Isima 2
Le modèle objet

OBJECTIFS

Ce chapitre présente une introduction à la notion de génie logiciel et tente un état de l'art sur les concepts du modèle objet. L'approche orientée objet permet une forte analogie entre la réalisation d'un modèle de simulation et le développement par objets d'un logiciel. On étudiera dans ce chapitre quelques éléments permettant de mieux appréhender la complexité des logiciels et de comprendre quels sont les buts du génie logiciel. Puis, on parlera de l'évolution de l'informatique vers les technologies à objets. Enfin, les principaux concepts du modèle objet seront étudiés en s'appuyant sur la relation classe/instance avec les notions d'encapsulation, d'héritage et d'envoi de messages polymorphes.

1 Introduction

Nous avons montré que nous souhaitons utiliser une approche orientée objet aussi bien pour l'analyse et la conception de modèles de systèmes complexes que pour la conception de logiciels d'animation graphique de résultats de simulation. Nous étudions les principaux concepts du modèle objet. Ces concepts découlent, dans leur majorité, de techniques de programmation pour la réalisation de logiciels complexes de qualité, que ce soit des logiciels de simulation [DAHL 66] ou des logiciels généraux [BOOCH 87] [MEYER 88]. Cox précise que l'évolution du génie logiciel permet, grâce aux techniques à objets, l'accès au stade industriel du développement des logiciels [COX 86].

Un système logiciel développé en utilisant les concepts de l'approche orientée objet peut être considéré comme un système d'objets communiquant par messages, chaque objet étant responsable de l'accomplissement de certaines tâches qui dépendent de son état. Cette vue d'un système logiciel est similaire à celle que l'on utilise avec la simulation à événements discrets pour modéliser les différentes entités d'un système réel et pour décrire leurs interactions au cours de la simulation. Il existe donc une forte analogie entre le développement par objets d'un logiciel et la réalisation d'un modèle de simulation.

2 Eléments de génie logiciel

2.1 La complexité du logiciel

Le développement de logiciels peut se révéler très complexe. La majorité des applications industrielles demandent un travail en équipe souvent difficile à gérer. Les principaux ouvrages constituant des références en génie logiciel pour l'approche orientée objet sont ceux de Meyer, Booch et Sommerville [MEYER 88,90], [BOOCH 87,91,92] et [SOMMERVILLE 88,92]. Nous tentons ici une synthèse rapide des principaux concepts présentés par ces auteurs. Selon Grady Booch, la complexité des grands logiciels découle principalement de quatre éléments [BOOCH 91] :

- l'étendue du domaine du problème à traiter,
- la difficulté à gérer le processus du développement,
- les contraintes de souplesse requises pour le logiciel final,
- les problèmes induits par le recensement et l'analyse fine des caractéristiques du système logiciel que l'on conçoit.

Cette complexité du logiciel est la cause des nombreux défauts reprochés aux logiciels, et ce, malgré le travail d'importantes équipes de développement. Les principaux reproches effectués sont les suivants [BOOCH 91] [MEYER 88] [SOMMERVILLE 92] :

- les logiciels conçus ne correspondent pas aux besoins et sont livrés en retard,
- les coûts de développement sont excessifs, et les coûts de maintenance (évolutive ou curative) sont encore plus importants (difficultés, introduction de nouvelles erreurs,...).
- les logiciels sont difficilement portables et n'utilisent pas efficacement les ressources disponible, de plus ils se bloquent fréquemment sur des erreurs de programmation.

2.2 Les Buts du Génie Logiciel

Le Génie Logiciel tente de maîtriser les problèmes liés à la complexité du logiciel et, par là même, à réduire les coûts de développement et de maintenance qui sont souvent prohibitifs (figure II.1). Une enquête présentée dans [BOOCH 91] a montré que pour 6,8 millions de dollars correspondant à 9 ensembles de projets, la répartition des coûts est donnée figure II.1 :

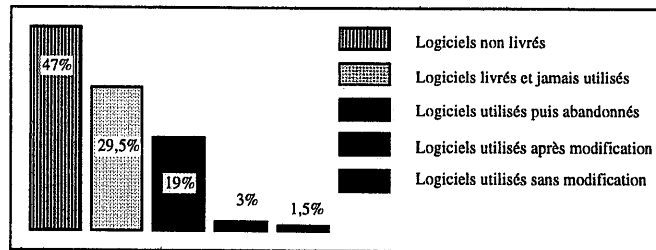


Figure II.1 : Répartition des coûts sur 9 projets (6,8 millions de dollars)

Cette enquête a également montré que sur les différents ensembles de projets, les coûts de maintenance sont situés entre 45% et 75% du coût total du logiciel. La maintenance consiste aussi bien à réaliser des adaptations qu'à corriger les diverses erreurs de conception ou d'implémentation. Les techniques du Génie Logiciel doivent donc proposer un ensemble de solutions permettant de maîtriser la phase de conception tout en prenant en compte la maintenance future.

Booch et Sommerville expriment les buts du Génie Logiciel à l'aide de quatre propriétés [BOOCH 87], [SOMMERVILLE 88], qui sont la modifiabilité, l'efficacité, la fiabilité et l'intelligibilité. Nous les examinons sommairement.

La modifiabilité : un système logiciel est susceptible d'être modifié pour l'une des deux raisons suivantes :

1 Répondre à un changement dans les spécifications d'un système,
2 Corriger une erreur introduite antérieurement dans le processus de développement.

Pour pouvoir modifier efficacement un logiciel, il faut être capable d'assimiler toutes les décisions de conception qui ont été prises, sans quoi les modifications ne sont que de vulgaires raccommodages aveugles et indépendants des choix originels. Pour être jugé "modifiable", un logiciel doit également autoriser des changements sans que cela n'augmente sa complexité globale.

L'efficacité : elle implique que le système conçu exploite au mieux les ressources disponibles. Ces ressources sont classées en deux groupes : les ressources en temps et les ressources en espace correspondant respectivement à l'utilisation de la puissance de calcul et à l'utilisation de la mémoire disponible.

La fiabilité : elle est primordiale pour tout système logiciel. La fiabilité du logiciel final doit être prise en compte dès le début de la conception, elle ne peut être "rajoutée". Il faut donc non seulement éviter les défauts de conception, mais également être en mesure d'effectuer les reprises nécessaires lors de dysfonctionnements logiciels (tolérance aux pannes).

L'intelligibilité : il s'agit de rendre le logiciel compréhensible grâce à différents facteurs situés à plusieurs niveaux. Au plus bas niveau, le logiciel doit être lisible grâce à un bon style de codage. A un niveau plus élevé, nous devrions être capables d'isoler facilement les données et les opérations à effectuer sur ces données. L'intelligibilité dépend fortement du langage de programmation choisi pour implémenter le logiciel.

Les principes qui viennent d'être énoncés ne sont pas les seuls critères de qualité que l'on trouve dans la littérature. Bertrand Meyer [MEYER 88] propose un ensemble de facteurs externes et internes pour quantifier la qualité d'un logiciel.

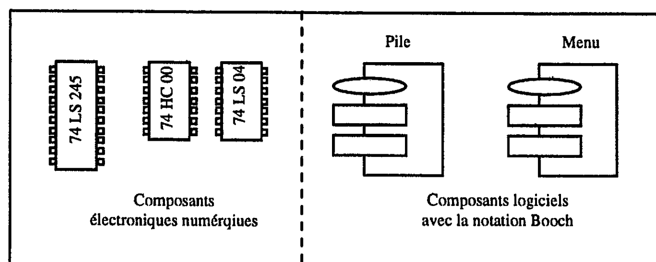
Les facteurs externes sont ceux détectés par l'utilisateur du produit, tels que la facilité d'utilisation, la vitesse d'exécution ou encore la facilité d'adaptation à des changements de spécification. Les facteurs internes correspondent à ceux perçus par les informaticiens, tels que la lisibilité, la modularité... Si les facteurs internes sont les plus importants, ils ne peuvent être pris en compte que grâce aux facteurs externes.

Bertrand Meyer propose un ensemble de qualités que devrait posséder un logiciel dont cinq sont essentielles, à savoir : la validité par rapport à la spécification, la robustesse (utilisation dans des conditions anormales), l'extensibilité, la réutilisabilité et la compatibilité qui offre la possibilité de combiner le logiciel avec d'autres. D'autres qualités spécifiées par Bertrand Meyer mais jugées moins fondamentales sont l'efficacité déjà énoncée plus haut, la portabilité, la vérifiabilité (aptitude aux tests), la facilité d'utilisation et l'intégrité (protection des informations manipulées) [MEYER 88].

Ces qualités ne sont pas toutes compatibles entre elles. Il convient donc d'établir des compromis en définissant de manière explicite le poids accordé aux facteurs internes et externes. Parmi les qualités énoncées, il nous semble que la réutilisabilité est fondamentale, c'est pourquoi nous l'étudions dans le paragraphe suivant.

2.3 La réutilisabilité et la notion de composant logiciel

Tout développeur reconnaîtra avoir codé de nombreuses fois les mêmes algorithmes à cause de certaines variantes qui ne pouvaient pas être paramétrées dans le langage de programmation utilisé. Le cas se présente fréquemment lors d'algorithmes de manipulation de structures de données telles que les tables, les listes, les arbres ou les tables hachées dynamiques. Les algorithmes qui gèrent ces structures sont souvent réécrits pour gérer des entités de types différents. De même, il existe des algorithmes qu'il faut modifier à chaque adjonction d'un nouveau type abstrait dans le logiciel.



II.2 Composants matériels et logiciels avec leurs interfaces

Les algorithmes fondamentaux connus sont recensés dans des ouvrages tels que [KNUTH 75], [SEDGEWICK 85] ou encore [AHO 89]. Cependant ils sont inlassablement réécrits car la majorité des langages qui les implémentent ne permet pas d'accéder à un degré d'abstraction suffisant. De plus, des obstacles économiques ralentissent la diffusion de composants réutilisables [MEYER 88].

L'analogie classique, qui consiste à proposer la notion de composants logiciels qui soient aussi largement utilisables que le sont les composants électroniques, est détaillée par Brad Cox [COX 86] (Figure II.2). Pour que ces bibliothèques de composants soient utilisées, il faudrait qu'elles soient aussi connues que les "data book" des composants électroniques. Ainsi un utilisateur (client potentiel) pourrait consulter les caractéristiques des composants logiciels et choisir celui qui correspond à son besoin. Contrairement à leurs homologues électroniques, il serait alors possible de télécharger les composants logiciels désirés. Il faudrait également pouvoir proposer des gammes de prix adaptées et concevoir des protections pour ce type d'utilisation soumise aux problèmes de copies informatiques.

Si la réutilisation du code source des composants est envisagée, il en est de même pour les travaux relatifs à l'analyse et à la conception. Les techniques du Génie Logiciel préconisent une cohérence totale entre l'analyse et l'implémentation. C'est un principe très louable mais dont l'application est généralement utopique. S'il est nécessaire, lors de la conception, d'appréhender l'architecture d'un projet de façon claire et cohérente, il faut reconnaître que les analyses reposent en majorité sur une documentation extérieure au code du logiciel. L'idéal serait de pouvoir effectuer la phase de conception dans le code du logiciel lui-même : on garantirait ainsi la cohérence entre la conception et l'implémentation. Cette démarche commence à être réalisable avec les langages à objets et de nombreux auteurs souhaitent maintenant aboutir à la généralisation de langages qui autorisent la présence d'un maximum d'éléments de conception dans le code source d'un logiciel [DAHL 66] [COX 86] [MEYER 88].

Un des autres avantages concernant la réutilisation du code, du travail de conception ou d'analyse, est l'augmentation du degré de fiabilité. Plus on utilise un élément, plus les opportunités de découvrir des erreurs sont grandes. Les éléments que l'on réutilise ont en effet déjà subit des tests, les coûts de maintenance sont donc susceptibles de diminuer.

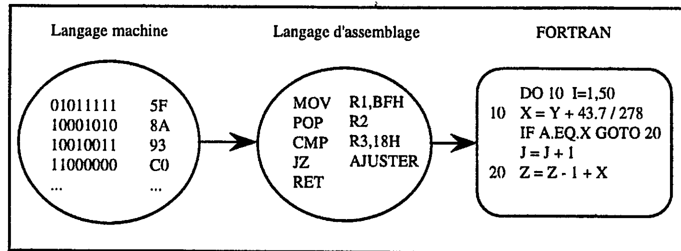
Nous allons maintenant présenter le processus d'évolution qui a conduit au concept d'objet. Tout comme Budd [BUDD 91], nous choisissons de suivre progressivement le passage des sous-programmes aux modules, aux types de données abstraits pour aboutir aux objets.

3 L'évolution vers les objets

3.1 Les origines

L'origine des langages de programmation nous ramène aux langages machine puis aux langages d'assemblage avec lesquels la majorité des premiers programmes étaient écrits par une seule personne et dont la taille dépassait rarement quelques dizaines de milliers de lignes (figure II.3). Pour résoudre les problèmes induits par des développements de taille plus conséquente, les premiers langages dits évolués furent introduits (COBOL, FORTRAN,...). Les possibilités en matière de développement se sont ainsi accrues, et les besoins suivirent tant et si bien que les problèmes à résoudre ne pouvaient plus l'être que par des équipes de développeurs, réunissant leurs efforts pour aboutir à la solution escomptée. Le travail de groupe introduisit alors un nouveau degré de

complexité dans la conception et la réalisation des systèmes logiciels. En effet, il est de notoriété publique que dix programmeurs ne travaillent pas dix fois plus vite qu'un seul programmeur. Le travail de groupe introduit des problèmes complexes de communication aussi bien au niveau humain qu'au niveau des protocoles de connexion des divers composants logiciels écrits par les différents membres d'un groupe. L'outil choisi pour aborder ce type de difficulté est l'abstraction. Il s'agit de la faculté que possède l'être humain d'occulter les caractéristiques d'un problème qui ne sont pas significatives pour sa résolution afin de se concentrer sur celles qui le sont.



II.3 Les premières évolutions des techniques de programmation

3.2 Les sous-programmes

Le premier niveau d'abstraction utilisé fut la notion de sous-programme (ou encore procédure, routine, fonction,...). Le mécanisme des sous-programmes a permis de gérer de manière simple et efficace toutes les tâches exécutées de manière répétitive. Les sous-programmes furent l'un des premiers moyens pour, d'une part, structurer un programme et d'autre part, effectuer "un masquage d'information". En effet un programmeur peut utiliser un sous-programme écrit par un autre programmeur en ne connaissant que le nom et les paramètres de ce sous-programme. Il était devenu inutile de connaître les détails de l'implémentation. Ce mécanisme de décomposition est utilisé depuis Descartes, qui conseillait déjà de diviser les difficultés en autant de parcelles qu'il serait nécessaire pour les résoudre. Néanmoins, Timothy Budd met en évidence au moins une des raisons essentielles de l'insuffisance du niveau d'abstraction des sous-programmes : les problèmes de visibilité de certains identificateurs ne sont que partiellement résolus [BUDD 91]. En effet, supposons qu'un développeur désire mettre au point un ensemble de fonctions pour manipuler une pile.

L'exemple de la pile a été très souvent repris (Booch, Meyer, Cox, Budd,...) car il permet d'exploiter de manière synthétique différentes notions. Notre développeur aura donc écrit un ensemble de routines de gestion d'une pile (pour empiler, dépiler, vider la pile,...). Ces routines doivent toutes accéder à la pile ainsi qu'à son pointeur, qui doivent, pour des raisons évidentes d'efficacité, être déclarés comme des variables globales. Or, si ces données sont globales, elles deviennent accessibles à partir des autres parties du programme : la protection des données n'est donc plus assurée.

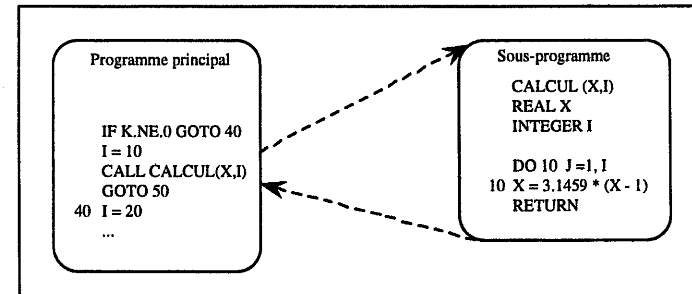


Figure II.4 : Mécanisme d'appel et de retour d'un sous-programme

3.3 Les modules

La notion de module a été introduite pour résoudre le problème évoqué précédemment. Un module est dissocié en une partie visible dite "publique" et une partie "privée". La partie publique, appelée couramment **interface**, est accessible de l'extérieur du module tandis que la partie privée n'est accessible que de l'intérieur du module. Les techniques pour déterminer les modules et leurs interfaces sont clairement expliquées par David Parnas dans [PARNAS 72].

Les modules répondent effectivement au problème du masquage de l'information et des détails d'implémentation. Il est possible de créer un module de gestion de pile où les données sont masquées pour l'extérieur du module qui ne possède que l'accès aux procédures de l'interface. Cependant, il peut être intéressant de disposer de plusieurs piles, mais les modules ne donnent pas la possibilité de dupliquer leurs zones de données.

Un des dangers associé à la notion de modules est celui d'une décomposition en modules trop importante d'un système logiciel, conduisant à une augmentation de la complexité de ce logiciel. Il existe donc, pour tout problème, un niveau de décomposition "optimal". Ce niveau doit non seulement prendre en compte le nombre de modules, mais aussi le couplage entre les différents modules.

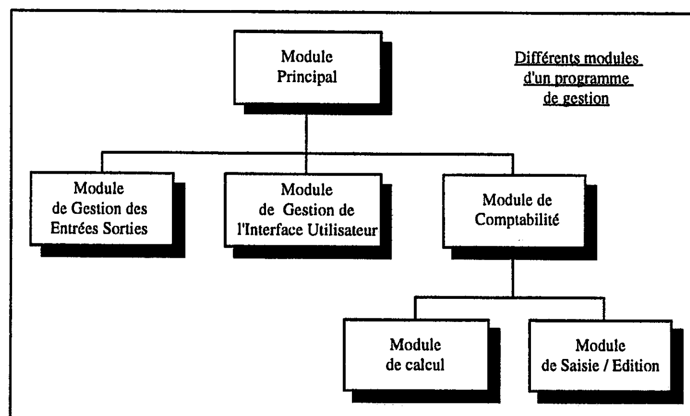


Figure II.5 : Un découpage en module utilisant uniquement la programmation structurée

3.4 Les types de données abstraits

Les types de données abstraits sont une extension du concept de modules car ils autorisent une duplication des zones de données. Le développeur conçoit un nouveau type avec ses différentes opérations associées, ce qui permet au programmeur de manipuler ce type abstrait de la même manière qu'un type prédéfini. Prenons l'exemple classique de la manipulation des nombres complexes. Un programmeur décide de décrire un nouveau type "complexe" comme étant composé de deux valeurs réelles (figure II.6). Il souhaite ensuite implémenter les opérations d'addition, de soustraction, de multiplication, de division,... Seules les opérations précisées dans l'interface du type de données abstrait pourront manipuler les données d'un nombre complexe : les données sont

donc protégées. Il devient également possible, avec ce type abstrait complexe, de déclarer une multitude de variables du type "complexe" car, non seulement les types abstraits définis sont alors connus du système, mais il est possible de réaliser des copies de la zone de données d'un type abstrait. Toutes les nouvelles variables sont des **instances** du type "complexe" qui partagent les opérations définies sur le type de données abstrait, chacune possédant sa zone de données protégée. Les types de données abstraits ont été implémentés par exemple avec la notion de "package" dans le langage ADA [BOOCH 87] [BARNES 88].

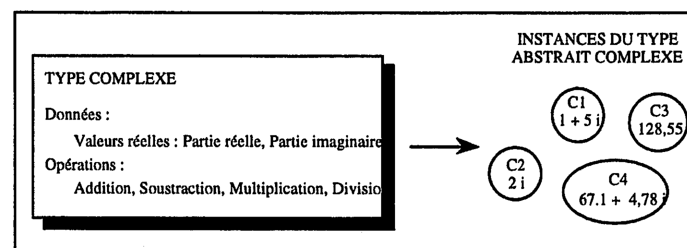


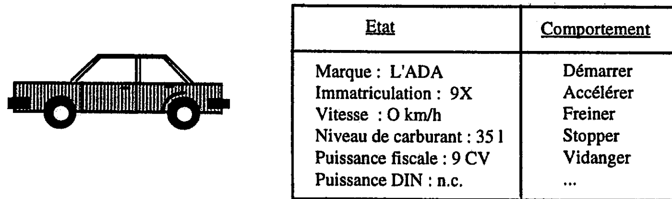
Figure II.6 : Exemple d'instances d'un type de données abstrait

3.5 La notion d'objet

Les prémisses de la notion d'objet datent du projet du missile Minuteman en 1957 [TEN DYKE 89]. La conception et la simulation du fonctionnement de ce missile reposaient sur une poignée de composants logiciels prenant en charge la partie physique du missile, les trajectoires, les différentes phases de vol,... Le fonctionnement du système global reposait sur l'échange de messages d'information entre les différents composants. Chaque composant logiciel était conçu par un spécialiste et possédait ses données privées. De même le composant était virtuellement isolé du reste du programme par l'ensemble de ses méthodes servant d'interface.

Au cours des années soixante, le besoin de trouver des solutions plus générales aux problèmes de simulation, a conduit Ole-Johan Dahl et Kristen Nygaard à introduire les langages SIMULA et SIMULA67, qui proposent et implémentent un ensemble de concepts révolutionnaires à l'époque [DAHL 66]. Ces concepts ne seront réellement exploités que beaucoup plus tard. Le premier concept proposé est celui de l'unification des données et des codes en une **classe**

d'objets. Le second concept, avancé par Dahl et Nygaard, consiste à dissocier une classe de ses instances. Une classe forme un modèle pour la création d'instances qui ne sont que des représentations individuelles de ce modèle. Le troisième concept se rapporte à l'organisation des classes entre elles (il se nommait préfixage). Le préfixage autorise la mise en commun des caractéristiques de certaines classes pour d'autres classes en utilisant un **graphe d'héritage**. Les années soixante-dix virent apparaître le langage Smalltalk au PARC de Xerox Corporation. Alan Kay, imprégné de Simula, formalisa dans Smalltalk la notion de **message** comme étant la seule technique d'interaction entre objets [GOLDBERG 83].



Etat	Comportement
Marque : L'ADA	Démarrer
Immatriculation : 9X	Accélérer
Vitesse : O km/h	Freiner
Niveau de carburant : 35 l	Stopper
Puissance fiscale : 9 CV	Vidanger
Puissance DIN : n.c.	...

Figure II.7 : Un objet de la classe voiture

La notion d'objet complète la notion de type abstrait de données en plusieurs points. Un objet **encapsule** un **état** (la valeur de ses données ou **attributs**) et un comportement (ses opérations ou **méthodes**) (Figure II.7). Une approche orientée-objet ne se limite pas à un ensemble de nouveaux concepts ; elle propose plutôt une nouvelle manière de penser qui conduit à un nouveau processus de décomposition des problèmes. Cette approche s'oppose à une approche classique qui donne la priorité aux fonctions d'un logiciel plus qu'aux données. L'expérience montre que, généralement, au cours de l'évolution d'un logiciel, les données manipulées par le logiciel sont plus stables que les traitements qui leur sont associés. L'approche orientée-objet donne donc la priorité aux données manipulées par le logiciel. Avec cette approche, les problèmes abordés peuvent être modélisés par une collection d'objets qui prennent chacun en charge une tâche spécifique. La résolution du problème est conduite par la manière dont interagissent les différents objets.

Les états et les comportements possibles des objets sont définis par des **classes** qui sont des modèles pour la construction des objets. Chaque objet constitue une **instance** d'une classe. Toutes les instances d'une même classe possèdent les mêmes caractéristiques et les mêmes réactions. Chaque instance est identifiée par la valeur de ses attributs (Figure II.8).

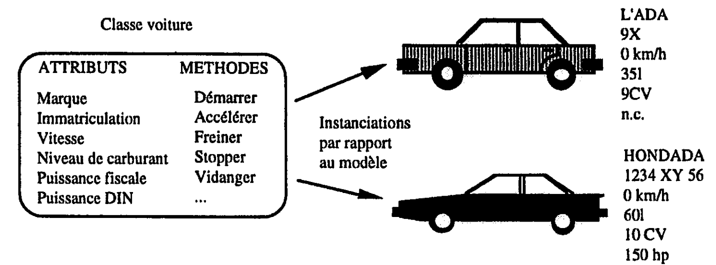


Figure II.8 : Création de deux instances d'une même classe

4 Les concepts du modèle objet

4.1 Introduction

Nous venons de voir que l'approche orientée objet est issue de la lente évolution des mécanismes d'abstraction. Elle apparaît comme un concept unificateur des approches guidées par les données et de celles basées sur une décomposition fonctionnelle. L'approche orientée objet s'applique à de nombreux domaines de l'informatique, de l'algorithmique pure aux bases de données en passant par le graphisme et les interfaces utilisateurs. Cependant l'approche orientée-objet est encore adolescente et s'accompagne d'une terminologie hétérogène parsemée de synonymes. Nous nous proposons de présenter les principaux concepts de cette approche ainsi que la terminologie associée.

4.2 L'abstraction

L'abstraction a été précédemment présentée comme l'un des outils fondamentaux que l'humanité s'évertue à utiliser pour affronter la complexité du monde qui l'entoure. Il convient de donner maintenant quelques définitions proposées par différents auteurs :

"Une abstraction dénote les caractéristiques essentielles d'un objet qui le distinguent des autres sortes d'objets en lui donnant des limites conceptuelles bien définies et ce relativement à la vision d'un observateur" [BOOCH 91].Ll

"Abstraction : principe consistant à ignorer les aspects d'un sujet qui ne sont pas significatifs pour l'objectif en cours de manière à se concentrer complètement sur ceux qui le sont" [COAD 91].

L'abstraction est donc une opération de l'esprit qui isole d'une notion un élément en négligeant les autres, elle se concentre sur la vue externe d'un objet de manière à séparer le comportement essentiel d'un objet de son implémentation. Les objets peuvent communiquer les uns avec les autres au moyen de messages. L'ensemble des messages que peut recevoir un objet constitue le protocole de cet objet. Ce protocole correspond totalement à la notion d'interface déjà présentée pour les types de données abstraits et pour les modules. Nous parlerons donc indifféremment de **protocole** ou d'**interface**. De même, les notions d'opérations (Ada [BOOCH 87] [BARNES 88]), de méthodes (Smalltalk [GOLDBERG 83]) et de fonctions membres (C++ [STROUSTRUP 86]) sont identiques et représentent les différents comportements de l'objet. Le fait de ne considérer que les comportements possibles d'un objet, sans se soucier des détails d'implémentation, correspond à l'utilisation de **l'abstraction procédurale** que l'on peut définir de la manière suivante :

"Abstraction procédurale : principe par lequel toute opération qui aboutit à un effet bien déterminé peut être considérée par ses utilisateurs comme une seule entité, en dépit du fait que cette opération peut se décomposer en un ensemble d'opérations de plus bas niveau" [COAD 91].

Cette forme d'abstraction n'est pas la seule utilisée par l'approche orientée objet qui se base également sur l'**abstraction de donnée**. En effet, si l'architecture d'un système peut être obtenue à partir des fonctions ou à partir des données, ce sont les données qui offrent une meilleure stabilité au cours du temps. L'expérience montre qu'une conception fonctionnelle descendante dite classique ne facilite pas les activités de maintenance [BOOCH 91] [COAD 91]. Bertrand Meyer donne de nombreux exemples, montrant que pour atteindre des buts tels que la réutilisabilité et l'extensibilité d'un logiciel, il est préférable de baser l'architecture d'un système sur les données du système [MEYER 88]. Celles-ci devant être protégées, voici une définition de l'abstraction de données :

"Abstraction de données : principe qui consiste à définir un type de données en terme d'opérations s'appliquant aux objets de ce type, avec la contrainte que les données propres de tels objets ne puissent être modifiées ou consultées que par le biais de ces opérations" [COAD 91].

Cette définition précise que lorsque l'on définit un objet par ses attributs

(données qui constituent son état) et les services qu'il peut nous rendre, le seul moyen d'accéder aux attributs consiste à utiliser les méthodes que l'objet nous propose. Ceci constitue, par ailleurs, un bon style de programmation au sens du génie logiciel. Il existe pour cela un ensemble complet de consignes et même une "loi" (la loi de Demeter) pour obtenir un style de codage orienté objet correct [LIEBERHERR 89] [BUDD 91].

4.3 L'encapsulation

L'encapsulation consiste à intégrer le code et les données d'une entité au sein d'un objet. L'encapsulation empêche également les utilisateurs clients d'un objet de connaître les détails de son implémentation en ne fournissant qu'une vue externe (correspondant à un masquage d'information). L'encapsulation peut être considérée comme une généralisation de l'abstraction de donnée ce qui ressort de la définition suivante (figure II.9) :

"L'encapsulation est le processus qui consiste à empêcher d'accéder aux détails d'un objet qui ne contribuent pas à ses caractéristiques essentielles" [BOOCH 91].

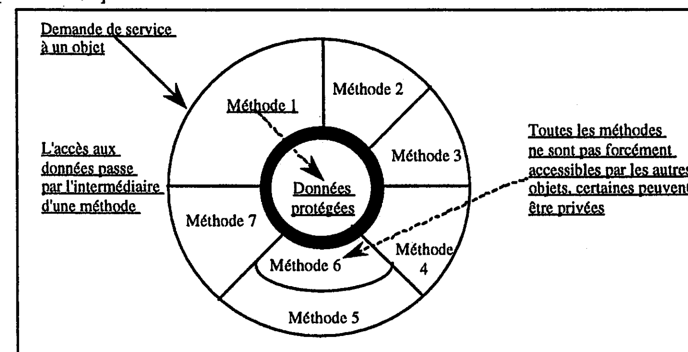


Figure II.9 : Représentation classique de l'encapsulation

Seule l'interface (ou protocole) d'un objet doit paraître visible aux yeux d'un client potentiel. David Parnas montre dans [PARNAS 72] comment l'encapsulation permet de minimiser les opérations de maintenance. En effet, les modifications sur les détails de l'implémentation n'ont aucune répercussion sur les clients de l'objet. Seul le changement de l'interface affecte les clients.

4.4 L'envoi de messages

L'envoi de messages est le seul mode de communication entre objets. Un message constitue une requête pour activer une des méthodes d'un objet. Le message doit correspondre à une partie du comportement de l'objet sans quoi une erreur se produit.

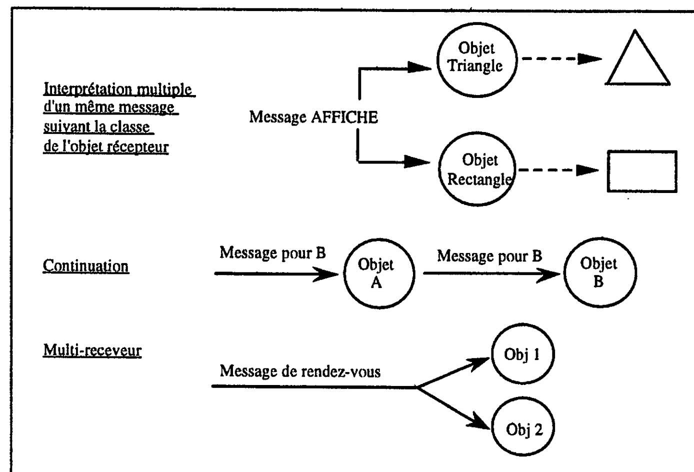


Figure II.10 : Différents modes de transmission de message

Prenons un exemple pour illustrer le type de situation que l'on peut rencontrer. Si un objet "pile" reçoit un message qui lui demande de dépiler, cet objet réagit au message en activant la méthode appropriée. Par contre, si le message dépiler est envoyé à un objet arbre, il ne peut bien sûr pas être traité. Les expressions "envoi d'un message", "demande de service" ou "activation d'une méthode" sont équivalentes pour spécifier qu'un objet va exécuter une procédure ou fonction faisant partie de son comportement. Après avoir abordé quelques concepts supplémentaires, nous détaillerons par la suite les points suivants (Figure II.10) :

- L'interprétation d'un message qui peut être différente suivant l'objet récepteur.
- La notion de continuation qui autorise l'envoi de la réponse à un message, à un objet autre que l'émetteur de la requête.
- La notion de multi-receveur qui précise plusieurs destinataires pour un message.

4.5 La modularité

Le fait de décomposer un logiciel en composants individuels permet de réduire sa complexité [PARNAS 72]. Mais, la modularité est aussi un des éléments clés de la réutilisabilité et de l'extensibilité. La décomposition en modules est prise en compte lors de l'analyse, puis lors de la conception, pour se répercuter enfin sur la programmation.

La synthèse des divers conseils recensés par Grady Booch dans [BOOCH 91] préconise la construction de modules aussi indépendants les uns des autres que possible. Chaque module doit regrouper le maximum de données liées au même type abstrait. Booch propose alors la définition suivante :

"La modularité est la propriété d'un système qui a été décomposé en un ensemble de modules cohérents et faiblement couplés" [BOOCH 91].

Bertrand Meyer propose cinq critères pour évaluer le degré de modularité des méthodes de conception. Ces critères sont la décomposabilité, la composabilité, la compréhensibilité, la continuité, et la protection [MEYER 90].

Bertrand Meyer présente également les notions d'ouverture et de fermeture de modules. Un module est dit fermé dès qu'il peut être utilisé par des modules clients. Un module est dit ouvert lorsqu'il peut encore être étendu. Les deux notions sont toutes deux nécessaires simultanément à la conduite de projets. Seule l'approche orientée-objet, avec les techniques d'héritage, autorise une ouverture / fermeture simplifiée des modules en cours d'élaboration, et ce, grâce aux faibles dépendances entre les différents objets.

4.6 Le typage

4.6.1 La notion de type et de classe

La notion de type est liée au concept de type de données abstrait. Le commun des mortels peut assimiler une classe à un type mais il existe des nuances que l'on peut consulter dans [BOOCH 91]. Nous considérons cependant par la suite, que les classes peuvent être vues comme des types. Grady BOOCH donne la définition suivante du **typage** :

"Le typage est l'affectation stricte d'une classe à un objet, de manière à ce que les objets de différents types ne puissent pas être interchangés, ou tout du moins qu'ils ne puissent être interchangés que de manière restrictive" [BOOCH 91].

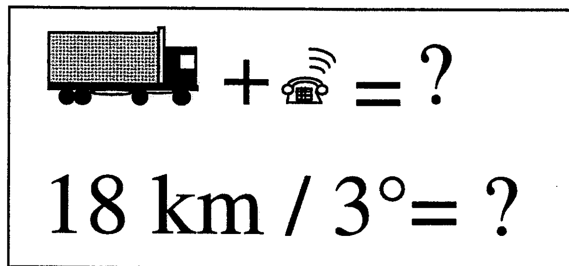


Figure II.11 : Un contrôle de cohérence peut être effectué grâce à un typage fort

Booch considère que le typage n'est qu'un élément mineur de la modélisation par objets. En matière d'implémentation, un langage fortement typé, faiblement typé ou non typé pourra toujours être un langage à objets.

Dans les langages fortement typés, les affectations ou les opérations qui ne seraient pas cohérentes avec les types des objets sont détectées à la compilation (par exemple affecter un objet Bateau à une variable de "type" Avion, ou encore additionner un nombre complexe à une chaîne de caractères) (figure II.11). Cette fonctionnalité est désirée pour des raisons de validité des logiciels. Cependant, de nombreux langages typés donnent la possibilité de faire des **enregistrements variants** pour manipuler des variables qui peuvent avoir différents types. Cette fonctionnalité, incluse dans la plupart des langages majeurs, montre un réel besoin de typage souple.

4.6.2 Le typage statique et le typage dynamique

Pour distinguer un langage à typage statique d'un langage à typage dynamique, il suffit de savoir si les types sont associés aux identificateurs des variables ou aux valeurs (représentant un contenu instantané d'une variable). Dans les langages à typage statique (Eiffel, Simula, C++,...), les types sont associés aux identificateurs au moyen d'instructions déclaratives ou lors d'une allocation dynamique. Par contre, dans les langages à typage dynamique (Smalltalk, Objective-C, CLOS...), les types ne sont pas associés aux variables mais aux contenus de ces variables. Chaque contenu est capable de déterminer de quelle classe il est, car il possède toutes les informations nécessaires. Avec ce concept, les références d'objets ne sont plus typées, il n'est donc plus possible d'effectuer des affectations illégales. Ceci facilite grandement la constitution de composants hautement réutilisables.

Cependant, le problème du choix entre un typage statique et un typage dynamique est lié à la réponse que l'on souhaite obtenir aux questions suivantes : A quel moment vérifie-t-on la cohérence du message "Activer la méthode M de l'objet O" ? Quand faut-il déterminer si l'objet O possède la méthode M ? Est-ce à la compilation ou à l'exécution ? Si cette vérification est effectuée à la compilation, le typage est statique ; si c'est à l'exécution le typage est dynamique. Dans le premier cas on apporte une plus grande sécurité, dans le second une très grande souplesse.

Les principaux arguments recensés par Booch et Meyer en faveur d'un typage statique restent cependant les suivants :

- Sans vérification de type à la compilation, de simples erreurs de programmation ne seront détectées qu'à l'exécution.
- Les déclarations de type servent à la documentation d'un logiciel.
- Les compilateurs peuvent générer un bien meilleur code si les types sont déclarés.

La souplesse perdue par l'utilisation d'un langage statiquement typé peut être compensée par la sécurité apportée, spécialement lors de développements de logiciels de taille importante, le domaine du prototypage restant encore très associé aux langages dynamiquement typés.

4.7 La classification

4.7.1 La notion de hiérarchie

La décomposition d'un système complexe peut conduire à de larges ensembles d'objets. Pour gérer ces objets de manière simple, il convient de déterminer les relations qui existent entre ces objets. Certaines de ces relations peuvent être abordées avec la notion de hiérarchie qui permet d'assurer un ordre à l'ensemble des abstractions obtenues. Les méthodes utilisées pour organiser un ensemble d'objets peuvent être indifféremment la décomposition ou la taxonomie, par analogie avec le monde biologique.

4.7.2 La généricité

La généricité est une technique conçue pour obtenir des composants logiciels réutilisables en permettant de garantir la cohérence des types. Elle est fréquemment utilisée pour manipuler des structures de données générales comportant des éléments de même type. La généricité provient du langage Algol 68 qui permettait de définir des modules où le type des entités manipulées est paramétré. Cette technique a été reprise par le langage Ada avec la notion de "package générique". On parle également du développement de classes paramétrées dans le langage Eiffel ou de la notion de "template" du langage C++. Les paramètres génériques effectifs sont des types simples ou des classes. Nous considérons que la généricité est une relation entre une classe paramétrée et ses instances qui sont elles-mêmes des classes (et non des objets). Ceci nous conduit à considérer les classes paramétrées comme étant en fait des **métaclasses paramétrées.** C'est principalement pour cette raison que nous plaçons la généricité parmi les techniques de classification (Figure II.12).

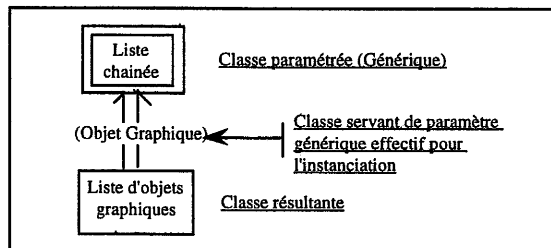


Figure II.12 : Exemple de généricité

Lorsque des vérifications statiques sont effectuées sur les types des paramètres génériques effectifs (qui doivent satisfaire certaines conditions, posséder certaines opérations,...) on parle de **généricité contrainte.** Si les vérifications sont inhibées, il s'agit de **généricité non contrainte.**

4.7.3 La notion d'héritage simple

La notion d'héritage fait partie des concepts clés de la modélisation par objets. Coad et Yourdon proposent la définition suivante :

"Héritage : mécanisme destiné à exprimer les similitudes entre classes, et ce en simplifiant la définition de classes possédant des similitudes avec celles précédemment définies. L'héritage met en oeuvre les principes de généralisation et de spécialisation en partageant explicitement les attributs et les services communs au moyen d'une hiérarchie de classes" [COAD 91].

Prenons un exemple d'héritage simple entre deux classes, l'une appelée classe "mère" et l'autre classe "fille". La classe "fille" **hérite** des caractéristiques de sa classe "mère" (attributs et méthodes) mais elle se distingue par ses caractéristiques propres. La classe "fille" est appelée **sousclasse,** et la classe "mère" est appelée **superclasse.** La terminologie "orientée objet" parle aussi de **classes de base** pour les superclasses et de **classes dérivées** pour les sousclasses. L'action de **dérivation** consiste à créer une nouvelle classe par enrichissement d'une classe de base (nouvelles méthodes, nouveaux attributs). Lorsque ce concept est étendu à une famille d'objets, on obtient un arbre d'héritage. La hiérarchie des classes qui en découle est appelée hiérarchie d'héritage ou de **généralisation / spécialisation.** Il convient également de présenter une autre terminologie utilisée par Bertrand Meyer pour qui les superclasses sont appelées les **ancêtres,** les sousclasses étant appelées **descendants** avec la nuance des **descendants propres** d'un ancêtre lorsque ceux-ci sont les descendants directs. La notion d'héritage fait aussi intervenir la notion de **classe abstraite.** Une classe abstraite est une classe qui ne peut pas avoir d'instances. Ces classes sont utilisées comme superclasses dans un graphe d'héritage. Elles permettent de raisonner à un niveau d'abstraction plus élevé que celui des classes destinées à produire des objets. Du fait du rôle des classes abstraites, on utilise fréquemment la terminologie de **superclasse abstraite** pour les nommer (figure II.13).

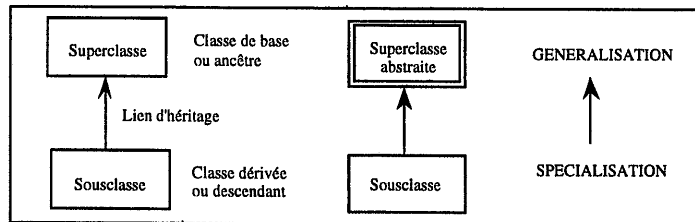


Figure II.13 : Présentation graphique de la notion d'héritage

Il convient de compléter la terminologie exposée avec celle introduite par le plus "pur" des langages à objets : Smalltalk [GOLDBERG 83]. Dans un système Smalltalk, tout est objet, y compris les classes qui sont des objets instances de classes de plus haut niveau appelées **métaclasses**. Les **métadonnées** sont des données qui décrivent d'autres données. Ainsi une classe peut être considérée comme un méta-objet car elle décrit les attributs et les méthodes des objets. Elle est le moule qui permet de fabriquer des instances. Les modèles de simulation sont eux-mêmes des métadonnées car ils décrivent les systèmes qu'ils modélisent. La relation entre une métaclasse et une classe est une relation d'instanciation, tout comme la relation entre une classe est un objet. Hormis le cas de généricité une métaclasse instancie une seule classe. Si on considère la relation d'héritage comme une relation verticale, on pourrait établir la relation d'instanciation sur un plan horizontal et donc perpendiculaire au plan d'héritage. Si l'on souhaite représenter sur un même schéma les classes et les métaclasses on perçoit le « monde des métaclasses » comme un monde parallèle reproduisant les mêmes relations d'héritages que celui du « monde des classes » mais à un niveau d'abstraction plus élévé. La communication entre les « deux mondes » se faisant sur un plan horizontal par les relations d'instanciation.

La cohérence conceptuelle du "tout objet" amène naturellement la définition d'une métaclasse qui décrit les attributs et les comportements d'une classe. En effet il existe des « varibles globales » ou des « fonctions globales » qui ne sont pas propres à un seul objet mais qui sont commun à tous les objets d'une même classe. Prenons par exemple les fenêtres de Windows. La couleur du bandeau des fenêtres peut être spécifiée par l'utilisateur. Cette couleur sera ensuite appliquée à toutes les fenêtre Windows, c'est un attribut commun à toutes les instances de la classe Fenêtre Windows. La méthode changer la couleur des fenêtres est une **méthode de classe** (ou encore dite **méthodes de la fabrique)**, la couleur des bandeaux des fenêtres est une **variables de classe** ou **attributs de classe**. On peut ,

ainsi distinguer les caractéristiques « globales », propres à une classe des classiques **méthodes d'instance** (telle que déplacer une fenêtre) qui s'applique à une seule instance de fenêtre que l'on déplace et des variables d'instance (telles que les coordonnées à l'écran d'une fenêtre.

Les méthodes de classe se révèlent être à l'usage, très utiles pour toute une catégorie de services. La méthode de création d'un objet n'est pas une méthode d'instance. De même, supposons qu'un logiciel permette de spécifier un système de production à l'aide d'objets graphiques. Si l'on souhaite générer du code de simulation pour le système de production spécifié, nous pouvons associer à chaque classe d'objets graphiques un code de simulation particulier, suivant le langage de simulation à objets utilisé. Toutes les instances graphiques représentant le système de production sont stockées dans un ensemble d'objets appelé container. Quel que soit le langage de simulation retenu, le code de simulation correspondant à chaque classe d'objet, ne doit pas être généré à chaque instance rencontrée dans la collection mais une seule fois pour chaque classe ayant des instances. La méthode à appliquer est une méthode de classe et non une méthode d'instance. Cette méthode doit utiliser une variable de classe, pour compter le nombre d'instances générées, ou tout du moins indiquer si une classe a produit des instances (Figure II.14). Par contre si l'on souhaite générer la partie de code de simulation provoquant l'instanciation de tous les objets de simulation correspondant aux objets graphiques présents dans le container, il s'agit d'une méthode d'instance.

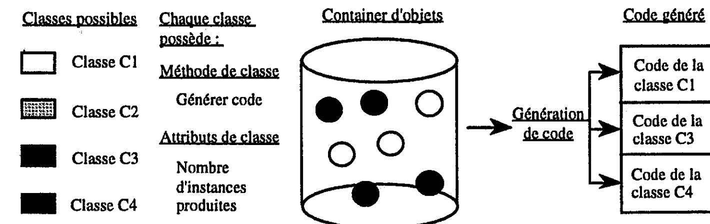


Figure II.14 : Exemple de méthode et de variable de classe

Donnons un exemple classique d'arbre d'héritage simple correspondant au domaine de l'animation des systèmes flexibles de production. La classe abstraite de base est un transporteur comportant un identificateur, deux couples de coordonnées graphiques et des méthodes servant à initialiser et à consulter les coordonnées et l'identificateur. Ensuite, on définit une classe convoyeur qui hérite

de toutes les caractéristiques de la classe transporteur, mais qui possède une vitesse et une longueur. Les méthodes propres à un convoyeur concernent l'initialisation et la consultation de la vitesse et de la longueur, le chargement et le déchargement de pièces. Notons qu'un convoyeur possède, par héritage, les attributs : coordonnées graphiques, identificateur,... ainsi que les méthodes de consultation et d'initialisation de ces attributs (Figure II.15).

Selon Brad Cox, l'héritage est une fonctionnalité non primordiale à la modélisation par objets. Cependant, il précise qu'il s'agit d'une technique d'organisation extrêmement puissante et utile pour construire et utiliser un ensemble de classes réutilisables [COX 86]. Cox souligne également qu'une conception à base d'héritage est naturelle et réduit notablement les temps de développement. Budd [BUDD 91] s'accorde sur de nombreux points avec Cox en spécifiant lui aussi que l'héritage n'est pas un concept central de la modélisation par objets, mais constitue plutôt une technique d'implémentation offrant de nombreux avantages. Ce point de vue est en opposition avec les idées d'autres auteurs comme Booch, Coad et Yourdon [BOOCH 91] [COAD 91] pour qui l'encapsulation, l'héritage et la communication par messages font partie des clés d'une analyse et d'une conception par objets. Pour notre part, nous considérons l'héritage, au même titre que la composition, comme une des relations privilégiées et fondamentales entre les classes pour ce qui concerne la modélisation par objets. Nous pensons que l'héritage est un concept puissant et naturel, qui se retrouve dans de nombreuses sciences. L'héritage en tant que simple technique d'implémentation (partage de code), peut cependant mener à des aberrations comme l'**héritage de construction** qui nous semble être la pire des utilisations possibles de l'héritage. L'héritage permet, s'il est bien utilisé, de conserver au niveau du code une vision conceptuelle. Il nous semble donc plutôt bizarre de vouloir utiliser ce concept lors de l'implémentation pour partager du code alors qu'aucune sémantique réelle de la relation d'héritage ne peut être mise en œuvre.

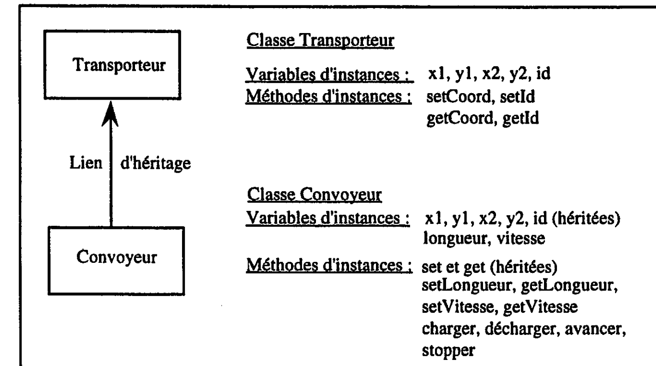


Figure II.15 : Exemple d'héritage simple d'attributs et de méthodes

4.7.4 Avantages et inconvénients de l'héritage simple

Nous allons exposer comment l'héritage amène un certain nombre d'avantages dans les domaines suivants : la réutilisabilité, la fiabilité, le partage de code, le développement incrémental de composants logiciels, le prototypage rapide et le polymorphisme.

Quand une classe hérite d'un certain nombre de méthodes de sa superclasse, ce code n'a pas à être réécrit. Nous avons précédemment souligné que beaucoup de concepteurs et de développeurs passent leur temps à écrire des portions de code déjà connues ; l'héritage de méthodes permet d'éviter certaines redondances.

Reprenons l'exemple de la classification des transporteurs : les méthodes et les attributs de la classe abstraite "transporteur" peuvent être réutilisés. La réutilisabilité du code est un gage supplémentaire de fiabilité. Intéressons nous maintenant aux navettes et aux convoyeurs qui héritent de la superclasse "transporteur". Les deux classes "navette" et "convoyeur" partagent le code des méthodes de la superclasse "transporteur". Il peut également devenir intéressant de créer une superclasse abstraite intermédiaire qui recense les caractéristiques et les comportements communs aux "transporteurs statiques" (les spécificités propres aux "convoyeurs à accumulation" ou aux "convoyeurs à bande" étant spécifiées dans des sousclasses). De même, les classes "navette" et "chariot filo-

guidé" peuvent être considérées comme des spécialisations d'une superclasse abstraite "transporteur dynamique" (Figure II.16). Lorsque l'on incrémente un modèle objet par rajout de nouvelles classes possédant quelques petites différences (spécificités propres) avec les superclasses existantes, on parle de programmation différentielle.

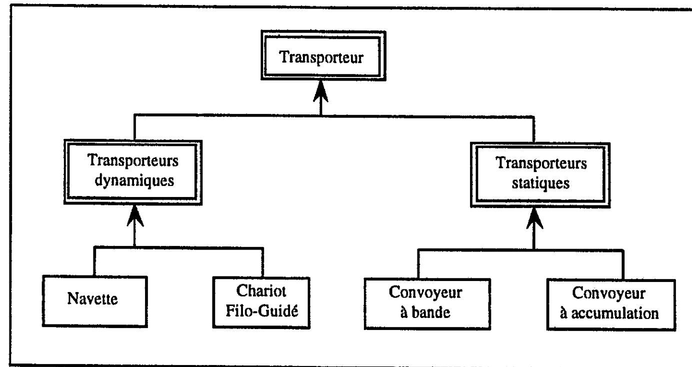


Figure II.16 : Exemple de classification par héritage

Parmi les avantages de l'héritage sur l'exemple proposé on peut citer, d'une part, une taille plus faible du code source du modèle élaboré qu'avec une approche classique ne factorisant pas les caractéristiques communes, et d'autre part, une maintenance facilitée. En effet, les problèmes sont plus vite localisés et ne concernent que de faibles portions de code. Nous pouvons donner brièvement deux exemples de cas de maintenance favorables à l'utilisation de l'héritage, l'un concernant une maintenance curative et l'autre une maintenance évolutive :

1) Supposons que l'on souhaite corriger une erreur d'implémentation sur la classe "navette" (par exemple le stockage des points d'arrêts dans une structure de donnée différente), il suffit de modifier le code spécifique à la classe "navette" sans avoir à se soucier du code hérité. Ce code particulier à la classe navette est plus petit et donc plus simple que si toutes les caractéristiques des navettes (attributs et méthodes) étaient stockées dans un seul et même module.

2) Si l'on souhaite rajouter une capacité en nombre de pièces aux transporteurs, il suffit de rajouter un attribut capacité à la superclasse abstraite "transporteur" ainsi que les méthodes d'initialisation, accès et modification correspondantes. Après cette opération, toutes les sousclasses héritent de ces modifications et disposent donc d'un attribut capacité et des méthodes associées.

Le prototypage rapide [JORDAN 89] est une technique basée sur les notions d'héritage et de composants réutilisables. Les efforts de conception et de développement peuvent se concentrer sur ce qui est nouveau et inhabituel dans le problème que l'on cherche à traiter. C'est donc uniquement cette nouvelle partie qui est ajoutée dans la hiérarchie des classes de composants existantes. Le développement des systèmes est ainsi accéléré. Ces techniques sont connues sous les terminologies de **prototypage rapide** ou de **programmation exploratoire**. Ce style de programmation itérative basée sur une succession de prototypes est également très utile lorsque les objectifs du système à concevoir ne sont que vaguement maîtrisés et/ou compris à l'origine d'un projet.

Après avoir exposé les avantages de l'héritage, nous en examinons le coût. Il peut être chiffré en temps d'exécution : il est plus coûteux en temps de gérer une hiérarchie de descripteurs de classes pour retrouver l'adresse d'une méthode que d'appeler directement cette méthode. Néanmoins, le gain de temps en conception et en maintenance reste un atout bien plus important que les nanosecondes perdues pour gérer le mécanisme d'héritage. De plus, le gain en qualité au niveau de la conception peut facilement aboutir à de meilleures performances globales. Ce type de raisonnement ne convaincra cependant pas certains programmeurs inconditionnels de l'assembleur.

Un autre reproche concerne la taille souvent importante des programmes résultants. Ce gaspillage est très faible avec certains langages à objets (tels que C++), qui ne conservent à l'exécution que très peu d'informations sur les classes. Cependant certaines implémentations, notamment celles adoptant un modèle à typage dynamique, sont assez gourmandes en mémoire (Smalltalk, CLOS ou encore Objective-C dans une moindre mesure...). Soulignons que la maîtrise des coûts de développement et la production rapide de code de haute qualité valent bien quelques sacrifices financiers au profit des vendeurs de mémoires dynamiques.

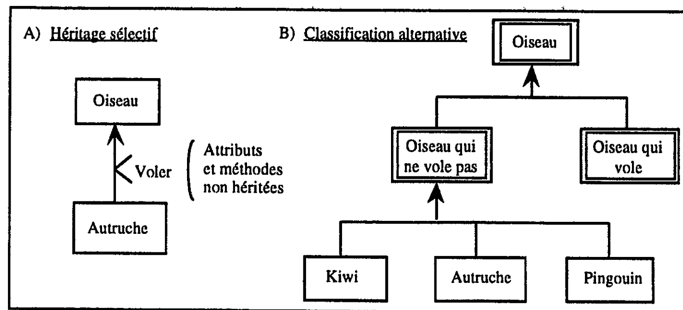


Figure II.17 : Héritage sélectif avec filtrage d'une méthode

Parmi les reproches faits à l'héritage, on trouve la possibilité de violer le principe d'encapsulation. En effet, avec un mécanisme d'héritage primaire, une sousclasse peut accéder à une variable d'instance de sa superclasse, et appeler une de ses opérations privées, ou encore accéder directement aux superclasses de sa superclasse. Ceci peut être évité car le concept actuel de l'héritage introduit également la notion de masquage d'informations. Une superclasse peut ainsi masquer certains attributs et certaines méthodes qui ne pourront pas être hérités par ses sousclasses. Dans certains cas, il peut également être utile de ne pas hériter de certaines fonctionnalités : on parle alors d'**héritage sélectif, d'exception à l'héritage** ou de contraintes sur l'héritage. L'exemple classique pour la représentation de la connaissance est celui de l'autruche qui est un oiseau mais qui ne vole pas (Figure II.17). Un héritage sélectif peut souvent être contourné par une conception soigneuse de classes abstraites.

Par analogie avec la notion de module, qui, utilisée à l'extrême, conduit à des solutions complexes, une mauvaise utilisation de l'héritage peut, elle aussi, conduire à une autre forme de complexité. La compréhension d'un logiciel qui utilise vingt niveaux d'héritage peut nécessiter de nombreux aller-retours sur le graphe d'héritage, ce qui limite également les performances de certains langages à objets dans la recherche des méthodes.

Le lecteur intéressé par les aspects théoriques de l'héritage peut se reporter aux travaux de Ducournau et Habib [DUCOURNAU 89]. De même, les différences entre le **soustypage** et l'héritage sont exposées dans [COOK 90].

4.7.5 L'héritage multiple et l'héritage à répétition

L'**héritage multiple** donne la possibilité à une classe d'hériter de plusieurs superclasses. Un exemple d'utilisation peut être la description d'un véhicule amphibie qui est à la fois une automobile et un bateau ou encore d'une classe "robot industriel" qui hérite des classes "stock", "machine" et "transporteur statique". Des conflits et des collisions peuvent se produire lorsque l'on hérite de deux méthodes ou de deux attributs portant le même identificateur. Par exemple, la classe "automobile" possède la méthode "avancer" qui met en oeuvre les roues par le biais du moteur de l'automobile, et la classe "bateau" possède également la méthode "avancer" mettant en oeuvre l'hélice au moyen du moteur du bateau. De même, les classes "stock", "machine" et "transporteur statique" possèdent toutes un identificateur ; la classe robot dispose alors de trois identificateurs (Figure II.18). Ces conflits sont, dans certaines implémentations, gérés de manière explicite, c'est à dire en précisant de quelle superclasse provient la méthode (par exemple C++ [ELLIS 90]).

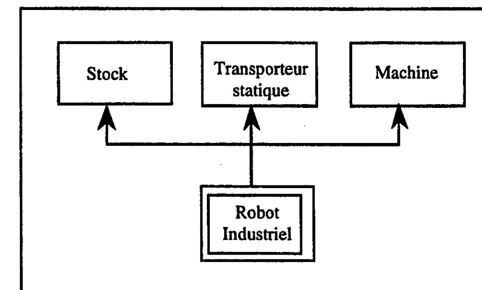


Figure II.18 : Exemple d'héritage multiple

Le conflit peut également être géré en laissant le système choisir la version de la méthode la plus appropriée. Une autre technique utilisée consiste à renommer les noms en collision lors de la déclaration de l'héritage. C'est cette solution qui est proposée par Bertrand Meyer [MEYER 88] pour le langage Eiffel. Dans la majorité des cas, les collisions non résolues par le développeur sont dénoncées par les compilateurs.

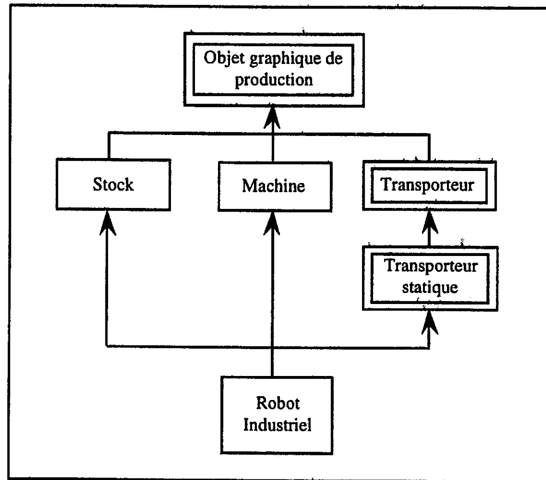


Figure II.19 : Héritage à répétition

L'exemple du robot est simpliste et met en évidence trop de duplications d'attributs (un robot ne possède pas trois identificateurs). Il semble qu'il conviendrait de définir une superclasse abstraite commune "objet graphique de production" pour les classes stocks, machines, et transporteurs (Figure II.19). Si la solution paraît meilleure, le problème de duplication est toujours présent. On parle d'**héritage à répétition** lorsqu'une classe hérite deux fois (ou plus) d'une de ses superclasses (ici objet graphique de production). Les solutions proposées pour gérer les duplications de l'héritage à répétition sont liées au renommage. Les primitives non renommées sont partagées et les primitives renommées sont dupliquées.

Beaucoup de langages à objets ne proposent pas d'héritage multiple, notamment la spécification de Smalltalk80 [GOLDBERG 83] et d'Objective C [COX 86] [COX 91]. Certaines versions plus récentes de Smalltalk implémentent cependant la notion d'héritage multiple. Il est d'ailleurs dommage que ce langage possède des implémentations avec des caractéristiques différentes.

4.7.6 La notion de composition ou d'agrégation

La relation de composition ou d'agrégation est une relation fondamentale du processus de classification. Elle peut se comprendre comme la relation « avoir » au sen large, ou également un sens plus précis « posséder », « être composé de ». Voici un exemple de composition ; attachons-nous à modéliser le problème suivant en termes de classes et d'objets : une voiture est constituée d'un certain nombre d'éléments (carrosserie, moteur, direction, transmission, roues, ...). En considérant ces éléments comme des classes d'objets, il convient de ne pas se méprendre en disant que la classe "voiture" hérite des superclasses "roue", "carrosserie",... Ce type de relation n'est pas du tout un héritage multiple mais une **composition**. L'objet "voiture" constitue un tout composé de parties (les roues, la carrosserie, la direction, le moteur, ...) (Figure II.20). On parle aussi **d'agrégation**, la voiture réalisant une agrégation des autres classes. Les hiérarchies issues d'agrégation mènent à la notion de **niveaux d'abstraction**. Avec cette notion, la classe "voiture" se situe à un niveau d'abstraction supérieur par rapport aux classes "roues", "carrosseries",... Un autre exemple peut être donné dans le domaine des systèmes de production : une unité de production / stockage peut être elle-même composée d'autres unités de production / stockage. Suivant le niveau d'abstraction correspondant, on parlera d'usine, d'atelier ou d'îlot de production. La relation d'agrégation est donc utilisée en tant qu'outil de décomposition hiérarchique.

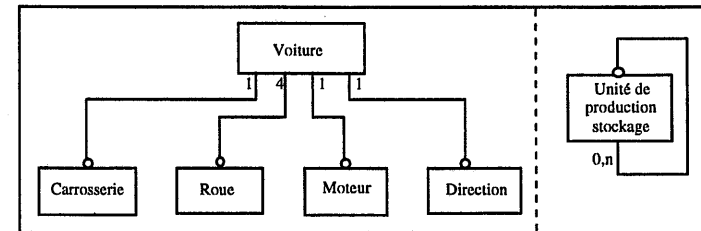


Figure II.20 : Exemples de relation de composition

Donnons un autre exemple classique pour différencier les relations de composition et d'héritage. Il s'agit des avions AWACS qui ont la particularité de se comporter comme une sorte de radar. Avec cette vision, on peut donc effectuer un héritage multiple, la classe AWACS héritant à la fois de la classe radar et de la classe avion (Figure II.21). Supposons maintenant qu'un AWACS dispose de plusieurs radars identiques, il faut maintenant faire intervenir les deux relations :

un héritage de la superclasse avion et une composition avec la classe radar (un AWACS étant "composé de" 1 à n radars identiques).

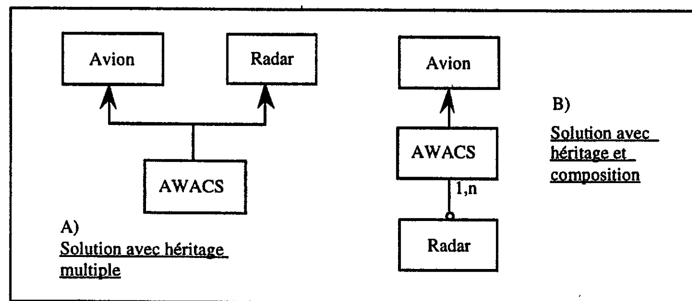


Figure II.21 : Exemple d'héritage et de composition possibles

4.8 La classification dynamique et l'héritage dynamique

La classification dynamique correspond à la possibilité de changement de classe dont disposent les objets au cours de leur vie. Avec une classification dynamique, un objet peut être instance de classes différentes à différents moments. Un objet de la "classe moto routière" peut être transformée en moto de la classe "chopper" chez un préparateur. Nous dissocions trois types de changements de classe :

- le transtypage simple d'un objet d'une classe en un objet d'une autre classe,
- l'assemblage de différents objets (pouvant être de classes distinctes) en un objet d'une autre classe,
- l'éclatement d'un objet d'une classe donnée en plusieurs objets pouvant être de classes différentes.

La notion de classification dynamique comprend bien sûr la possibilité de rajout ou de suppression d'attributs ou de méthodes à une classe au cours de la vie d'un système d'objets.

L'héritage dynamique impose que les informations propres à une classe restent sa propriété exclusive. Les informations héritées n'existent qu'à un seul endroit du graphe d'héritage et elles sont exploitées par un mécanisme

dynamique. L'héritage statique implique par contre la recopie des informations héritées de la superclasse directe. Il est plus efficace et souvent utilisé pour la gestion des variables d'une classe (variables propres et variables héritées de la superclasse directe). L'héritage dynamique est par contre utilisé pour partager les méthodes dans certaines implémentations inspirées de Smalltalk80. Un héritage dynamique complet facilite le rajout ou la suppression des liens d'héritage au cours de la vie d'un système d'objets. Il apporte une souplesse intéressante mais est plus coûteux en temps d'exécution, car les solutions gérant les répercussions sur la hiérarchie d'héritage nécessitent une recherche dynamique des informations héritées. Cette souplesse se paie également dans le sens où il devient impossible de vérifier à la compilation la validité des appels de méthodes ; la fiabilité des systèmes logiciels conçus reste alors totalement à la charge du programmeur.

4.9 Le polymorphisme et la liaison dynamique

4.9.1 Introduction au polymorphisme

Nous venons de voir que l'héritage est principalement utilisé pour la réutilisabilité. C'est également dans ce but et dans celui de l'extensibilité que le **polymorphisme** et la **liaison dynamique** viennent compléter l'héritage. Ces concepts apportent une puissance et une souplesse qui n'avaient pas encore été atteintes avec les autres techniques de développement. Le polymorphisme permet de définir plusieurs formes pour une méthode commune à une hiérarchie d'objets. C'est à l'exécution que l'on détermine quelle forme appeler suivant la classe de l'objet courant grâce à la liaison dynamique. La notion de **méthode polymorphe,** mise en valeur par l'héritage, constitue une des caractéristiques les plus puissantes des langages à objets.

Fixons immédiatement les idées avec un exemple. Nous disposons d'une classe de base "objet graphique de production" et de ses sousclasses "machine", "stock" et "transporteur". Les instances des objets graphiques de production sont stockées dans une collection d'objets regroupant tous les objets d'un système de production (Figure II.22). Nous souhaitons afficher sur un écran tous les objets d'un système de production. Il est particulièrement intéressant de pouvoir spécifier, à l'aide d'un code général, que chaque objet de la collection, quelle que soit sa classe, reçoit un message lui demandant d'activer sa méthode d'affichage. Dans ce contexte le code pourrait être du style :

```
Pour tous les objets de la collection
  objet AFFICHE
Fin pour
```

Un code qui n'utiliserait pas le polymorphisme pourrait utiliser une instruction à choix multiple suivant la classe des objets rencontrés. Un tel code serait donc à modifier à chaque ajout ou suppression de classe. Voici un exemple de code sans appel de méthode polymorphe.

```
Pour tous les objets de la collection
  cas MACHINE              : afficher_machine
  cas CONVOYEUR            : afficher_convoyeur
  cas STOCK                : afficher_stock
  cas NAVETTE              : afficher_navette
  ...
Fin Pour
```

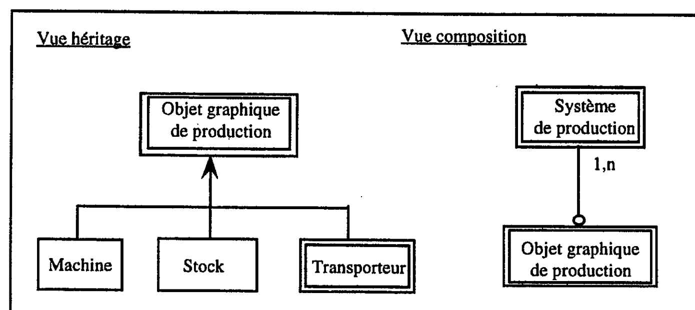


Figure II.22 : Vue héritage et composition pour exemple de polymorphisme pur

Ce que nous venons de détailler se nomme polymorphisme pur. Le message AFFICHE est polymorphe. Dans un modèle à typage statique, on parle de méthodes virtuelles, ou de méthodes différées lorsque les méthodes possèdent la faculté de prendre différentes formes suivant le contexte d'exécution. Par contre, avec un modèle dynamique, les messages sont toujours considérés comme pouvant être polymorphes. Dans les deux cas, le polymorphisme est disponible grâce à la liaison dynamique (ou **édition des liens retardée**) implémentée sous diverses formes. La représentation graphique d'une machine est différente de celle d'un stock ou d'un transporteur, cependant, à l'exécution de notre exemple, la bonne méthode d'affichage sera appelée. Ceci suppose bien sûr que chaque classe possède sa propre méthode nommée *AFFICHE*.

Le polymorphisme peut être contraint par l'héritage, comme c'est le cas dans les langages à typage statique tels que Simula, C++ et Eiffel. Ceci signifie que le compilateur vérifie que la classe "objet graphique de production" possède bien une primitive virtuelle *AFFICHE* et que tout descendant de la classe "objet graphique de production" possède ou hérite d'une implémentation de cette méthode. Par exemple, il est impossible et incohérent de demander à une voiture de décoller puis de voler à 3000 pieds. Dans les langages à typage dynamique, ce type d'incohérence peut n'être détectée qu'à l'exécution. Ceci semble incompatible avec les objectifs de validité, de robustesse et de fiabilité. Mais d'un autre côté, la souplesse apportée par ce manque de contrôle strict est propre à favoriser une meilleure réutilisabilité et une meilleure extensibilité. Par contre, avec les langages à typage statique, le compilateur vérifie qu'un programmeur ne demande pas à un objet un service qu'il ne sait pas exécuter.

Reprenons l'exemple de la collection d'objets graphiques. Nous souhaitons disposer d'une méthode permettant de sauver tous les objets sur disque. Indépendamment de cette première méthode nous voulons également disposer d'une méthode pour effacer les objets graphiques de l'écran. Une des implémentations possibles est la suivante :

```
Première méthode :     Pour tous les objets de la collection
                          objet SAUVE
                       Fin pour

Deuxième méthode :     Pour tous les objets de la collection
                          objet EFFACE
                       Fin pour
```

Il est aisé de remarquer que nous avons une redondance de code maladroite pour exécuter l'affichage, la sauvegarde et l'effacement (ou toute autre fonction polymorphe). Avec le modèle objet, il est possible de disposer d'un composant unique pour le parcours de la collection d'objet. Cette méthode doit accepter une méthode polymorphe comme paramètre, ce qui donnerait :

```
Composant(METHODE_POLYMORPHE)
Pour tous les objets de la collection
      objet METHODE_POLYMORPHE
Fin pour
```

Certains langages à objets disposant d'un typage statique, comme SIMULA, n'autorisent pas l'implémentation de tels composants. D'autres, comme le langage C++, l'autorisent mais au prix d'expressions syntaxiques qui pourraient être cocasses si la maintenance n'existait pas (passage en paramètre de pointeurs sur fonctions membres virtuelles [HILL 91]).

4.9.2 Les différentes formes de polymorphisme

Nous venons d'évoquer le polymorphisme pur. Nous allons aborder la **surcharge** qui est une forme très simple de polymorphisme, la **redéfinition** et le **polymorphisme multiple** qui en sont des formes sophistiquées.

Il convient de préciser certains aspects qui distinguent le polymorphisme pur de la surcharge de méthodes. La surcharge est une forme primitive de polymorphisme permettant seulement de disposer de plusieurs sémantiques pour le même identificateur de méthode. La surcharge distingue deux fonctions de même nom par leur **signature d'arguments** (nombre et type des arguments) qui doit être différente, on parle de **surcharge paramétrique**. La surcharge ne permet pas l'extensibilité du logiciel. Par contre grâce au polymorphisme pur, un code tel que celui écrit pour l'affichage d'objets graphiques n'a plus à être modifié. Quel que soit le nombre de sousclasses d'objets graphiques que l'on ajoutera ou retirera au logiciel, les méthodes utilisant un polymorphisme pur doivent avoir la même signature d'arguments.

Pour les langages à typage dynamique, les **sélecteurs de message** permettent d'identifier les méthodes. La signature d'argument est intégrée dans le sélecteur (la généricité peut également être considérée comme une forme de surcharge paramétrique).

Le polymorphisme avec **redéfinition** ou **substitution** est utilisé lorsque certaines sousclasses d'une hiérarchie d'héritage ont besoin de "redéfinir" une méthode polymorphe héritée, la redéfinition pouvant intégrer un appel à l'implémentation héritée et la substitution n'effectuant pas d'appel aux méthodes de la superclasse (figure II.23).

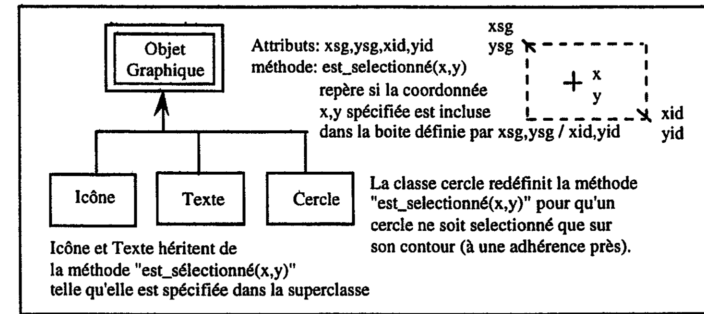


Figure II.23 exemple de redéfinition de méthode polymorphe

Le **polymorphisme multiple** fait intervenir plusieurs objets (de classes différentes) pour déterminer la bonne méthode à appeler. On peut donner l'exemple d'opérateurs mathématiques diadiques polymorphes ; l'addition d'un réel et d'un complexe ne se traite pas comme l'addition d'un vecteur et d'un complexe. De même, une méthode d'affichage graphique peut être polymorphe suivant la classe du dispositif matériel d'affichage et suivant la classe d'objet graphique. Le polymorphisme multiple est associé à la notion de multi-receveur introduite dans CEYX, qui est un système d'objets développé sous Le_Lisp par Jean-Marie Hullot [HULLOT 83].

A l'exclusion de la surcharge, les diverses formes de polymorphisme permettent au développeur de créer des composants possédant un haut degré de réutilisabilité et d'extensibilité. Sans cette aide, un développeur devrait écrire un code constitué d'instructions de choix multiples suivant la ou les classes d'objets manipulées, ce code devrait être mis à jour à chaque ajout ou suppression de classes. Les formes avancées de polymorphisme constituent probablement les outils les plus puissants de la programmation par objets qui n'existent que lorsque l'on dispose de l'héritage et de l'édition de lien dynamique.

4.10 La persistance

L'intervalle de temps délimité par l'instant de création et l'instant de destruction d'un objet correspond à la durée de vie d'un objet. Les différentes catégories d'objets peuvent être classées suivant leur durée de vie de la manière suivante [BOOCH 91] :

- les objets résultats temporaires lors de l'évaluation d'une expression,
- les objets locaux à une méthode en cours d'exécution,
- les objets locaux à un module (ou à la description d'une classe) qui sont alloués de manière statique,
- les objets alloués de manière globale pour tout le logiciel,
- les objets alloués dynamiquement,
- les objets qui continuent d'exister entre deux exécutions du logiciel qui les manipule.

La persistance peut être définie comme suit : "La persistance est la propriété qui permet à un objet de continuer d'exister après que son créateur ait cessé d'exister, y compris à un endroit différent de celui où il a été créé" [BOOCH 91]. Les objets, dont la durée de vie outrepasse la durée d'exécution d'un logiciel, sont appelés **objets persistants**. La persistance ne se limite pas simplement aux considérations de durée de vie, mais elle concerne aussi les problèmes d'allocation d'espace. En effet, un objet créé à une adresse en mémoire centrale peut être sauvé sur disque (avec sa classe et son état), puis restauré à partir de son image sur disque à une autre adresse de mémoire centrale.

Outre les applications immédiates aux SGBD, de simples applications de CAO ou de DAO manipulent des ensembles d'objets en mémoire. Ces ensembles constituent des modèles que l'on souhaiterait archiver sur disque, puis restaurer en mémoire de manière simple. Il serait donc souhaitable d'implémenter des portions de code du style :

```
Archivage  Pour toute la collection d'objets
             objet SAUVER sur un fichier
           fin pour

Relecture  Tant que Non fin du fichier
             Lire et instancier l'objet lu, suivant sa classe !
           fin tque
```

Ces extraits de code se veulent hautement réutilisables et extensibles. Le cas d'une méthode SAUVER polymorphe pour toutes les classes d'objets manipulées ne pose pas de problème. Par contre, la relecture pose des problèmes car il faut être capable de reconnaître la classe de l'objet archivé, d'allouer l'espace mémoire pour une instance et d'initialiser l'objet avec l'état archivé. Peu d'implémentations supportent directement la persistance. Les solutions proposées pour le langage Objective-C sur station NeXT™ déchargent complètement le développeur de toute intervention sur les parties de code abstraites, telle que celle énoncée plus haut, en cas de rajout d'une ou plusieurs classes. Des bibliothèques ont été développées pour fournir la notion de persistance aux langages à objets qui ne la

possèdent pas en standard (par exemple la bibliothèque de classe OOPS pour le langage C++ [GORLEN 87]).

4.11 La concurrence et les modèles d'acteurs

La modélisation par objets conduit à décomposer le logiciel en classes d'objets. Pour distribuer un logiciel, il faut échapper à l'architecture séquentielle conventionnelle de la majorité des ordinateurs. Un modèle orienté objet d'un système concurrent peut donc être vu comme un système d'objets, pouvant être actifs simultanément et communiquant les uns avec les autres au moyen de messages. En fait, il apparaît que la concurrence est une conséquence naturelle du concept d'objet. La concurrence repose sur le potentiel d'exécution parallèle de parties de calculs. Les composants d'un programme peuvent être exécutés séquentiellement ou en parallèle, sur un seul ou sur plusieurs processeurs. La concurrence permet de faire abstraction d'une partie des détails de l'exécution. Il existe plusieurs modes de concurrence pour la résolution de problèmes [AGHA 90] :

- Utiliser la concurrence en mode pipeline. Il faut énumérer les étapes menant à la constitution d'éléments de solutions utilisables par d'autres étapes. Il faut ensuite faire évoluer en parallèle ces différentes étapes pour aboutir aux solutions finales.

- Diviser pour conquérir. Ceci implique l'élaboration concurrente de différents sous-problèmes et la jonction des solutions pour obtenir la solution globale du problème. Il n'y a pas d'interaction entre les différentes procédures traitant les sous-problèmes.

- Résoudre le problème par coopération. Ceci implique un réseau de connexions complexes entre différents objets coopérants. Chaque objet possède son processus de calcul et peut communiquer avec les autres objets, pour, par exemple, partager les résultats intermédiaires au fur et à mesure du calcul (en envoyant des messages). Avec une approche processus, ce mode de concurrence s'adapte bien à la simulation [BEZIVIN 88a] (simulation par un réseau d'acteur ou chaque message est estampillé par un temps virtuel ou par une décomposition en clients (actifs) et serveurs (passifs)).

- Coopération concurrente par tableau noir. Le principe consiste à considérer un "tableau noir" comme étant une structure contenant une solution en cours de construction incrémentale. Plusieurs modules, considérés comme des sources de connaissances, sont spécialisés dans différentes opérations

qu'ils peuvent réaliser sur le tableau noir. Le contrôle est assuré par un module "stratégie" qui analyse l'état d'avancement vis-à-vis de la résolution du problème et qui décide des opérations à réaliser [CAROMEL 90].

Le mode de communication par messages proposé dans Smalltalk par Alan Kay peut supporter le fonctionnement en parallèle de plusieurs processus communicants [BEZIVIN 88b]. C'est avec Simula que les classes ont été pour la première fois utilisées pour représenter des tâches. Les processus de Simula sont prévus pour tous s'exécuter sur le même processeur séquentiel, on parle de quasi-parallélisme avec la notion de **coroutines** [DAHL 66] [BIRTWISTLE 73]. Il s'agit du mécanisme le plus simple qui soit pour simuler la concurrence avec un seul processeur (Figure II.24). Chaque coroutine se partage le processeur central à tour de rôle, les données sont partagées et les communications ne sont pas limitées. Cette technique est très simple à implémenter et ne nécessite que quelques fonctions en langage d'assemblage pour effectuer les changements de contexte de manière efficace. Les coroutines conviennent particulièrement à la réalisation de modèles de simulation qui doivent partager un ensemble de données communes. Avec cette technique, les contraintes d'exclusion pour les accès concurrents n'ont pas à être vérifiées car ils ne peuvent pas se produire physiquement.

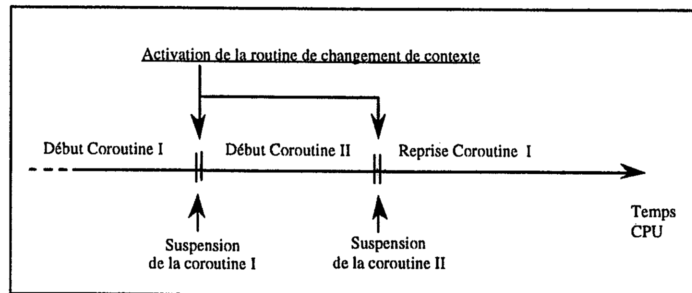


Figure II.24 : Mécanisme des coroutines

Les concepts de la modélisation par objets conviennent également très bien pour des systèmes hautement parallèles implémentés sur plusieurs processeurs indépendants qui partageraient les mêmes données. Un modèle objet est bien adapté aux systèmes distribués car il définit de manière implicite les composants distribués et la manière dont ils communiquent. Cox [Cox 87] donne l'exemple

d'une usine d'assemblage automatisée, où un nombre important de micro-ordinateurs supervisent une partie localisée de l'usine en temps réel, tout en communiquant avec d'autres ordinateurs ayant des responsabilités différentes.

Parmi les autres tentatives de gestion des problèmes posés par la concurrence, on peut citer les **modèles d'acteurs** qui se rapprochent sensiblement des modèles objets à base de classes. Les modèles d'acteurs présentés par Hewitt au MIT dans les années soixante-dix sont issus des techniques de représentation de la connaissance [HEWITT 73] puis du parallélisme [LIBERMAN 87]. Plusieurs modèles ont été proposés et implémentés sous forme de langages prototypes dont les plus connus sont PLASMA, ACT 1 et 2, ABCL 1, ... Un des derniers modèles en date est celui proposé par Gul Agha [AGHA 90]. Nous allons essayer de présenter les caractères communs de ces modèles.

Un modèle d'acteurs n'utilise qu'un seul type d'objet : les acteurs. La notion d'acteur rejoint la notion d'objet actif autonome car elle regroupe des données protégées (appelées **accointances**) et un ensemble de procédures (appelé **script)** qui définit le comportement de l'acteur. Le script détermine à quel type appartient un acteur. La liste des accointances d'un acteur est susceptible d'évoluer au cours du temps (rajout de nouvelles connaissances). Les modèles d'acteurs sont donc totalement dynamiques. Le concept de classe n'existe pas, les acteurs sont simplement des entités autonomes qui peuvent se dupliquer. Les acteurs sont des composants indépendants et interactifs d'un système qui communiquent par envois de messages unidirectionnels et asynchrones (certaines implémentations proposent également des communications synchrones).

Les instanciations sont réalisées par **copie différentielle**. Il s'agit d'une duplication d'un acteur, le script de l'acteur produit est identique à celui de l'acteur qui se reproduit. Les accointances peuvent avoir des valeurs différentes et il est même possible d'ajouter de nouvelles accointances (qui n'existaient pas dans l'acteur d'origine).

Le partage des informations est réalisé par **délégation**, à savoir qu'un acteur qui ne peut pas traiter un message le délègue à un autre acteur : son **mandataire** (figure II.25). Le mécanisme de délégation remplace en quelque sorte l'héritage. Le mandataire donne accès à un acteur qui joue le rôle d'une superclasse dans la hiérarchie d'héritage. Un message, qui ne peut pas être compris par l'acteur récepteur, est transmis à son mandataire qui est "délégué" au traitement du message. Cette technique, lorsqu'elle est itérée, permet de parcourir un arbre d'héritage de manière dynamique. Ce type de parcours est réalisable car il est possible de changer dynamiquement le mandataire d'un acteur. Le modèle réalisé gagne en souplesse car l'héritage devient un cas particulier de l'envoi de

messages. Avec ce modèle, il est possible de mettre en œuvre simplement la classification et l'héritage dynamique.

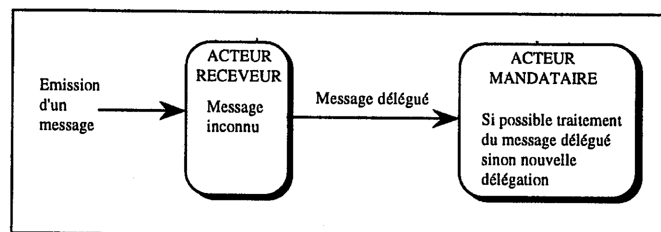


Figure II.25 : Mécanisme de délégation dans un modèle d'acteurs

Avec un modèle d'acteurs, la réponse à un message n'est pas forcément renvoyée à l'émetteur. Cette réponse peut être transmise à un tiers en suivant le concept de continuation. Le destinataire du résultat est spécifié en tant qu'accointance du message. Le potentiel des techniques de continuation et de délégation reste encore à approfondir.

Nous préférons ne pas détailler des notions telles que le changement de comportement, les acteurs non-sérialisés et sérialisés,... qui dépendent des implémentations, les concepts qui leurs sont associés n'ont pas encore été unifiés. Le lecteur intéressé par plus de détails et de références sur les modèles d'acteurs peut se reporter à l'article de Roche et Laurent [ROCHE 89] et à celui d'Agha [AGHA 90]. Afin d'assister le développeur dans sa démarche de modélisation, un environnement de développement (adapté à la visualisation, au contrôle, à la mise au point de programmes parallèles) a été proposé par Jean Pierre Briot. L'environnement en question s'appelle Actalk (acteurs en Smalltalk) et a tout d'abord été conçu pour l'évaluation des langages d'acteurs [Briot, 1989]. Cette plate-forme est aujourd'hui disponible sur Internet pour tous types de développements.

4.12 Les agents

L'intelligence artificielle a pour but initial la modélisation des connaissances et du raisonnement humain. Un compromis à cet objectif initial ambitieux est connu sous le nom d'*Intelligence Artificielle Distribuée* et se base sur la collaboration d'une multitude d'agents simples et autonomes,

organisés en société pour résoudre un problème qui peut être complexe. Les *systèmes multi-agents* prennent donc comme référence des interactions sociales élémentaires dans le but de favoriser l'émergence d'organisations complexes. Le concept d'acteur se distingue de celui d'agent par son manque d'organisation sociale (les interactions entre acteurs ne sont pas issues d'une organisation sociale). Par contre, les modèles d'acteurs sont vraisemblablement une base appropriée pour l'implémentation des systèmes multi-agents qui se séparent en deux catégories : les *systèmes d'agents cognitifs* et les *systèmes d'agents réactifs*.

Les systèmes d'agents réactifs présentent de nombreux agents simples, sans mémoire, avec une vision locale de leur environnement. Ces agents réagissent à des stimuli élémentaires leur permettant d'exprimer leur comportement et au besoin de coopérer, de s'organiser, de se reproduire... On parle d'*éco-résolution* lorsque l'ensemble des interactions non déterministes d'agents réactifs cherchant à se satisfaire permet l'apparition d'états stationnaires sur des systèmes initialement dynamiques.

D'une manière opposée, les systèmes d'agents cognitifs ne présentent que peu d'agents ; par contre, ces agents possèdent une mémoire du passé, connaissent leur environnement ainsi que les autres agents. Le fonctionnement des systèmes d'agents cognitifs est donc basé sur des protocoles de communication qui, dans la majorité des cas, sont équivalents à ceux développés pour les modèles d'acteurs. Le lecteur intéressé pourra se reporter à [Ferber 1989, 1995]. Tout comme il existe plusieurs modèles d'acteurs, il convient de dissocier les systèmes multi-agents des *agents endomorphes* de Bernard Zeigler [Zeigler 1990]. Ces agents « intelligents » sont capables d'incorporer, d'utiliser (et même de construire) des modèles d'eux- mêmes.

4.13 La gestion de la mémoire

Lors de la réalisation de logiciels de taille importante, il arrive que les objets "meurent" (lorsqu'ils ne sont plus référencés), ce qui peut entraîner un gaspillage de la mémoire. Lorsqu'un logiciel utilise abondamment des allocations dynamiques de mémoire, une demande d'allocation peut échouer car l'espace requis est utilisé par des objets morts non recyclés. Une bonne programmation ne devrait pas entraîner l'apparition d'objets morts. De plus, certains programmes ne

créent que très peu d'objets par rapport à la taille mémoire disponible. Cependant, l'expérience montre qu'un ramasse-miettes reste très utile pour compenser les faiblesses humaines. Bertrand Meyer propose que les ramasse-miettes soient du ressort de l'implémentation. Il faut alors gérer la détection des objets morts, ainsi que la récupération de la place qu'ils occupaient. La solution préconisée par Meyer est celle d'un glanage de cellule par coroutine qui peut être activé ou désactivé à volonté. D'autres solutions sont possibles pour les langages ne supportant pas ce mécanisme, le lecteur intéressé pourra se reporter à [MEYER 88]. Récemment le langage Java a montré qu'une implémentation efficace des ramasse-miettes était tout à fait viable avec les performances des machines actuelles.

5 Conclusion

Nous avons privilégié les concepts du modèle objet à base de classes et nous avons très peu détaillé les modèles d'acteurs. En outre, les frames ou schémas permettant une approche objet pour la représentation de la connaissance n'ont pas été décrits. Les concepts qui ont été présentés vont nous permettre d'aborder les cycles de développement et les méthodes d'analyse et de conception par objets. En effet si l'on souhaite conserver l'analogie évoquée entre le développement par objets d'un logiciel et la réalisation d'un modèle de simulation, il nous faudra étudier non seulement les concepts et techniques issus de la programmation par objets, mais également les méthodes d'analyse et de conception par objets. Notre prochain chapitre va donc présenter un échantillon des principales méthodes d'analyse et de conception par objets.

POINTS CLES

- La majorité des applications logicielles sont complexes entraînant des coûts de développement et de maintenance prohibitifs.
- Les techniques du génie logiciel tentent de proposer un ensemble de solutions permettant de maîtriser les coûts de développement.
- La réutilisabilité apporte des solutions que ce soit pour réutiliser du code, du travail de conception ou d'analyse.
- La notion d'objet n'est qu'une évolution du niveau d'abstraction utilisé par les informaticiens.

- Une classe d'objets est un type de donnée abstrait décrivant les attributs propres et les méthodes communes à toutes les instances (ou objets) de ce type.
- Un objet encapsule un état qui est défini par la valeur de ses attributs, ainsi qu'un comportement défini par ses méthodes. L'encapsulation protège également l'accès aux attributs, car toute consultation ou modification d'une valeur d'attribut propre à un objet ne peut être réalisée que par les méthodes de cet objet.
- L'envoi de message est le seul mode de communication entre objets.
- Un typage statique associe les types aux identificateurs et autorise des contrôles utiles pour la validité d'un logiciel. Un typage dynamique associe un type au contenu instantané d'une variable apportant ainsi une souplesse intéressante même si elle peut se révéler dangereuse en terme de validité du logiciel qui l'utilise.
- La classification d'un ensemble de classes utilise la notion d'héritage qui met en œuvre les relations de généralisation/spécialisation en partageant explicitement les attributs et les services communs au moyen d'une hiérarchie de classe.
- Les relations de composition (ou d'agrégation) sont aussi fondamentales que l'héritage pour la classification au sein d'un modèle objet.
- La notion de méthode polymorphe, mise en valeur par l'héritage constitue une des caractéristiques les plus puissantes du modèle objet.
- Les objets dont la durée de vie outrepasse celle d'une exécution du logiciel qui les a créé sont dits persistants.
- Les modèles d'acteurs se rapprochent sensiblement des modèles à base de classes et utilisent la notion d'objets actifs (déjà présente dans SIMULA) qui possèdent leur propre flot de contrôle décrit par un script (décrivant le rôle de l'acteur).
- Les agents sont des acteurs particuliers en ce sens qu'ils établissent des relations sociales entre eux. « L'assouvissement » de leurs objectifs propres en utilisant leur « réseaux sociaux » permet parfois un « éco-résolution » de certains problèmes dits complexes.

EXERCICES

2.1 Expliquez la relation classe/instance.

2.2 Essayez de donner une différence entre l'abstraction de données et l'encapsulation.

2.3 Une classe abstraite est elle toujours une superclasse abstraite ?

2.4 Le polymorphisme peut-il être utilisé sans héritage et sans liaison dynamique ?

2.5 Donnez les principaux avantages et inconvénients d'un typage dynamique.

2.6 Pourquoi les implémentations d'une fonction virtuelle doivent-elles toujours avoir la même signature d'arguments ?

2.7 Expliquez les techniques qui permettent de mettre en oeuvre une édition de lien dynamique; quels en sont les avantages et les inconvénients.

2.8 Donnez un exemple d'héritage à répétition. Evoquez les problèmes que pose un tel héritage et présentez une ou plusieurs solutions.

2.10 La relation entre une métaclasse et une classe correspond-t-elle à une relation d'héritage ou à une relation d'instanciation ?

2.11 Est-ce qu'une métaclasse peut-instancier plusieurs classes ?

2.12 Dessiner un schéma comportant la classe véhicule terrestre et les sousclasses vélo, moto, auto, camion. Dessiner également les métaclasses associés et toutes les relations d'héritage et d'instanciation ?