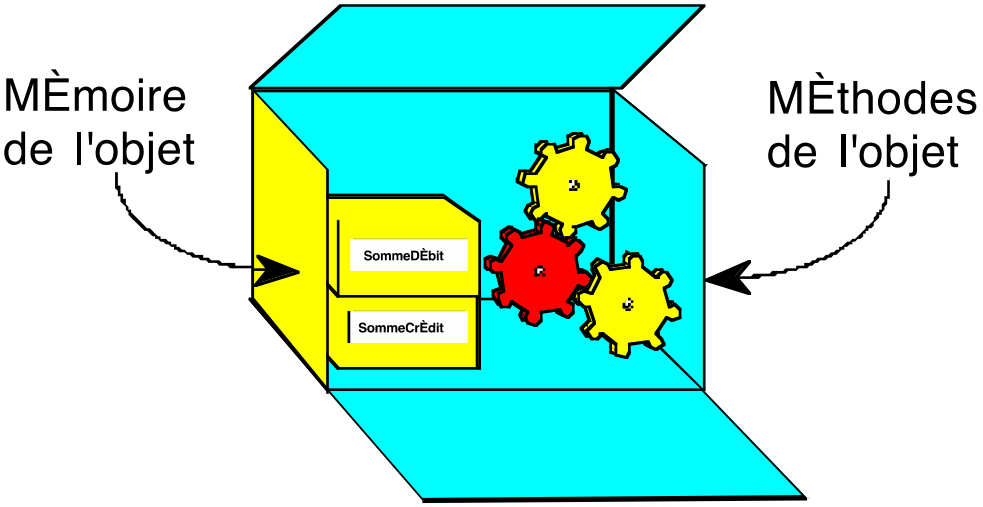


Le premier principe, le plus fondamental est certainement celui de l'encapsulation qui consiste à définir un objet selon deux points de vue. La vue publique d'un objet est celle qui le fait apparaître comme une "boîte noire" qui offre une série de services qui peuvent être

activés en envoyant à l'objet des messages appropriés. L'ensemble de ces messages reconnus par l'objet forment l'interface de l'objet. Le seul moyen (en principe tout au moins) de provoquer une action chez un objet consiste à envoyer le message approprié à cet objet. Le cas échéant, au terme de l'action, l'objet récepteur du message pourra retourner une information à l'objet émetteur en réponse à son message.



Un objet peut aussi être considéré selon le second point de vue en "ouvrant" la boîte noire et en accédant ainsi à sa vie "privée". On observe alors que tout objet comporte deux composants internes : sa mémoire et ses méthodes. La mémoire de l'objet est un ensemble d'informations décrivant l'état actuel de l'objet et éventuellement l'historique de ses états antérieurs. Les méthodes sont au contraire des algorithmes qui décrivent minutieusement les comportements qui seront produits en réponse aux messages reçus.



Bien entendu, ces algorithmes consulteront, et modifieront le cas échéant les informations de la mémoire de l'objet pour produire le comportement attendu. Ainsi, si l'on considère l'exemple de la modélisation d'un compte bancaire par un objet informatique, les messages renseignés dans l'interface de l'objet pourraient être :

`Solde()` qui "demande" à l'objet de répondre en fournissant son solde actuel;

`Déposer(Montant)` et `Retirer(Montant)` qui "demandent" à l'objet d'enregistrer respectivement un dépôt ou un retrait d'un certain montant (qui doit être précisé en argument).

Ces messages sont déjà suffisants pour permettre à plusieurs objets d'interagir et donc d'évoluer. On notera que l'on peut donc "utiliser" ces objets `Compte` sans rien connaître de

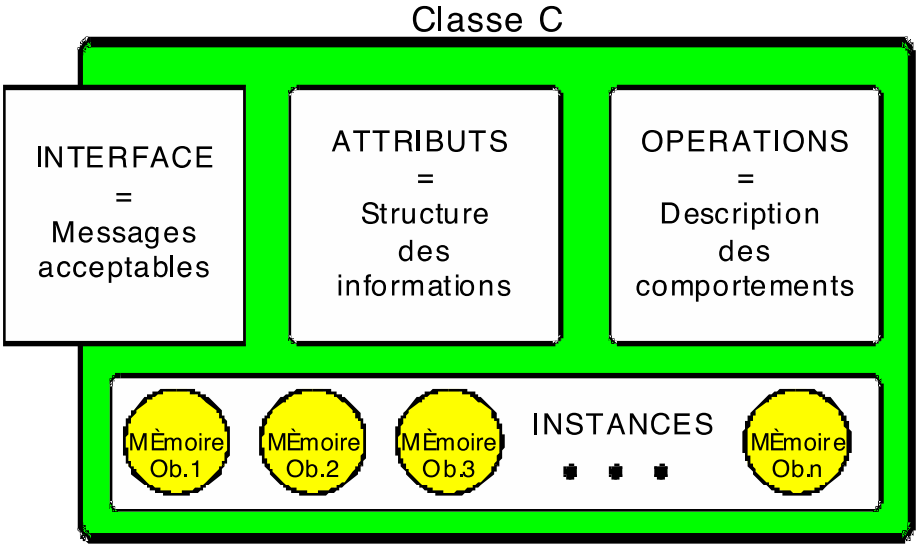
l'organisation interne de leur mémoire ni de l'algorithme qui est employé pour produire les comportements attendus.

Il faut bien voir que ce mécanisme d'encapsulation peut agir à plusieurs niveaux. Un objet peut ainsi être constitué, en interne, de plusieurs objets composants de plus bas niveau et contenir les références vers d'autres objets du système. Ainsi, de façon interne, un objet `Compte` peut être, entre autres, constitué d'un objet `ListeOpérations` lui-même décomposable en de nombreux objets `Opération` ainsi que d'une référence vers un objet `Personne` décrivant le titulaire du compte.

b. Principe d'abstraction

Si le principe d'encapsulation n'est pas spécifique du paradigme objet, il en va de même du principe d'abstraction qui est lui aussi intimement lié à la science informatique depuis ses débuts. L'abstraction est présente à au moins deux niveaux. Le premier est celui de la réification : c'est "l'opération essentielle du paradigme objet par laquelle quelque chose (chose physique, relation, événement, situation, idée, loi, etc.) est représenté informatiquement sous la forme d'un objet". [FERBER 92]. Un second niveau d'abstraction est cependant presque toujours présent. Il s'agit de la mise en évidence de classes d'objets partageant un certain nombre de caractéristiques communes et disposant des mêmes comportements. Le mécanisme inverse de l'abstraction est celui de l'instanciation et consiste à construire un objet informatique à partir de sa classe.

Ce principe d'abstraction est également très présent dans le domaine des systèmes d'informations et en particulier des modèles conceptuels dont le modèle Entité-Association est l'un des mieux connus. Les concepts de classe et de type d'entités sont extrêmement voisins; la richesse du premier se situant dans sa capacité à prendre en compte non seulement les aspects informationnels des entités représentées, mais aussi les aspects comportementaux.

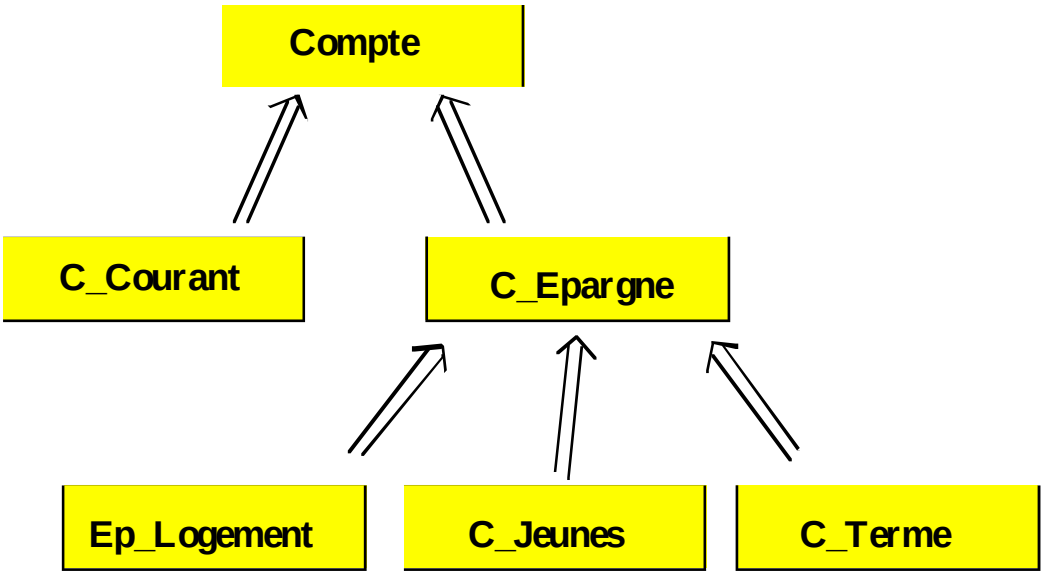


Aussi une classe peut-elle être décomposée en quatre constituants essentiels dont trois sont partagés par l'ensemble des objets-instances de la classe. Il s'agit respectivement de l'interface de la classe, décrivant les messages acceptables avec la spécification des

comportements associés, de la liste des attributs définissant la structure des informations mémorisées par chaque objet et de la liste des opérations (ou méthodes) qui décrivent les algorithmes nécessaires à la production des comportements attendus. Le quatrième composant de la classe est constitué par l'ensemble des instances elles-mêmes ou plus exactement des informations (valeurs) mémorisées par chacune d'elles.

c. Principe de généralisation et spécialisation

Troisième pilier du paradigme objet, le principe de généralisation et spécialisation est celui qui est probablement le plus novateur. On sait que la redondance dans un système d'information est toujours génératrice de problèmes lorsqu'elle est mal maîtrisée. Il en va de même dans le code des logiciels.



Le principe de spécialisation permet de s'affranchir en bonne partie de cette redondance en offrant la possibilité de décrire de nouvelles classes d'objets (dites sous-classes) par l'énoncé des différences par rapport à une première classe, appelée superclasse. Tous les éléments (messages, attributs ou opérations) qui ne sont pas redéfinis sont simplement hérités de la superclasse et sont donc immédiatement disponibles sans aucune redondance de définition. Ainsi, par exemple, si l'on désire décrire des classes `CompteCourant` et `CompteEpargne`, il s'indique de les faire hériter d'une même superclasse que l'on peut appeler classe `Compte`. Ainsi tous les éléments de description partagés par les différents types de comptes peuvent être décrits une et une seule fois au niveau de la classe `Compte` (par exemple les opérations `Solde()` ou `Déposer(Montant)`) tandis que les caractéristiques plus spécifiques seront décrites dans leurs classes respectives (par exemple `CalculerIntérêts()` pour la classe `CompteEpargne`).

La spécialisation consiste donc à décrire de nouvelles classes en raffinant la description de classes déjà définies tandis que la généralisation permet de travailler dans l'autre sens, par une nouvelle forme d'abstraction qui conduit à définir une classe plus générale pour rassembler des caractéristiques communes à plusieurs classes. Généralisation et spécialisation sont à la base du mécanisme de l'héritage qui permet, ainsi, en association

avec l'abstraction, d'étendre le principe de non-redondance du niveau des données à celui des algorithmes.

d. Polymorphisme, surcharge, généricité...

La combinaison des trois mécanismes fondamentaux permet de faire émerger plusieurs concepts nouveaux qui apportent d'intéressantes améliorations dans la conception et l'architecture des logiciels en permettant une écriture plus souple et plus économique des comportements associés aux classes.

Le polymorphisme est ainsi la propriété qui permet de définir, dans plusieurs classes ayant un ancêtre commun, un message de même nom et correspondant à une spécification globalement identique. Toutefois dans chaque classe, le comportement associé à ce message peut être modulé différemment pour s'adapter aux spécificités de chaque classe. Ainsi, pour reprendre l'exemple des comptes bancaires, le message `Retirer(Montant)` pourra activer des comportements bien distincts dans le cas d'un `CompteCourant` (où il est possible de faire des retraits même lorsque le solde est négatif) et dans un `CompteEpargne` (pour lequel le solde doit toujours rester positif). Le message `Retirer(Montant)` est donc un message polymorphe. Cette propriété est très intéressante, non seulement pour limiter l'inflation des identificateurs de procédures dans un programme, mais surtout pour permettre une expression plus naturelle des traitements qui s'affranchissent ainsi des spécificités des objets concernés.

La surcharge est un concept fort voisin de celui de polymorphisme. Il s'agit cette fois de permettre la redéfinition d'un comportement associé à un message à plusieurs niveaux de la hiérarchie d'héritage afin de raffiner progressivement les caractéristiques d'un comportement au fur et à mesure de la spécialisation des classes. Au besoin, et toujours pour préserver la non-redondance, il est possible à un comportement de faire lui-même appel au comportement de même nom de sa superclasse et de ne spécifier que les actions vraiment spécifiques à la classe courante. Toujours en référence aux comptes bancaires, si le comportement `Retirer(Montant)` est décrit au niveau de la superclasse `Compte`, il peut être surchargé dans `CompteEpargne` à l'aide d'un algorithme du type suivant :

`Si` le `Solde()` est suffisant `alors`
  `Retirer(Montant)` comme spécifié dans la superclasse.

Enfin, la généricité consiste à décrire un comportement à un certain niveau de la hiérarchie d'héritage en faisant appel à des messages polymorphes qui ne sont décrits que plus bas dans la hiérarchie. Ainsi, par exemple, il est possible de décrire dans la classe `Compte` un message `Virement(Montant, Bénéficiaire)` en faisant appel à l'opération `Retirer(Montant)` alors que celle-ci n'est pas (encore) définie, mais en sachant qu'elle le sera certainement dans une classe descendante et qu'elle le sera de façon distincte suivant la classe descendante. Toutefois, il est bien clair qu'une classe comportant des comportements génériques ne peut être instanciée, car tous les comportements nécessaires ne sont pas disponibles. La généricité est donc également un outil important de lutte contre la redondance au niveau des algorithmes.

3. LANGAGES DE PROGRAMMATION ET PARADIGME OBJET

Les grands traits du paradigme objet étant ainsi brossés, il convient de s'intéresser aux possibilités qui sont offertes à l'informaticien pour mettre en oeuvre ces principes. Comme presque toujours, les environnements de programmation offrent différentes solutions de compromis que nous classerons en trois catégories correspondant à trois niveaux d'intégration du paradigme objet.

La première catégorie, que nous nommerons *langages à objets* est certainement la plus "pure" et correspond à des langages dans lesquels objets et classes sont les concepts premiers et incontournables : tout doit y être exprimé par des objets. SMALLTALK est certainement le langage le plus représentatif de cette catégorie tandis qu' Eiffel en est un autre exemple.

Vient ensuite une seconde catégorie de langages, dits *langages orientés objets* ou encore langages hybrides qui sont en fait des langages "classiques" ayant intégré une portion très significative du paradigme objet. On y trouve donc la possibilité de manipuler des variables et des procédures traditionnelles, mais aussi des objets, des classes ainsi que la possibilité de mettre en oeuvre l'héritage. SIMULA et C++ sont certainement les chefs de file de cette catégorie où l'on trouve également TurboPascal 5.5 et suivants, Clipper 5 muni de librairies telles que Class(y), ...

Enfin, une troisième catégorie de langages, que nous proposons de nommer *langages avec objets* consiste en des langages impératifs classiques dans lesquels une série de classes prédéfinies ont été intégrées et qui offrent donc essentiellement la possibilité d'instancier ces classes. Il n'y a cependant que peu ou pas de possibilités de définir des classes nouvelles ni d'utiliser l'héritage. Visual Basic, HyperCard, Clipper 5 (natif) sont des exemples de langages de cette troisième catégorie.

4. PARADIGME OBJET VERSUS PARADIGME IMPERATIF

Nous voudrions à présent développer quelques argumentations visant à situer le paradigme orienté objet vis-à-vis du paradigme impératif et plus généralement de l'approche traditionnelle de la conception d'application et de la programmation.

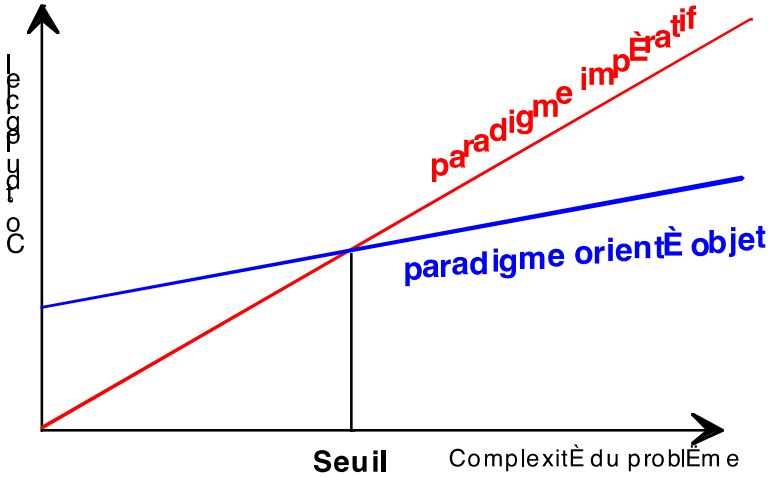
a. Tâche ou objet

Tout d'abord, il est évident que l'approche classique se fonde sur une analyse fonctionnelle du logiciel, c'est-à-dire sur une découpe en tâches tandis que l'approche objet est elle beaucoup plus centrée sur les données. Or il apparaît que les structures de données des organisations sont plus stables dans le temps que les tâches à effectuer sur ces données. De plus, les structures de données évoluent le plus souvent par simple ajout de données nouvelles tandis que les tâches sont volontiers refondues et réorganisées. L'ajout de données nouvelles peut être facilement pris en compte par l'héritage et les modifications des comportements par le mécanisme de surcharge alors que dans une approche traditionnelle, il s'avère hélas souvent nécessaire de revoir en profondeur certaines procédures. De ce point de vue, l'approche objet offre donc plus de garanties de stabilité, d'adaptabilité et de possibilités de capitaliser les développements antérieurs.

b. "Top-down" ou "bottom-up"

Une seconde caractéristique qui tend à opposer les paradigmes impératifs et objet réside dans le sens de progression dans l'analyse du logiciel. On sait que l'approche traditionnelle, souvent qualifiée d'analyse descendante et structurée est par essence "top-down". La tâche est ainsi décomposée en sous-tâches plus simples jusqu'à ce que ces tâches deviennent élémentaires. Dans le cas d'une analyse structurée pure, il n'y a, en fait, pas de souci de réutilisation et deux tâches semblables seront en principe analysées séparément créant ainsi des algorithmes fortement redondants. On sait que dans la pratique, on veille souvent à réduire cette redondance et partant à économiser ainsi du travail d'analyse et d'encodage.

L'analyse orientée objet est, quant à elle, fortement orientée "bottom-up". Bertrand MEYER ne dit-il pas que "il n'y a pas de raison théorique de limiter le nombre de primitives d'une classe" [MEYER 90]. Aussi, il encourage vivement le concepteur à doter chaque classe de tous les services qui sont potentiellement utiles à son exploitation sans se limiter à ceux qui seront effectivement utilisés dans le système en cours de développement. Cette position se comprend mieux lorsque l'on se rappelle l'objectif de réutilisation du paradigme objet. Tout développement contribue à construire, pas à pas, un modèle orienté objet de l'organisation utilisatrice. Aussi, des services inexploités ici seront probablement mis en oeuvre dans un autre logiciel qui réutilisera la même classe. La meilleure stratégie réside probablement dans un certain compromis entre les deux approches descendante et ascendante [DELACHARLERIE 93].



c. Coût de développement

On attire souvent l'attention sur les possibilités de réutilisation offertes par le paradigme orienté objet en les opposant aux difficultés bien connues que l'on rencontre avec l'approche traditionnelle. Pour comparer les coûts qui sont du reste très difficiles à évaluer (temps de développement, lignes de code...), on peut cependant faire l'hypothèse assez réaliste que dans le cas de la programmation impérative et de l'analyse structurée le coût du logiciel est relativement proportionnel à la complexité du problème à traiter. Dans le cas d'une approche orientée objet, il semble judicieux de diviser les coûts en deux parties que nous nommerons coûts fixes et coûts variables. En effet, si, comme le soulignent les chantres du nouveau paradigme, la réutilisation permet de réduire les coûts variables liés au raffinement des classes, on oublie trop souvent de considérer que la création des classes "de base" de

l'application représente un coût de départ non négligeable. Le coût d'un développement orienté objet n'est donc souvent bénéficiaire que si la taille du problème dépasse un certain seuil qu'il est évidement très difficile de quantifier.

Toutefois, pour nous qui envisageons la mise en oeuvre du paradigme objet dans un cadre pédagogique, il est probable que pas mal de "problèmes" que nous voudrions étudier en classe se révèlent trop "étriqués" pour que l'approche objet soit réellement intéressante et que nous donnions par cela une image faussée de cette approche.

d. Programmation événementielle

Le domaine dans lequel le paradigme orienté objet semble le plus adapté réside très certainement dans le développement des environnements informatiques modernes régis par l'occurrence d'événements et non plus par un processus unique. Les environnements graphiques tels que ceux offerts par Windows, OS/2 ou le Macintosh en sont de beaux exemples. De plus, l'évolution marquée vers les systèmes multi-tâche et l'intégration des ordinateurs personnels dans des réseaux de plus en plus complexes contribue à promouvoir un style de programmation que l'on peut appeler événementiel.

Il est évident qu'ici aussi, le paradigme objet apporte une approche beaucoup plus naturelle pour concevoir et réaliser des logiciels ayant à réagir à de multiples événements dont on ne peut connaître à l'avance l'ordre d'occurrence.

e. Objets et logiciels bureautiques

Enfin, il nous paraît intéressant de montrer que les logiciels de bureautique, beaucoup plus prisés aujourd'hui que les environnements de programmation ont, eux aussi, une nette orientation vers les objets. En effet, les traitements de texte, tableurs et autres systèmes de gestion de bases de données structurent très souvent les données manipulées par l'utilisateur en objets ayant de nombreuses propriétés modifiables : ce sont entre autres les objets paragraphes, sections, pages, cellules, cadres, tables, tableaux... Ici aussi, la justification est d'ordre ergonomique : il est, par exemple, plus facile à un utilisateur de penser que le texte d'un paragraphe doit être justifié à droite plutôt que de penser le travail en termes de tâche en désignant les points du texte à partir desquels le mode de justification doit être modifié.

Ces objets ne sont cependant pas de "vrais objets" au sens où nous les avons définis au début de ce papier. Ils s'en rapprochent cependant de plus en plus, car il devient à présent possible de modifier les comportements attachés à ces "pseudo-objets" grâce, entre autres, à des macros. C'est, par exemple, le cas des formulaires et états que l'on peut définir dans le logiciel Access et qui peuvent ainsi réagir de façon ciblée à différents événements.

5. VERS LA MORT DU PARADIGME IMPERATIF ?

Nous venons de montrer que les arguments en faveur du paradigme orienté objet sont nombreux et il ne paraît guère douteux que cette nouvelle approche tendra progressivement à se généraliser tant au niveau des environnements de programmation que dans les outils bureautiques.

Annoncer la mort du paradigme impératif n'en serait pas moins une aberration monumentale et pour de nombreuses raisons. En effet, nous avons montré que l'approche classique reste souvent plus facile à mettre en oeuvre sur des problèmes de petite taille. De surcroît, cette façon de penser est considérée comme "naturelle" par une large majorité des informaticiens actuels qu'ils soient professionnels ou non. On connaît en effet le poids du passé et des habitudes dans tous les domaines et dans celui-là en particulier.[2]

De toute façon, tant que les ordinateurs seront construits sur les principes de von Neumann, le paradigme impératif restera le plus proche du fonctionnement interne de la machine et le plus apte à en exploiter au mieux les performances.

Enfin, et c'est probablement la raison la plus fondamentale, le paradigme impératif reste sous-jacent au paradigme objet, car les algorithmes des opérations doivent finalement être exprimés comme un ensemble structuré d'ordres. Aussi, prétendre connaître le paradigme objet sans être capable de programmer de façon impérative s'avère tout simplement impossible.

Dès lors, plutôt que d'affronter les deux paradigmes afin de désigner celui qui serait le "meilleur", il y a plutôt lieu de les situer dans le prolongement l'un de l'autre et de mettre en évidence leur complémentarité. Le paradigme impératif relève plutôt du niveau de l'implémentation du logiciel tandis que le paradigme objet devrait d'abord être vu comme un outil au service de la conception et de l'organisation du logiciel.

6. PARADIGME OBJET ET ENSEIGNEMENT

Au terme de cette réflexion, on ne peut éviter de se reposer la question de la place à attribuer au paradigme objet dans notre enseignement et en particulier dans le cadre des formations à l'informatique au niveau secondaire. Trancher la question est évidement impossible, car les arguments que l'on peut avancer sont divergents.

D'une part, il semble clair que l'évolution vers le paradigme objet est irréversible, mais d'autre part, il faut bien reconnaître que la mise en oeuvre des langages orientés objets tels que TurboPascal sur des problèmes simples se révèle souvent complexe et rébarbative.

Quoi qu'il en soit, il sera toujours possible à l'enseignant d'adopter une "pensée orientée objet" dans son approche des problèmes, même si les outils qu'il emploie restent tout à fait classiques. En fait, bien plus que l'emploi d'un langage ou d'un autre, c'est l'éveil aux grands

[2] On sait en effet que COBOL et FORTRAN restent les deux langages les plus "employés" même si de nombreux langages plus modernes et performants sont disponibles aujourd'hui.

principes que nous avons rappelés en début d'article qui s'avère le plus important et le plus formateur.

BIBLIOGRAPHIE

CLOUTIER J.F., *Apport de différents paradigmes de programmation comme autant d'outils de pensée* in Actes du colloque francophone sur la didactique de l'informatique., Université René Descartes, Paris, 1, 2, 3 septembre 1988., Editions de l'Epi, Paris, 1989, pp. 195-204.

DELACHARLERIE A., *Vers une méthode de conception orientée objet applicable aux développements xBase*, Institut d'Informatique, FNDP, Namur, 1993.

FERBER J., *Conception et programmation par objets*, Hermes Publishing, Paris, 1991.

MASINI G., NAPOLI A., COLNET D., LEONARD D., TOMBRE K., *Les langages à objets*, InterEditions, Paris, 1991.

MEYER B., *Conception et programmation par objets : Pour du logiciel de qualité*, InterEditions, Paris, 1990.

TAYLOR D.A., *Technologie orientée objet : Le guide du décideur*, Addison-Wesley France, Paris, 1993.