Chapitre 2

PRINCIPES DE BASE

Un langage à objets utilise les notions de *classe* et d'*instance*, que l'on peut comparer aux notions de type et de variable d'un langage tel que Pascal. La classe décrit les caractéristiques communes à toutes ses instances, sous une forme similaire à un type enregistrement (« record » Pascal). Une classe définit donc un ensemble de *champs*. De plus, une classe décrit un ensemble de *méthodes*, qui sont les opérations réalisables sur les instances de cette classe. Ainsi une classe est une entité autonome. Au lieu d'appliquer des procédures ou fonctions globales à des variables, on invoque les méthodes des instances. Cette invocation est souvent appelée *envoi de message*. De fait, on peut considérer que l'on envoie un message à une instance pour qu'elle effectue une opération, c'est-à-dire pour qu'elle détermine la méthode à invoquer.

On utilise souvent le terme d'*objet* à la place d'instance. Le terme « instance » insiste sur l'appartenance à une classe : on parle d'une instance d'une classe donnée. Le terme « objet » réfère de façon générale à une entité du programme (qui peut être une instance, mais aussi un champ, une classe, etc.).

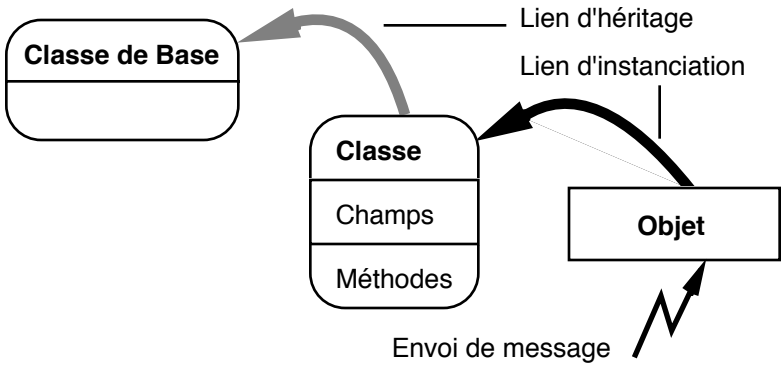

Figure 2 - Illustration des notions de base

L'*héritage* est la dernière notion de base des langages à objets : une classe peut être construite à partir d'une classe existante, dont elle étend ou modifie la description. Ce mécanisme de structuration est fondamental dans les langages à objets ; il est décrit dans la section 2.3 de ce chapitre.

La figure 2 décrit ces quatre notions de base, et introduit les conventions que nous utiliserons dans les autres figures.

2.1 CLASSES ET INSTANCES

La notion d'objet ou instance recouvre toute entité d'un programme écrit dans un langage à objets qui stocke un état et répond à un ensemble de messages. Cette notion est à comparer avec la notion usuelle de variable : une variable stocke un état, mais n'a pas la capacité par elle-même d'effectuer des traitements. On utilise pour cela dans les langages classiques des sous-programmes, fonctions ou procédures. Ceux-ci prennent des variables en paramètre, peuvent les modifier et peuvent retourner des valeurs.

Dans un langage classique, on définirait par exemple un type *Pile* et des procédures pour accéder à une pile ou la modifier :

- *Empiler*, qui prend une pile et une valeur à empiler ;
- *Dépiler*, qui prend une pile ;
- *Sommet*, qui prend une pile et retourne une valeur.

Cette séparation entre variables et procédures dans les langages classiques est la source de nombreux problèmes en ce qui concerne l'encapsulation : pour des raisons de sécurité de programmation, on ne souhaite pas que n'importe quelle procédure puisse accéder au contenu de n'importe quelle variable. On est alors amené à introduire la notion de *module*. Un module exporte des types, des variables et des procédures. De l'extérieur, les types sont *opaques* : leur implémentation n'est pas accessible. On ne peut donc manipuler les variables de ce type que par l'intermédiaire des procédures exportées par le module. À l'intérieur du module, l'implémentation des types est décrite, et est utilisable par les corps des procédures.

Ainsi, on peut encapsuler la notion de pile dans un module. Ce module exporte un type *Pile*, et les procédures *Empiler*, *Dépiler* et *Sommet*, dont l'implémentation n'est pas connue de l'extérieur. Par cette technique, un utilisateur du module ne pourra pas modifier une pile autrement que par les procédures fournies par le module, ce qui assure l'intégrité d'une pile.

Le module constitue donc le moyen de regrouper types et procédures, pour construire des types abstraits. Les langages Clu, Ada et ML, parmi d'autres, offrent de telles possibilités. Dans les langages qui n'offrent pas de modularité (Pascal, C, Lisp etc.), le programmeur doit faire preuve d'une grande rigueur pour reproduire artificiellement, c'est-à-dire sans aide du langage, l'équivalent de modules.

L'approche des langages à objets consiste à intégrer d'emblée la notion de variable et de procédures associées dans la notion d'objet. L'encapsulation est donc fournie sans mécanisme additionnel. De même qu'une variable appartient à un type dans

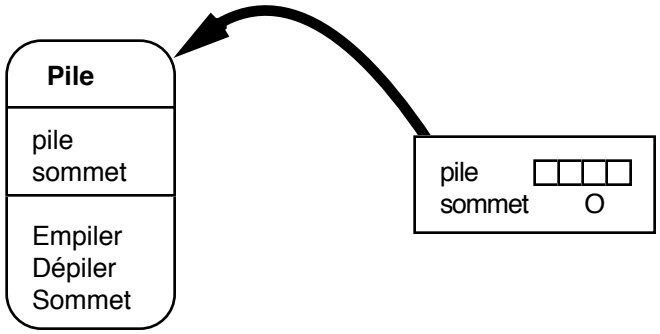

Figure 3 - La classe *Pile* et une instance

un langage classique, dans un langage à objets un objet appartient à une classe. La *classe* est à la fois un type et un module : elle contient une description de type, sous forme de *champs*, ainsi qu'un ensemble de procédures associées à ce type, appelées *méthodes*.

Définir une classe

On définira ainsi une classe *Pile* par :

- un état représentant la pile (tableau, liste, etc.), constitué de champs. Pour une représentation par tableau, on aura ainsi deux champs : le tableau lui-même et l'indice du sommet courant.
- la méthode *Empiler*, qui prend une valeur en paramètre.
- la méthode *Dépiler*.
- la méthode *Sommet*, qui retourne une valeur.

On crée des objets à partir d'une classe par le mécanisme d'*instanciation*. Le résultat de l'instanciation d'une classe est un objet, appelé instance de la classe (voir figure 3). L'instanciation est similaire à la création d'une variable d'un

type enregistrement. L'instance créée stocke un état constitué d'une valeur pour chacun de ses champs. Les champs sont eux-mêmes des objets. Un certain nombre de classes sont prédéfinies dans les langages, telles que la classe des entiers, des caractères, etc.

L'encapsulation, c'est-à-dire l'accès contrôlé aux objets, est assurée naturellement par les classes. Bien que différant d'un langage à l'autre, comme nous le verrons, on peut considérer dans un premier temps que les champs sont *privés* alors que les méthodes sont *publiques*. Ceci signifie que les champs sont visibles uniquement depuis le corps des méthodes de la classe, alors que les méthodes sont visibles de l'extérieur. Nous allons maintenant décrire le mécanisme d'invocation des méthodes.

2.2 MÉTHODES ET ENVOI DE MESSAGE

Dans les langages à objets, les objets stockent des valeurs, et les méthodes permettent de manipuler les objets. Ceci est comparable aux langages classiques, dans lesquels les variables stockent des valeurs et les procédures et fonctions permettent de manipuler les variables. Mais, contrairement aux procédures et fonctions qui sont des entités globales du programme, les méthodes appartiennent aux classes des objets. Au lieu d'appeler une procédure ou fonction globale, on invoque une méthode *d'un objet*. L'exécution de la méthode est alors réalisée dans le contexte de cet objet.

L'invocation d'une méthode est souvent appelée *envoi de message*. On peut en effet considérer que l'on envoie à un objet, par exemple une pile, un message, par exemple « empiler la valeur 20 ». Dans un langage classique, on appellerait la procédure *Empiler* avec comme paramètres la pile elle-même et la valeur 20.

Cette distinction est fondamentale. En effet, l'envoi de message implique que c'est le *receveur* (ici la pile) qui décide comment empiler la valeur 20, grâce à la méthode détenue par

sa classe. Au contraire, l'appel de procédure des langages classiques implique que c'est la procédure (ici *Empiler*) qui décide quoi faire de ses arguments, dans ce cas la pile et la valeur 20. En d'autres termes, la programmation impérative ou fonctionnelle privilégie le contrôle (procédures et fonctions) alors que la programmation par objets privilégie les données (les objets), et décentralise le contrôle dans les objets.

Le corps d'une méthode est exécuté dans le contexte de l'objet receveur du message. On a donc directement et naturellement accès aux champs et méthodes de l'objet receveur. C'est en fait seulement dans le corps des méthodes que l'on a accès aux parties privées de l'objet, c'est-à-dire en général ses champs (les langages diffèrent sur la définition des parties privées).

La définition conjointe des champs et des méthodes dans les classes est à la base du mécanisme d'héritage, que nous allons maintenant décrire.

2.3 L'HÉRITAGE

La notion d'héritage est propre aux langages à objets. Elle permet la définition de nouvelles classes à partir de classes existantes. Supposons que l'on veuille programmer le jeu des Tours de Hanoi. On dispose de trois tours ; sur la première sont empilés des disques de taille décroissante. On veut déplacer ces disques sur l'une des deux autres tours en respectant les deux règles suivantes :

- on ne peut déplacer qu'un disque à la fois ;
- on ne peut poser un disque sur un disque plus petit.

Le comportement d'une tour est similaire à celui d'une pile : on peut empiler ou dépiler des disques. Cependant on ne peut empiler un disque que s'il est de diamètre inférieur au sommet courant de la tour.

Si l'on utilise un langage classique qui offre l'encapsulation, on est confronté à l'alternative suivante :

- utiliser une pile pour représenter chaque tour, et s'assurer que chaque appel de la procédure *Empiler* est précédé d'un test vérifiant la validité de l'empilement.
- créer un nouveau module, qui exporte un type opaque *Tour* et les procédures *Empiler*, *Dépiler*, et *Sommet*. L'implémentation de *Tour* est une pile ; la procédure *Empiler* réalise le contrôle nécessaire avant d'empiler un disque. Les procédures *Dépiler* et *Sommet* appellent leurs homologues de la pile.

Aucune de ces deux solutions n'est satisfaisante. La première ne fournit pas d'abstraction correspondant à la notion de tour. La seconde est la seule acceptable du point de vue de l'abstraction mais présente de multiples inconvénients :

- Il faut écrire des procédures inutiles : *Dépiler* du module *Tour* ne fait qu'appeler *Dépiler* du module *Pile* ;
- Si l'on ajoute une fonction *Profondeur* dans le module *Pile*, elle ne sera accessible pour l'utilisateur de *Tour* que si l'on définit également une fonction *Profondeur* dans le module *Tour* comme on l'a fait pour *Dépiler* ;
- Le problème est encore plus grave si l'on décide d'enlever la fonction *Profondeur* du module *Pile* : la fonction *Profondeur* de *Tour* appelle maintenant une fonction qui n'existe plus ;
- Il n'est pas possible d'accéder directement à l'implémentation de la pile dans le module *Tour*, à cause de l'encapsulation. On ne peut pas ajouter la fonction *Profondeur* dans *Tour* sans définir une fonction équivalente dans *Pile*.

Ce que l'on cherche en réalité est la spécialisation d'une pile pour en faire une tour, en créant un lien privilégié entre les modules *Pile* et *Tour*. C'est ce que permet l'héritage par la définition d'une classe *Tour* qui hérite de *Pile*.

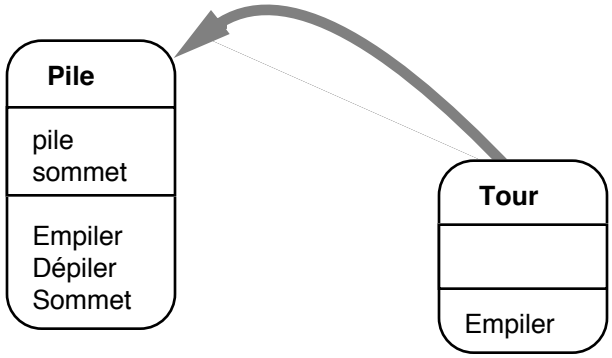

Figure 4 - La classe *Tour* hérite de la classe *Pile*

Définir une classe par héritage

Lorsqu'une classe *B* hérite d'une classe *A*, les instances de *B* contiennent les mêmes champs que ceux de *A*, et les méthodes de *A* sont également disponibles dans *B*. De plus, la sous-classe *B* peut :

- définir de nouveaux champs, qui s'ajoutent à ceux de sa classe de base *A* ;
- définir de nouvelle méthodes, qui s'ajoutent à celles héritées de *A* ;
- redéfinir des méthodes de sa classe de base *A*.

Enfin, les méthodes définies ou redéfinies dans *B* ont accès aux champs et méthodes de *B*, mais aussi à ceux de *A*.

Ces propriétés montrent le lien privilégié qui unit *B* à *A*. En particulier, si l'on ajoute des champs et/ou des méthodes à *A*, il n'est pas nécessaire de modifier *B*. Il en est de même si l'on retire des champs et/ou des méthodes de *A*, sauf bien sûr s'ils étaient utilisés dans les méthodes définies ou redéfinies dans *B*.

Dans notre exemple, on se bornera à redéfinir la méthode *Empiler*, pour faire le contrôle de la taille des disques et appeler la méthode *Empiler* de *Pile* si l'opération est valide (voir figure 4). Dans ce cas, on dira que l'on a spécialisé la classe *Pile* car on a seulement redéfini l'une de ses méthodes. Si l'on définissait de nouvelles méthodes dans la classe *Tour* (par exemple initialiser la tour avec *N* disques de taille décroissante), on aurait enrichi la classe *Pile*. Ce simple exemple montre déjà que l'héritage peut servir à réaliser deux opérations : la spécialisation et l'enrichissement.

L'arbre d'héritage

Telle que nous l'avons présentée, la notion d'héritage induit une forêt d'arbres de classes : une classe *A* représentée par un nœud d'un arbre a pour sous-classes les classes représentées par ses fils dans l'arbre. Les racines des arbres de la forêt sont les classes qui n'héritent pas d'une autre classe.

Si *C* hérite de *B* et *B* hérite de *A*, on dira par extension que *C* hérite de *A*. On dira indifféremment :

- *B* hérite de *A*
- *B* est une *sous-classe* de *A*
- *B* dérive de *A*
- *B* est une *classe dérivée* de *A*
- *A* est une (la) *superclasse* de *B*
- *A* est une (la) *classe de base* de *B*

Dans les deux dernières phrases, on emploie l'article défini pour indiquer que *A* est l'antécédent direct de *B* dans l'arbre d'héritage.

Certains langages imposent une classe de base unique pour toutes les autres, appelée souvent *Objet*. Dans ce cas, la relation d'héritage définit un arbre et non une forêt. Par abus de langage, on parle dans tous les cas de l'*arbre d'héritage*.

2.4 L'HÉRITAGE MULTIPLE

L'héritage que nous avons défini est dit *héritage simple* car une classe a au plus une classe de base. Une généralisation de l'héritage simple, l'*héritage multiple*, consiste à autoriser une classe à hériter directement de plusieurs classes.

Ainsi si la classe *B* hérite de *A1, A2, … An*, on a les propriétés suivantes :

- les champs des instances de *B* sont l'union des champs de *A1, A2, … An* et des champs propres de *B* ;
- les méthodes définies pour les instances de B sont l'union des méthodes définies dans *A1, A2, … An* et des méthodes définies dans *B*. *B* peut également redéfinir des méthodes de ses classes de base.

L'arbre ou la forêt d'héritage devient alors un graphe. Pour éviter des définitions récursives on interdit d'avoir des cycles dans le graphe d'héritage. En d'autres termes, on interdit à une classe d'être sa propre sous-classe, même indirectement.

Cette extension, apparemment simple, cache en fait de multiples difficultés, qui ont notamment trait aux problèmes de collision de noms dans l'ensemble des champs et méthodes héritées. Certaines de ces difficultés ne pourront être développées que dans la description plus détaillée des chapitres suivants.

Une difficulté intrinsèque de l'héritage multiple est la gestion de l'héritage répété d'une classe donnée : si *B* et *C* héritent de *A* par un héritage simple, et *D* hérite de *B* et de *C*, alors *D* hérite deux fois de *A*, par deux chemins *D*-*B*-*A* et *D*-*C*-*A* dans le graphe d'héritage (figure 5). Faut-il pour autant qu'une instance de *D* contienne deux fois les champs définis dans *A*, ou bien faut-il les partager ?

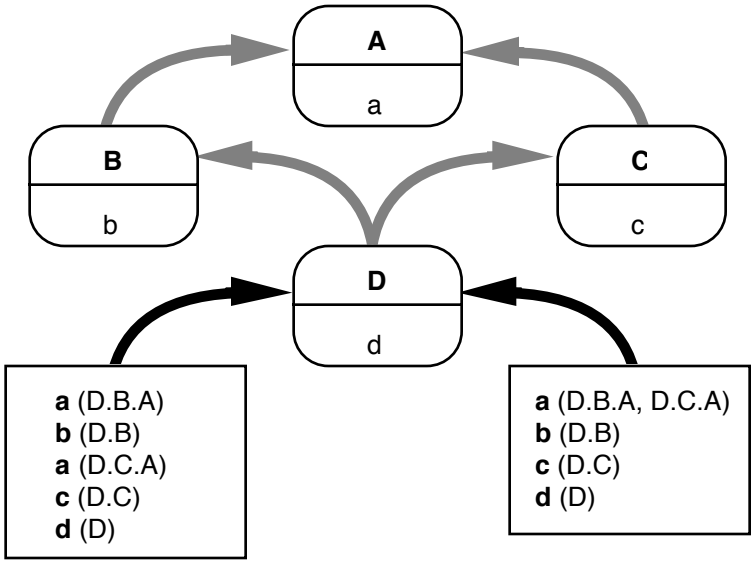

Figure 5 - Deux interprétations de l'héritage multiple

Selon les situations, on souhaitera l'une ou l'autre solution, comme le montrent les deux exemples suivants.

Soit *A* la classe des engins à moteur, qui contient comme champs les caractéristiques du moteur. Soit *B* la classe des automobiles et *C* la classe des grues. *D* est donc la classe des grues automobiles. Les deux interprétations de l'héritage multiple sont possibles : si l'on hérite deux fois de la classe des engins à moteur, la grue automotrice a deux moteurs : l'un pour sa partie automobile, l'autre pour sa partie grue. Si l'on hérite une seule fois de l'engin à moteur, on a un seul moteur qui sert à la fois à déplacer le véhicule et à manœuvrer la grue.

Soit maintenant *A* la classe des objets mobiles, contenant les champs position et vitesse. Soit *B* la classe des bateaux, qui contient par exemple un champ pour le tonnage, et soit *C* la classe des objets propulsés par le vent, qui contient un champ

pour la surface de la voile. *D* est alors la classe des bateaux à voile. Une instance de *D* a bien entendu une seule position et une seule vitesse, et dans ce cas il faut partager les champs de *A* hérités par différents chemins.

L'héritage multiple n'offre pas de réponse satisfaisante à ce problème. La première interprétation correspond à une composition de classes, la seconde à une combinaison. Le mécanisme de l'héritage est certainement imparfait pour capturer à la fois les notions de spécialisation, d'enrichissement, de composition et de combinaison. Mais l'héritage n'est pas le seul moyen de définir des classes ! L'un des pièges qui guettent le programmeur utilisant un langage à objets est la mauvaise utilisation de l'héritage. La question qu'il faut se poser à chaque définition de classe est la suivante : faut-il que *B* hérite de *A*, ou bien *B* doit-elle contenir un champ qui soit une instance de *A* ? *B* est-il une sorte de *A* ou bien *B* contient-il un *A* ? Nous reviendrons au chapitre 6 sur la pratique de la programmation par objets, et en particulier sur ces problèmes.

2.5 LE POLYMORPHISME

La notion de *polymorphisme* recouvre la capacité pour un langage de décrire le comportement d'une procédure de façon indépendante de la nature de ses paramètres. Ainsi la procédure qui échange les valeurs de deux variables est polymorphe si l'on peut l'écrire de façon indépendante du type de ses paramètres. De façon similaire la procédure *Empiler* est polymorphe si elle ne dépend pas du type de la valeur à empiler.

Comme le polymorphisme est défini par rapport à la notion de type, il ne concerne que les langages typés. On distingue plusieurs types de polymorphisme, selon la technique employée pour sa mise en œuvre :

- *Le polymorphisme ad hoc* consiste à écrire plusieurs fois le corps de la procédure, pour chacun des types de paramètres souhaités. C'est ce que l'on appelle souvent la *surcharge* : on

peut définir *Echanger (Entier, Entier)* et *Echanger (Disque, Disque)* de façon indépendante, de même que *Empiler (Pile, Entier)* et *Empiler (Pile, Disque)*. Ce type de polymorphisme est généralement résolu de façon statique (à la compilation). Il utilise le système de types pour déterminer quelle incarnation de la procédure il faut appeler, en fonction des types effectifs des paramètres de l'appel.

- *Le polymorphisme d'inclusion* est fondé sur une relation d'ordre partiel entre les types : si le type *B* est inférieur selon cette relation d'ordre au type *A*, alors on peut passer un objet de type *B* à une procédure qui attend un paramètre de type *A*. Dans ce cas la définition d'une seule procédure définit en réalité une famille de procédures pour tous les types inférieurs aux types mentionnés comme paramètres. Si *Entier* et *Disque* sont tous deux des types inférieurs à *Objet*, on pourra définir les procédure *Echanger (Objet, Objet)* et *Empiler (Pile, Objet)*.

- *Le polymorphisme paramétrique* consiste à définir un modèle de procédure, qui sera ensuite incarné avec différents types. Il est implémenté par la *généricité*, qui consiste à utiliser des types comme paramètres. Ainsi si l'on définit la procédure *Echanger (<T>, <T>)*, on pourra l'incarner avec *<T> = Entier* ou *<T> = Disque*. On peut faire de même pour *Empiler (Pile, <T>)*.

Ces trois types de polymorphisme existent dans divers langages classiques. En Pascal le polymorphisme existe mais n'est pas accessible à l'utilisateur ; on ne peut donc pas le qualifier puisque sa mise en œuvre est implicite. Par exemple les opérateurs arithmétiques sont polymorphes : l'addition, la soustraction, etc. s'appliquent aux entiers, aux réels, et même aux ensembles. De même, les procédures d'entrée-sortie *read* et *write* sont polymorphes puisqu'elles s'appliquent à différents types de paramètres.

Ada offre un polymorphisme ad hoc par la possibilité de surcharge des noms de procédures et des opérateurs. Il offre

également un polymorphisme paramétrique par la possibilité de définir des fonctions génériques. En revanche le polymorphisme d'inclusion est limité car très peu de types sont comparables par une relation d'ordre.

Le polymorphisme dans les langages à objets

La définition du polymorphisme est dépendante de la notion de type. Pourtant, tous les langages à objets ne sont pas typés. Un *langage à objets typé* est un langage à objets dans lequel chaque classe définit un type, et dans lequel on déclare explicitement les types des objets que l'on utilise. Les langages à objets typés fournissent naturellement le polymorphisme ad hoc et le polymorphisme d'inclusion. Certains langages offrent le polymorphisme paramétrique mais il ne fait pas partie des principes de base présentés dans ce chapitre.

Le polymorphisme ad hoc provient de la possibilité de définir dans deux classes indépendantes (c'est-à-dire n'ayant pas de relation d'héritage) des méthodes de même nom. Le corps de ces méthodes est défini indépendamment dans chaque classe, mais du point de vue de l'utilisateur, on peut envoyer le même message à deux objets de classes différentes. Ce polymorphisme ad hoc est intrinsèque aux langages à objets : il ne nécessite aucun mécanisme particulier, et découle simplement du fait que chaque objet est responsable du traitement des messages qu'il reçoit.

La définition de plusieurs méthodes de même nom dans une même classe ou dans des classes ayant une relation d'héritage est une forme de polymorphisme ad hoc qui n'est pas implicite dans les langages à objets, bien que la plupart d'entre eux offrent cette possibilité de surcharge. De plus, la redéfinition d'une méthode dans une classe dérivée, avec le même nom et les mêmes paramètres que dans la classe de base, ne constitue pas une surcharge mais une *redéfinition* de méthode, comme nous l'avons vu dans la description de l'héritage (section 2.5).

Les langages à objets disposent également naturellement d'un polymorphisme d'inclusion que l'on appelle aussi *polymorphisme d'héritage*. En effet, la hiérarchie des classes (dans le cas de l'héritage simple) induit un ordre partiel : si *B* hérite de *A* (directement ou indirectement), alors *B* est inférieur à *A*. Toute méthode de *A* est alors applicable à un objet de classe *B* : c'est ainsi que l'on a défini l'héritage des méthodes. Le polymorphisme d'héritage nous permet donc d'appliquer la méthode *Sommet* de la classe *Pile* à un objet de la classe *Tour*, puisque *Tour* est une sous-classe de *Pile*.

Le polymorphisme d'héritage s'applique également à l'héritage multiple, en définissant une relation d'ordre partiel compatible avec le graphe d'héritage de la façon suivante : une classe *B* est inférieure à une classe *A* si et seulement si il existe un chemin orienté de *B* vers *A* dans le graphe d'héritage. Le graphe étant sans cycle, on ne peut avoir à la fois un chemin orienté de *A* vers *B* et un chemin orienté de *B* vers *A*, ce qui assure la propriété d'antisymétrie.

Le polymorphisme d'héritage s'applique non seulement au receveur des messages, mais également au passage de paramètres des méthodes : si une méthode prend un paramètre formel de classe *A*, on peut lui passer un paramètre réel de classe *B* si *B* est inférieur à *A*. Ainsi la méthode *Empiler* prend un paramètre de classe *Entier*. On peut lui passer un paramètre de classe *Disque*, si *Disque* hérite de *Entier*.

Liaison statique et liaison dynamique

Le polymorphisme d'héritage interdit aux langages à objets un typage exclusivement statique : un objet déclaré de classe *A* peut en effet contenir, à l'exécution, un objet d'une sous-classe de *A*. Les langages à objets sont donc au mieux fortement typés, ce qui a des conséquences importantes pour la compilation de ces langages. Dans un langage à typage statique, le compilateur peut déterminer quelle méthode de quelle classe est effectivement appelée lors d'un envoi de message : on appelle cette technique la *liaison statique*.

Lorsque le typage statique ne peut être réalisé, on doit avoir recours à la *liaison dynamique*, c'est-à-dire la détermination à l'exécution de la méthode à appeler. La liaison dynamique fait perdre un avantage important des langages compilés : l'efficacité du code engendré par le compilateur. La liaison dynamique doit aussi être utilisée dans les langages non typés, car l'absence de système de types interdit toute détermination a priori de la méthode invoquée par un envoi de message. Dans les deux cas, nous verrons les techniques employées pour rendre la liaison dynamique efficace.

Les liens étroits entre polymorphisme, typage, et mode de liaison déterminent en grande partie les compromis réalisés par les différents langages à objets entre puissance d'expression du langage, sécurité de programmation, et performance à l'exécution. De ce point de vue, il n'existe pas aujourd'hui de langage idéal, et il est probable qu'il ne puisse en exister.

2.6 LES MÉTACLASSES

Nous avons défini jusqu'à présent la notion d'objet de façon assez vague : un objet doit appartenir à une classe. Certains langages permettent de considérer une classe comme un objet ; en temps qu'objet, cette classe doit donc être l'instance d'une classe. On appelle la classe d'une classe une *métaclasse* (voir figure 6).

La description que nous avons donnée d'une classe ressemble effectivement à celle d'un objet : une classe contient la liste des noms des champs de ses instances et le dictionnaire des méthodes que l'on peut invoquer sur les instances. La liste des champs d'une métaclasse a donc deux éléments : la liste des noms de champs et le dictionnaire des méthodes. L'instanciation est une opération qui est réalisée par une classe ; c'est donc une méthode de la métaclasse.

Une métaclasse peut également stocker d'autres champs et d'autres méthodes. Ainsi, l'arbre d'héritage étant une relation

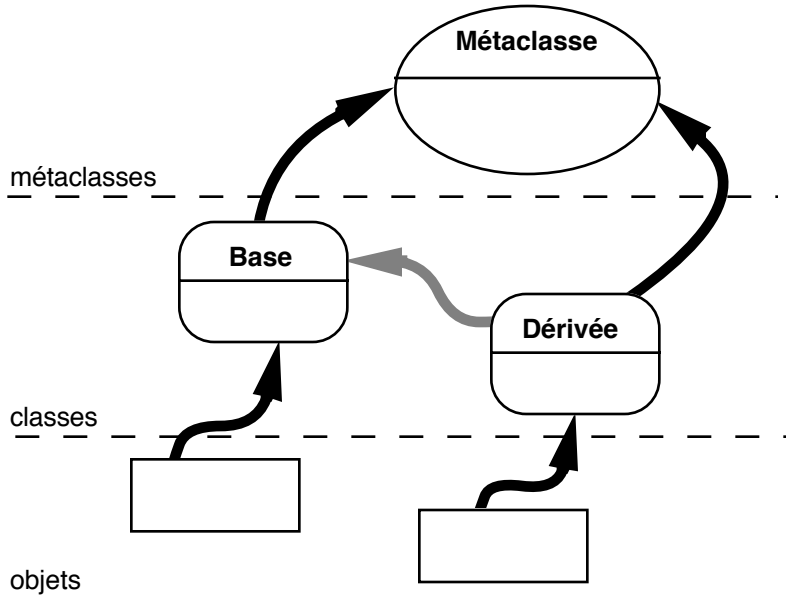

Figure 6 - La notion de métaclasse

entre les classes, chaque classe contient un champ qui désigne sa classe de base (représenté par les flèches grises épaisses dans les figures). Une méthode de la métaclasse permet de tester si une classe est sous-classe d'une autre classe.

Plusieurs modèles de métaclasses existent. Le plus simple consiste à avoir une seule métaclasse (appelée par exemple *Métaclasse*). Le plus complet permet de définir arbitrairement des métaclasses. Cela autorise par exemple la redéfinition de l'instanciation ou l'ajout de méthodes de classes (définies dans la métaclasse). Un modèle intermédiaire, celui de Smalltalk-80, prévoit exactement une métaclasse par classe. L'environnement de programmation se charge de créer automatiquement la métaclasse pour toute classe créée par le programmeur. Ce dernier peut définir des *méthodes de classes*, qui sont stockées dans le dictionnaire de la métaclasse. Cette approche rend les métaclasses pratiquement transparentes pour le programmeur, et offre un compromis satisfaisant dans la plupart des applications.

Dans tous les cas, la notion de métaclasse induit une régression à l'infini : en effet, une métaclasse est une classe, donc un objet, et a donc une classe ; cette classe a donc une métaclasse, qui est un objet, etc. Cette régression est court-circuitée par une boucle dans l'arbre d'instanciation. Par exemple, la classe *Métaclasse* est sa propre métaclasse.

Les métaclasses ont deux applications bien différentes. La première est de permettre une définition méta-circulaire d'un langage et de rendre accessible ses propres structures d'exécution. On appelle cela la *réification*. Cette propriété existe également en Lisp et permet d'écrire très simplement un interprète Lisp en Lisp. Un langage réifié permet également de construire facilement des moyens d'introspection pour aider à la mise au point des programmes : trace des envois de message et des invocations de méthodes, trace des changements de valeur des variables, etc.

La deuxième application des métaclasses est de permettre la construction dynamique de classes. Prenons l'exemple d'une application graphique interactive dans laquelle l'utilisateur peut créer de nouveaux objets graphiques utilisables comme les objets primitifs (cercles, rectangles, etc.). Chaque nouvel objet graphique, lorsqu'il est transformé en modèle, donne lieu à la création d'une nouvelle classe. Cette nouvelle classe est créée par instanciation d'une métaclasse existante. En l'absence de métaclasses, il faudrait d'une façon ou d'une autre simuler ce mécanisme, ce qui peut être fastidieux.

Disposer de métaclasses dans un langages à objets signifie que l'on peut dynamiquement (à l'exécution) définir de nouvelles classes et modifier des classes existantes. Cela interdit donc tout typage statique, et c'est la raison pour laquelle les métaclasses ne sont disponibles que dans les langages non typés. Certains langages typés utilisent néanmoins implicitement des métaclasses, en autorisant par exemple la redéfinition des méthodes d'instanciation. Il est également possible de définir des objets qui jouent le rôle des métaclasses pour la représentation, à l'exécution, de l'arbre d'héritage. Mais la

pleine puissance des métaclasses reste réservée aux langages non typés.