Conception et programmation orientées objet

Bertrand Meyer

Traduit de l'anglais par Pierre Jouvelot

 ISBN : 978-2-212-12270-1

EYROLLES

7

La structure statique : les classes

Grâce à l'examen du contexte de génie logiciel dans lequel se place notre étude, vous avez découvert les raisons qui militent pour la recherche d'une meilleure approche de conception modulaire : la réutilisabilité et l'extensibilité. Vous avez vu les limitations des approches traditionnelles : des architectures centralisées qui limitent la flexibilité. Vous avez découvert la théorie qui sous-tend l'approche orientée objet : les types abstraits de données. Vous en savez maintenant assez sur ce qui pose problème. Passons à la solution !

Ce chapitre et ceux de la partie C introduisent les techniques fondamentales de l'analyse, de la conception et de la programmation orientées objet. Tout au long de ceux-ci, nous développerons les notations nécessaires.

Notre première tâche consiste à examiner les blocs de base de notre construction : les classes.

7.1 LE SUJET N'EST PAS LES OBJETS

Quel est le concept central de la technologie objet ?

Réfléchissez à deux fois avant de répondre "objet". Les objets sont utiles, mais ils ne sont pas nouveaux. Cobol a toujours eu des structures ; Pascal a toujours proposé des enregistrements ; depuis que le premier programmeur C a écrit la première structure C, l'humanité a eu des objets.

Les objets restent certes importants pour décrire l'exécution d'un système orienté objet. Mais la base sur laquelle repose l'ensemble de la technologie orientée objet est la **classe**, que nous avons découverte dans le chapitre précédent. En voici, à nouveau, la définition :

Les objets sont étudiés en détail dans le prochain chapitre.

Définition : classe

Une classe est un type abstrait de données muni d'une implémentation éventuellement partielle.

Les types abstraits de données sont une notion mathématique, adaptée à l'étape de spécification (appelée aussi analyse). Parce qu'elle conduit à des implémentations, partielles ou totales, la notion de classe établit le lien nécessaire avec la construction logicielle —

conception et implémentation. Rappelez-vous qu'une classe est dite effective si son implémentation est totale, et retardée sinon.

Comme un ADT, une classe est un type : elle décrit un ensemble de structures de données possibles, appelées les ***instances*** de la classe. Les types abstraits de données ont aussi des instances ; la différence vient de ce qu'une instance d'un ADT est un élément purement mathématique (un élément d'un ensemble mathématique), alors qu'une instance d'une classe est une structure de données qui peut être représentée dans la mémoire d'un ordinateur et manipulée par un système logiciel.

Par exemple, si nous avions défini une classe `STACK` en partant de la spécification d'ADT du chapitre précédent et en ajoutant des informations de représentation adéquates, les instances de cette classe seraient des structures de données représentant des piles individuelles. Un autre exemple, développé dans le reste de ce chapitre, est une classe modélisant la notion de `POINT` dans un espace à deux dimensions, avec une représentation adéquate ; une instance de cette classe est une structure de données représentant un point. Dans l'une des représentations étudiées ci-dessous, la représentation cartésienne, chaque instance de `POINT` est un enregistrement ayant deux champs représentant les coordonnées horizontale et verticale, `x` et `y`, d'un point.

La définition d'une "classe" conduit par ailleurs à la définition d'un "objet". Un objet est simplement une instance d'une classe. Par exemple, une instance de classe `STACK` — une structure de données représentant une pile particulière — est un objet ; c'est aussi le cas d'une instance de la classe `POINT`, représentant un point particulier dans l'espace à deux dimensions.

Les textes logiciels qui servent à produire des systèmes sont les classes. Les objets sont une notion n'ayant un sens qu'à l'exécution : ils sont créés et manipulés par le logiciel durant son exécution.

Ce chapitre est consacré au mécanisme de base permettant d'écrire des éléments logiciels et de les combiner en systèmes ; en conséquence, il est centré sur les classes. Dans le prochain chapitre, nous explorerons les structures créées à l'exécution par un système orienté objet ; il nous faudra étudier certaines questions d'implémentation et revenir précisément sur la nature des objets.

7.2 ÉVITER LA CONFUSION CLASSIQUE

La prochaine section, pour les lecteurs qui n'aiment pas le rabâchage, est "LE RÔLE DES CLASSES", 7.3, page 173.

Une classe est un modèle, et un objet est une instance d'un tel modèle. Cette propriété est tellement évidente que les définitions précédentes ne devraient pas avoir besoin de commentaires supplémentaires ; mais elle a été source de tant de confusion dans certaines publications de vulgarisation que nous nous devons de prendre quelques instants pour clarifier ce qui devrait être une évidence. (Si vous pensez être armé contre un tel danger et si vous avez échappé à un enseignement approximatif des concepts orientés objet, vous pouvez sauter cette section, car elle ne fait que renforcer l'évidence.)

Que penseriez-vous de ceci ?

> Parmi les pays d'Europe, nous pouvons identifier l'Italien. L'Italien a une chaîne de montagnes qui le traverse du nord au sud et il aime la bonne cuisine, souvent à base d'huile d'olive. Son climat est de type méditerranéen, et il parle un langage admirablement musical.

Si quelqu'un parlait ou écrivait de cette manière, vous pourriez suspecter une nouvelle maladie neurologique, l'incapacité à faire la distinction entre les catégories (comme la nation italienne) et les membres individuels de ces catégories (comme les individus italiens), ce qui justifierait de donner au conducteur de l'ambulance l'adresse de la clinique du Dr. Sacks à New York.

Voir par exemple Oliver Sacks, ``The Man Who Mistook His Wife for a Hat and Other Clinical Tales", Harper Perennials, 1991.

Pourtant, dans la littérature consacrée au logiciel orienté objet, de telles confusions sont fréquentes. Considérez l'extrait suivant d'un livre connu traitant de l'analyse OO, et qui utilise l'exemple d'un système interactif pour apprendre à identifier les abstractions :

> *Il est possible de motiver l'introduction d'un objet "utilisateur" dans un espace de problème même si le système n'a pas besoin de conserver d'information concernant cet utilisateur. Dans cette situation, le système n'a pas besoin de l'habituel numéro d'identification, du nom, des droits d'accès ou autres. Mais le système doit contrôler l'utilisateur, en répondant à ses requêtes et en fournissant des réponses à jour. Et donc, de par l'existence de ces services demandés par la chose réelle (dans ce cas, l'utilisateur), nous devons ajouter, dans le modèle de l'espace du problème, un objet correspondant.*

[Coad 1990], *3.3.3, page 67.*

Dans la foulée, ce texte utilise les mots *objet*, *utilisateur* et *chose* dans deux sens appartenant à des niveaux d'abstraction différents :

- un utilisateur typique d'un système interactif en cours d'élaboration,
- le *concept* d'utilisateur en général.

L'exercice E7.1, page 218, vous demande de clarifier chaque utilisation du mot "objet" dans ce texte.

Bien qu'il s'agisse probablement d'un abus de terminologie (une peccadille que peu de gens peuvent affirmer ne pas avoir commise) plutôt qu'une confusion réelle de la part de l'auteur, cet exemple est malheureusement représentatif de la manière dont une partie de la littérature distingue le modèle de l'instance. Si vous débutez l'étude d'une nouvelle méthode avec ce genre de confusion élémentaire, apparente ou réelle, il est peu probable que vous ayez un aperçu rationnel de la construction logicielle.

Le moule et l'instance

Prenez ce livre — la copie que vous êtes en train de lire. Considérez-le comme un objet au sens commun du terme. Il présente ses propres caractéristiques individuelles : l'exemplaire peut être tout neuf, ou avoir déjà été parcouru par d'autres lecteurs ; vous avez peut-être écrit votre nom sur la première page ; ou il appartient à une bibliothèque et possède un code d'identification local imprimé sur sa tranche.

Les propriétés de base du livre, cependant, comme son titre, son éditeur, son auteur et son contenu, sont définies par une description générale qui s'applique à toutes les copies individuelles : le livre est intitulé *Construction Logicielle Orientée Objet*, il (NdT : sa version anglaise) est publié par Prentice-Hall, il concerne la méthode orientée objet, et ainsi de suite. Cet ensemble de propriétés ne définit pas un objet, mais une classe d'objets (appelée également, dans ce cas, le **type** de ces objets ; pour l'instant, les notions de type et de classe peuvent être considérées comme synonymes).

Appelons `OOSC` cette classe. Elle définit un certain moule. Les objets construits selon ce moule, comme votre copie du livre, sont appelés ***instances*** de la classe. Un autre exemple de moule serait le moulage en plâtre qu'un sculpteur créerait pour obtenir une version en négatif d'une conception pour un ensemble de statues identiques ; toute statue dérivée du moulage est une instance du moule.

Page 168.

Dans la citation du *Nom de la rose* qui figure au début de la partie C, le maître explique comment il a pu déterminer, à partir des traces dans la neige, que Brownie, le cheval des Abbott, avait marché ici auparavant. Brownie est une instance de la classe de tous les chevaux. Le signe sur la neige, bien qu'imprimé par une instance particulière, ne contient que l'information permettant de déterminer la classe (cheval), pas l'identité (Brownie). Puisque la classe, comme le signe, identifie tous les chevaux plutôt qu'un cheval particulier, l'extrait l'appelle également un signe.

Les mêmes concepts s'appliquent aux objets logiciels. Vous écrirez dans vos systèmes logiciels la description des classes, comme une classe `LINKED_STACK` décrivant les propriétés des piles dans une certaine représentation. Toute exécution particulière de votre système pourra utiliser ces classes pour créer des objets (les structures de données) ; chacun de ces objets est dérivé d'une classe, et est appelé une **instance** de cette classe. Par exemple, l'exécution peut créer un objet de liste chaînée, dérivée de la description donnée dans la classe `LINKED_STACK` ; un tel objet est une instance de la classe `LINKED_STACK`.

La classe est un texte logiciel. Elle est statique ; en d'autres termes, elle existe indépendamment de toute exécution. A contrario, un objet dérivé de cette classe est une structure de données créée dynamiquement, qui existe seulement dans la mémoire d'un ordinateur durant l'exécution d'un système.

Tout cela est, bien sûr, en accord avec l'étude précédente sur les types abstraits de données : quand nous avons spécifié `STACK` par un ADT, nous n'avons pas décrit une pile particulière, mais la notion générale de pile, un moule dont on peut dériver des instances individuelles à volonté.

Les phrases "`x` est une instance de `T`" et "`x` est un objet de type `T`" seront considérées comme synonymes dans cette étude.

Voir "Instances", page 460.

Avec l'introduction de l'héritage, il nous faudra distinguer entre les *instances directes* d'une classe (construites à partir du modèle exact défini par la classe) et ses *instances* au sens plus général (les instances directes de la classe ou d'une de ses spécialisations).

Métaclasses

Pourquoi tant de livres et d'articles confondent-ils deux notions aussi clairement distinctes que celles de classe et d'objet ? Une raison — sans être une excuse — est l'attrait pour le mot "objet", un mot simple de la langue de tous les jours. Mais il est traître. Comme nous l'avons déjà vu lors de l'étude de l'intégration, bien que certains des objets (les instances de classes) manipulés par les systèmes OO soient des représentations informatiques d'objets dans le sens usuel du terme, comme des documents, des comptes en banque ou des avions, beaucoup d'autres n'ont pas d'existence en dehors du logiciel ; on trouve là, en particulier, les objets introduits à des fins de conception et d'implémentation, les instances de classes comme un état `STATE` ou une liste `LINKED_LIST`.

Une autre source possible de confusion entre objets et classes est que, dans certains cas, nous devons traiter les classes comme des objets. Ce besoin ne se présente que dans certains contextes spéciaux, et est principalement rencontré par les développeurs d'environnements de développement orientés objet. Par exemple, un compilateur ou un interpréteur pour un langage orienté objet manipulera des structures de données représentant des classes écrites dans ce langage. La même chose se retrouve dans d'autres outils comme un navigateur (un outil utilisé

pour retrouver des classes et connaître leurs propriétés) ou un système de gestion de configuration. Si vous produisez de tels outils, vous créerez des objets qui représentent des classes.

> Pour reprendre une analogie utilisée auparavant, nous pouvons comparer cette situation à celle d'un employé de Prentice-Hall chargé de préparer le catalogue des titres sur le génie logiciel. Pour l'auteur du catalogue, OOSC, le concept qui se trouve derrière ce livre, est un objet — une instance d'une classe "entrée de catalogue". Au contraire, pour le lecteur de ce livre, ce concept est une classe, dont l'exemplaire particulier du lecteur est une instance.

Certains langages orientés objet, en particulier Smalltalk, ont introduit une notion de **métaclasse** pour gérer ce genre de situation. Une métaclasse est une classe dont les instances sont elles-mêmes des classes — ce que le *Nom de la rose* appelait des "signes de signe".

Nous éviterons, cependant, dans cette présentation, de recourir aux métaclasses, car elles posent plus de problèmes qu'elles n'apportent de solutions. En particulier, l'ajout des métaclasses rend difficile la vérification statique de type, condition nécessaire pour produire du logiciel fiable. Les applications majeures des métaclasses sont, d'ailleurs, obtenues plus facilement via d'autres mécanismes :

- Vous pouvez utiliser les métaclasses pour introduire dans toutes les classes, ou seulement dans certaines, un ensemble de caractéristiques. Vous obtiendrez le même résultat en arrangeant la structure d'héritage de façon que toutes les classes soient descendantes d'une classe générale adaptable, `ANY`, qui contiendra les déclarations des caractéristiques universelles.

"Classes universelles", page 562.

- Quelques opérations peuvent être considérées comme caractérisant une classe plutôt que ses instances, justifiant leur inclusion dans l'ensemble des caractéristiques d'une métaclasse. Mais ces opérations sont rares et connues ; la plus évidente est la création d'un objet, suffisamment importante pour mériter une construction spéciale du langage, l'instruction de création. (D'autres opérations similaires, comme la duplication d'un objet, seront prises en compte par les caractéristiques de la classe `ANY`.)

Voir "L'instruction de création", page 233.

- Il reste l'utilisation des métaclasses pour obtenir de l'information sur une classe, comme peut le vouloir un navigateur : le nom de la classe, la liste de ses caractéristiques, la liste de ses parents, la liste de ses fournisseurs. Mais nous n'avons pas besoin de métaclasse pour obtenir cela. Il suffira de concevoir une classe bibliothèque, `E_CLASS`, d'une manière telle que chaque instance de `E_CLASS` représentera une classe et ses propriétés. Quand nous créerons une telle instance, nous passerons à l'instruction de création un argument représentant une certaine classe `C` ; puis, en appliquant les diverses caractéristiques de `E_CLASS` à cette instance, nous pourrons tout apprendre sur `C`.

En pratique, donc, nous pouvons éluder le concept additionnel de métaclasse. Mais, même dans une méthode, un langage ou un environnement qui offrent cette notion, la présence de métaclasses ne justifie pas la confusion entre les moules et leurs instances — les classes et les objets.

7.3 LE RÔLE DES CLASSES

Après avoir pris soin d'éliminer une confusion absurde, mais fréquente et dommageable, nous pouvons maintenant revenir aux propriétés centrales des classes et, en particulier, étudier pourquoi elles sont si importantes pour la technologie objet.

Pour comprendre l'approche orientée objet, il est essentiel de bien comprendre que les classes jouent deux rôles que les approches pré-OO avaient toujours traités comme distincts : le module et le type.

Les modules et les types

Les langages de programmation et autres notations utilisées en développement logiciel (langage de conception, langage de spécification, notation graphique pour l'analyse) introduisent toujours une notion de module et un système de type.

Voir le chapitre 3.

Un module est une unité de décomposition logicielle. Les diverses formes de module, comme les routines ou les paquetages, ont été étudiées dans un chapitre précédent. Indépendamment du choix exact de la structure de module, nous pouvons considérer la notion de module comme un concept **syntaxique**, puisque la décomposition en modules n'affecte que la forme des textes logiciels, et non ce que peut faire le logiciel ; il est effectivement possible, en principe, d'écrire tout programme Ada comme un seul paquetage, ou tout programme Pascal comme un seul programme principal. Une telle approche n'est, bien sûr, pas recommandée, et tout développeur logiciel compétent utilisera les possibilités de module du langage pour décomposer son logiciel en morceaux plus maniables. Mais si nous prenons un programme existant, par exemple en Pascal, nous pouvons toujours fusionner tous les modules en un seul, et obtenir néanmoins un système qui fonctionne avec une sémantique équivalente. (La présence de routines récursives rend le processus de conversion moins trivial, mais ne modifie fondamentalement en rien cette discussion.) Ainsi la pratique de décomposition en modules est plus motivée par de solides principes de gestion de projet et d'ingénierie que par une nécessité intrinsèque.

Les types semblent, à première vue, un concept tout à fait différent. Un type est une description statique de certains objets dynamiques : les divers éléments de données qui seront traités durant l'exécution d'un système logiciel. L'ensemble des types contient habituellement des types prédéfinis comme `INTEGER` et `CHARACTER`, ainsi que des types définis par le développeur : type enregistrement (aussi appelé type structure), type pointeur, type ensemble (comme dans Pascal), type tableau et autres. La notion de type est un concept **sémantique** puisque chaque type influence directement l'exécution d'un système logiciel en définissant la forme des objets que le système créera et manipulera lors de l'exécution.

La classe comme module et type

Dans les approches non OO, les concepts de module et de type restent distincts. La propriété la plus remarquable de la notion de classe est qu'elle généralise ces deux concepts, les fusionnant en une seule construction linguistique. Une classe est un module, ou une unité de décomposition logicielle ; mais c'est aussi un type (ou, dans les cas faisant intervenir la généricité, un modèle de type).

L'essentiel de la puissance de la méthode orientée objet vient de cette identification. L'héritage, en particulier, ne peut être complètement appéhendé que si nous le considérons comme fournissant à la fois des extensions de module et des spécialisations de type.

Ce qui n'est pas encore clair, c'est de percevoir *comment* il est possible, en pratique, d'unifier deux concepts qui semblent, au début, si distants. La discussion et les exemples du reste de ce chapitre répondront à cette question.

7.4 UN SYSTÈME DE TYPES UNIFORME

La simplicité et l'uniformité du système de types sont des aspects importants de l'approche OO que nous développerons, conséquences d'une propriété fondamentale :

Règle de l'objet

Tout objet est instance d'une certaine classe.

La règle de l'objet s'appliquera non seulement aux objets composés définis par le développeur (comme les structures de données ayant plusieurs champs), mais aussi aux objets de base comme les entiers, les nombres réels, les valeurs booléennes et les caractères, qui seront tous considérés comme des instances de classes prédéfinies de bibliothèque (*INTEGER*, *REAL*, *DOUBLE*, *BOOLEAN*, *CHARACTER*).

L'acharnement à faire de toute valeur possible, aussi simple soit-elle, une instance d'une certaine classe peut paraître, de prime abord, exagéré ou même extravagant. Après tout, les mathématiciens et les ingénieurs ont utilisé les entiers et les réels avec succès depuis longtemps, sans savoir qu'ils manipulaient des instances de classe. Mais insister sur ce caractère uniforme est justifié pour plusieurs raisons :

- Il est toujours préférable d'avoir un cadre simple et uniforme plutôt que plusieurs cas spécifiques. Ici, le système de type sera entièrement basé sur la notion de classe.
- Décrire les types de base comme des ADT, et donc comme des classes, est simple et naturel. Il n'est pas difficile de voir comment définir, par exemple, la classe *INTEGER* avec des caractéristiques qui correspondent aux opérations arithmétiques comme "+", aux opérations de comparaison comme "<=" et à leurs propriétés associées, dérivées des axiomes mathématiques correspondants.

Les axiomes mathématiques définissant les entiers sont les axiomes de Peano.

- En définissant les types de base comme des classes, nous leur permettons de bénéficier de tous les atouts OO, en particulier l'héritage et la généricité. Si nous ne traitions pas les types de base comme des classes, nous devrions introduire de sérieuses limitations et de nombreux cas spéciaux.

  Comme exemple d'héritage, les classes *INTEGER*, *REAL* et *DOUBLE* seront les héritières de classes plus générales : *NUMERIC*, qui introduit les opérations arithmétiques de base comme "+", "-" et "*", et *COMPARABLE*, qui introduit les opérations de comparaison comme "<". Comme exemple de généricité, nous pouvons définir une classe générique *MATRIX* dont le paramètre générique représente le type des éléments de matrice de façon que les instances de *MATRIX* [*INTEGER*] représentent des matrices d'entiers, les instances de *MATRIX* [*REAL*] représentent des matrices de réels, et ainsi de suite. Comme exemple combinant la généricité et l'héritage, les définitions précédentes nous permettent également d'utiliser le type *MATRIX* [*NUMERIC*] , dont les instances représentent des matrices contenant des objets de type *INTEGER*, mais également des objets de type *REAL* et des objets de tout nouveau type *T*, défini par un développeur logiciel, qui hériterait de *NUMERIC*.

Avec une bonne implémentation, nous n'avons pas à craindre les conséquences négatives qui résulteraient de la décision consistant à définir tous les types comme des classes. Rien

n'empêche un compilateur d'avoir des connaissances spéciales à propos des classes de base ; le code qu'il génère pour des opérations sur des valeurs de type *INTEGER* ou *BOOLEAN* peut alors être tout aussi efficace que si ces types étaient des types prédéfinis dans le langage.

Obtenir un système de type complètement uniforme et cohérent impose d'utiliser plusieurs techniques OO importantes, que nous ne verrons que plus tard : les classes étendues, pour assurer une représentation adéquate des valeurs simples ; les opérateurs infixes ou préfixes pour autoriser la syntaxe arithmétique usuelle (comme `a < b` ou `-a` plutôt que des notations plus lourdes comme `a.less_than (b)` ou `a.negated`) ; la généricité contrainte, qui sert à définir des classes qui sont adaptées à des types ayant des opérations spécifiques, par exemple une classe *MATRIX* qui peut représenter des matrices d'entiers aussi bien que des matrices d'éléments ayant d'autres types numériques.

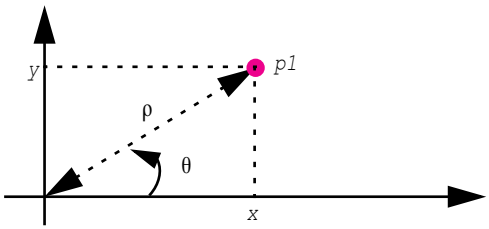
7.5 UNE CLASSE SIMPLE

Regardons maintenant à quoi ressemblent les classes en étudiant un exemple simple, mais typique, qui illustre certaines des propriétés fondamentales qui s'appliquent à presque toutes les classes.

Les caractéristiques

L'exemple choisi est celui de la notion de point, telle qu'elle pourrait apparaître dans un système graphique à deux dimensions.

Un point et ses coordonnées



Pour caractériser le type *POINT* comme un type abstrait de données, nous aurons besoin de quatre fonctions de requête *x*, *y*, ρ, θ. (Les noms des deux dernières seront prononcés *rho* et *theta* dans les textes logiciels.) La fonction *x* donne l'abscisse d'un point (coordonnée horizontale), *y* son ordonnée (coordonnée verticale), ρ sa distance à l'origine, θ l'angle avec l'axe horizontal. Les valeurs de *x* et *y* d'un point sont appelées ses coordonnées cartésiennes, celles de ρ et θ ses coordonnées polaires. Une autre fonction utile de requête est *distance*, qui renvoie la distance entre deux points.

Le nom `translate` fait référence à l'opération de "translation" de la géométrie.

La spécification d'ADT préciserait ensuite les commandes comme *translate* (pour déplacer un point d'une certaine distance horizontale et verticale), *rotate* (pour tourner le point d'un certain angle autour de l'origine) et *scale* (pour approcher ou éloigner le point de l'origine selon un certain facteur).

Il n'est pas difficile d'écrire la spécification complète d'ADT qui contient ces fonctions et certains des axiomes associés. Par exemple, deux des signatures de fonctions seront :

```
x: POINT → REAL
translate: POINT × REAL × REAL → POINT
```

et l'un des axiomes sera (pour tout point *p* et tous réels *a*, *b*) :

```
x (translate (p1, a, b)) = x (p1) + a
```

indiquant que translater un point de <*a*, *b*> incrémente son abscisse de *a*.

Vous pouvez, si vous le souhaitez, terminer cette spécification d'ADT vous-même. Le reste de cette étude supposera que vous avez compris l'ADT, que vous l'ayez ou non écrit formellement en entier, de façon que nous puissions nous focaliser sur son implémentation — la classe.

Exercice E7.2, page 218.

Attributs et routines

Un type abstrait de données comme *POINT* est caractérisé par un ensemble de fonctions, qui décrivent les opérations applicables aux instances de l'ADT. Dans les classes (les implémentations des ADT), les fonctions deviendront des caractéristiques — les opérations applicables aux instance de la classe.

Nous avons vu que les fonctions d'ADT sont de trois sortes : les requêtes, les commandes et les créateurs. Pour les caractéristiques, nous avons besoin d'une classification complémentaire, fondée sur la manière dont chacune d'elles est implémentée : par de l'espace mémoire ou du temps de calcul.

"Catégories de fonctions", page 139.

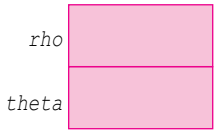
L'exemple des coordonnées des points illustre clairement la différence. Deux représentations courantes sont disponibles pour les points : cartésienne et polaire. Si nous choisissons la représentation cartésienne, chaque instance de la classe contiendra deux champs représentant le *x* et le *y* du point correspondant :

(*CARTESIAN_POINT*)

Représenter un point en coordonnées cartésiennes

Si *p1* est le point indiqué, obtenir son *x* ou son *y* ne nécessite qu'un accès au champ correspondant dans sa structure. Obtenir ρ ou θ, cependant, demande un calcul : pour ρ, nous devons calculer $\sqrt{x^2 + y^2}$ et, pour θ, nous devons calculer *arctg* (*y*/*x*) si *x* est non nul.

Si nous utilisons la représentation polaire, la situation est renversée : ρ et θ sont maintenant accessibles par simple accès à un champ, *x* et *y* demandant de petits calculs (de ρ *cos* θ et ρ *sin* θ).



(*POLAR_POINT*)

Représenter un point en coordonnées polaires

Cet exemple montre qu'on a besoin de deux sortes de caractéristiques :

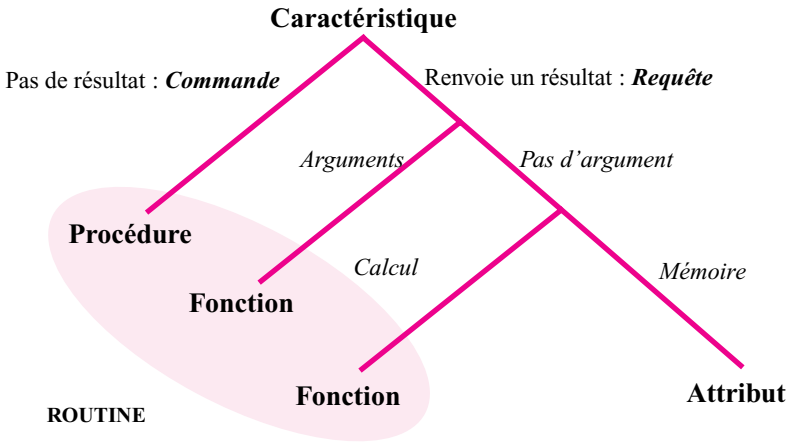
- Certaines caractéristiques seront représentées par de l'espace, c'est-à-dire en associant un élément d'information à chaque instance de la classe. Elles seront appelées **attributs**. Pour des points, `x` et `y` sont des attributs en représentation cartésienne ; `rho` et `theta` sont des attributs en représentation polaire.
- Certaines caractéristiques seront représentées par du temps, c'est-à-dire en définissant un certain calcul (un algorithme) applicable à toutes les instances de la classe. Elles seront appelées **routines**. Pour des points, `rho` et `theta` sont des routines en représentation cartésienne ; `x` et `y` sont des routines en représentation polaire.

Une distinction supplémentaire caractérise les routines (la seconde de ces catégories). Certaines routines renverront un résultat ; elles sont appelées **fonctions**. Ici, `x` et `y`, en représentation polaire, et `rho` et `theta`, en représentation cartésienne, sont des fonctions, puisqu'elles renvoient un résultat, de type `REAL`. Les routines qui ne renvoient pas de résultat correspondent aux commandes d'une spécification d'un ADT et sont appelées des **procédures**. Par exemple, la classe `POINT` contiendra les procédures `translate`, `rotate` et `scale`.

> Attention à ne pas confondre le mot "fonction" utilisé ici pour désigner des routines renvoyant un résultat dans des classes avec celui précédemment utilisé pour désigner les spécifications mathématiques des opérations dans les types abstraits de données. Ce conflit est fâcheux, mais découle de l'usage bien établi de ce mot à la fois en mathématiques et en informatique.

L'arbre suivant aide à visualiser cette classification des caractéristiques :

Classification des caractéristiques par rôle

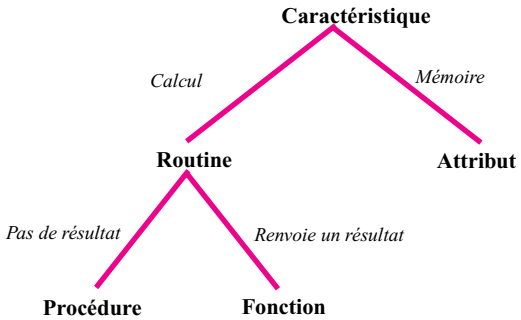


Il s'agit d'une classification externe, dans laquelle la question principale est de savoir comment une caractéristique apparaîtra à ses clients (ses utilisateurs).

Nous pouvons aussi adopter une vue plus interne, en utilisant comme critère principal la manière dont chaque caractéristique est implémentée dans la classe, ce qui conduit à une classification différente (voir figure suivante).

Accès uniforme

De prime abord, quelque chose peut paraître déroutant dans les classifications précédentes, et cela a peut-être retenu votre attention. Dans de nombreux cas, nous devrions être capable de



Classification des caractéristiques par implémentation

manipuler des objets, par exemple un point `p1`, sans nous préoccuper de savoir si la représentation interne de `p1` est cartésienne, polaire ou autre. Est-il donc approprié de faire une distinction explicite entre attribut et fonction ?

La réponse dépend du point de vue selon lequel on se place : la vue du fournisseur (comme le verrait l'auteur de la classe elle-même, ici `POINT`) ou la vue du client (comme le verrait l'auteur d'une classe qui utilise `POINT`). Pour le fournisseur, la distinction entre attribut et fonction est nécessaire et sensée puisque, dans certains cas, vous voudrez implémenter une caractéristique par de la mémoire et, dans d'autres, par du calcul, et la décision doit être indiquée quelque part. Mais forcer les **clients** à être conscients de cette différence serait fâcheux. Si je suis en train d'accéder à `p1`, je veux pouvoir trouver son `x` ou son ρ sans avoir à connaître la manière dont ces requêtes sont implémentées.

Le principe d'accès uniforme, introduit lors de l'étude de la modularité, répond à cette inquiétude. Ce principe indique qu'un client devrait être capable d'accéder à une propriété d'un objet en utilisant une notation unique, que la propriété soit implémentée par de la mémoire ou par du calcul (espace ou temps, attribut ou routine). Nous suivrons cet important principe en concevant ci-dessous une notation pour l'appel de caractéristique : l'expression qui désigne la valeur de la caractéristique `x` pour `p1` sera toujours

Voir "Accès uniforme", page 57.

```
p1.x
```

que ce soit pour accéder à un champ d'un objet ou pour exécuter une routine.

> Comme vous l'aurez noté, l'incertitude n'existe que pour les requêtes sans argument, qui peuvent être implémentées comme des fonctions ou comme des attributs. Une commande doit être une procédure ; une requête avec des arguments doit être une fonction, puisque les attributs ne peuvent pas avoir d'argument.

Le principe d'accès uniforme est essentiel pour garantir l'autonomie des composants d'un système. Il préserve la liberté qu'a le concepteur de classes d'expérimenter diverses techniques d'implémentation sans perturber les clients.

> Pascal, C et Ada violent ce principe en fournissant une notation différente pour un appel de fonction et pour un accès d'attribut. Dans de tels langages non orientés objet, ceci est compréhensible (bien que nous ayons vu qu'Algol W, un prédécesseur de Pascal, proposait l'accès uniforme dès 1966). Des langages plus récents comme C++ et Java n'imposent pas non plus ce principe. Quand l'accès uniforme est violé, tout changement de représentation interne (comme passer du polaire au cartésien ou à une autre représentation) peut causer des dégâts dans de nombreuses classes. Cela constitue une source majeure d'instabilité lors du développement logiciel.

"Documentation et assertions : la forme abrégée d'une classe", page 378.

Le principe d'accès uniforme impose également une contrainte sur les techniques de documentation. Si nous voulons appliquer le principe de manière cohérente, nous devons nous assurer qu'il n'est pas possible de déterminer, à partir de la documentation officielle d'une classe, si une requête sans argument est une fonction ou un attribut. Cela constituera une des propriétés du mécanisme standard de documentation d'une classe, connu sous le nom de formulaire abrégé.

La classe

Voici une version du texte de la classe *POINT*. (Toute occurrence de tirets consécutifs introduit un commentaire, qui va jusqu'au bout de la ligne ; les commentaires sont des explications pour le lecteur du texte de la classe et ne modifient pas la sémantique de la classe.) :

```
indexing
        description: "Points à deux dimensions"
class POINT feature
        x, y: REAL
                        -- Abscisse et ordonnée

        rho: REAL is
                        -- Distance à l'origine (0, 0)
                do
                        Result := sqrt (x ^ 2 + y ^ 2)
                end

        theta: REAL is
                        -- Angle avec l'axe horizontal
                do
                        ...Laissé au lecteur (exercice E7.3, page 218)
                end
        distance (p: POINT): REAL is
                        -- Distance à p
                do
                        Result := sqrt ((x - p.x) ^ 2 + (y - p.y) ^ 2)
                end

        translate (a, b: REAL) is
                        -- Déplacer de a horizontalement, b verticalement.
                do
                        x := x + a
                        y := y + b
                end

        scale (factor: REAL) is
                        -- Augmenter d'un facteur factor.
                do
                        x := factor * x
                        y := factor * y
                end

        rotate (p: POINT; angle: REAL) is
                        -- Rotation autour de p de angle.
                do
                        ...Laissé au lecteur (exercice E7.3, page 218) ...
                end
end
```

Les prochaines sections expliquent en détail les aspects les moins évidents du texte de cette classe.

Une classe consiste essentiellement en une clause qui recense les diverses caractéristiques et qui est introduite par le mot-clé **feature**. Il y a aussi une clause **indexing** qui donne une information générale *description*, utile au lecteur de la classe, mais qui ne joue aucun rôle lors de l'exécution. Par la suite, nous étudierons trois clauses optionnelles : **inherit** pour l'héritage, **creation** pour la création non standard et **invariant** pour l'introduction des invariants de classe ; nous verrons aussi comment inclure plusieurs options **feature** dans une classe.

7.6 CONVENTIONS DE BASE

La classe *POINT* illustre certaines des techniques qui seront utilisées dans les exemples qui vont suivre. Évoquons d'abord les conventions de base.

Reconnaître les sortes de caractéristiques

Les caractéristiques *x* et *y* sont simplement déclarées comme étant de type *REAL*, sans précision d'algorithme associé ; elles ne peuvent donc être que des attributs. Toutes les autres caractéristiques ont une clause de la forme :

```
is
        do
                ... Instructions...
        end
```

qui définit un algorithme ; cela indique que la caractéristique est une routine. Les routines *rho*, *theta* et *distance* sont déclarées comme renvoyant un résultat, de type *REAL* dans tous les cas, comme l'indiquent les déclarations de la forme :

```
        rho: REAL is ...
```

Cela les définit comme des fonctions. Les deux autres routines, *translate* et *scale*, ne renvoient pas de résultat (puisqu'elles n'ont pas de déclaration de résultat de la forme *:T* pour un type *T* donné), et sont donc des procédures.

Puisque *x* et *y* sont des attributs, alors que *rho* et *theta* sont des fonctions, la représentation choisie pour cette classe particulière de points est cartésienne.

Les corps de routines et les commentaires d'en-tête

Le corps d'une routine (la clause **do**) est une séquence d'instructions. Dans la tradition Algol-Pascal, vous pouvez utiliser des points-virgules pour séparer les instructions et les déclarations successives, mais ils sont optionnels. Nous les omettrons, pour des raisons de simplicité, quand les éléments sont situés sur des lignes séparées, mais nous les inclurons toujours pour délimiter des instructions ou des déclarations qui apparaissent sur la même ligne.

Pour plus de détails, voir "La guerre des points-virgules", page 866.

Toutes les instructions des routines de la classe *POINT* sont des affectations ; pour l'affectation, la notation utilise le symbole := (emprunté également aux conventions Algol-Pascal). Ce symbole ne devrait, bien sûr, pas être confondu avec le symbole d'égalité =, utilisé, comme en mathématiques, comme opérateur de comparaison.

Une autre convention de la notation est l'utilisation de commentaires d'en-tête. Comme nous l'avons déjà noté, les commentaires sont introduits par deux tirets consécutifs --. Ils peuvent apparaître n'importe où dans le texte d'une classe, quand l'auteur de la classe juge utile de donner une explication aux lecteurs. Un rôle spécial est joué par le **commentaire d'en-tête**, qui, suivant en cela une règle générale de style, devrait apparaître après le mot-clé **is** au début de chaque routine, indenté comme l'illustrent les exemples de la classe *POINT*. Un tel commentaire d'en-tête devrait brièvement indiquer la raison d'être de la routine.

Les attributs devraient aussi avoir un commentaire d'en-tête après leur déclaration, aligné avec les commentaires d'en-tête de la routine, comme illustré ici avec *x* et *y*.

La clause indexing

Voir "Une remarque concernant l'indexation des composants", page 81.

Au début d'une classe se trouve une clause débutant par le mot-clé **indexing**. Elle contient une seule entrée, étiquetée *description*. La clause d'indexation n'a pas d'effet sur l'exécution du logiciel, mais sert à associer de l'information à une classe. Dans sa forme générale, elle peut contenir une entrée, ou plus, de la forme

```
index_word: index_value, index_value, ...
```

où *index_word* est un identificateur arbitraire et où chaque *index_value* est un élément arbitraire du langage (identificateur, entier, chaîne).

Le bénéfice est double :

- Les lecteurs de la classe ont accès à un résumé de ses propriétés, sans avoir à regarder les détails.
- Dans un environnement de développement logiciel qui permet la réutilisation, les outils de recherche (souvent appelés *navigateurs*) peuvent utiliser l'information d'indexation pour aider les utilisateurs potentiels à découvrir les classes disponibles ; les outils permettent aux utilisateurs d'entrer divers mots de recherche et de les apparier avec les mots d'index et avec leur valeur.

Le chapitre 36 décrit un mécanisme général de navigation OO .

L'exemple introduit une unique entrée d'index, utilisant *description* comme mot d'index et, comme valeur d'index, une chaîne décrivant l'objectif de la classe. Toutes les classes de ce livre, sauf certains courts exemples, incluront une entrée **indexing**. Vous êtes fortement encouragé à suivre cet exemple et à débuter tout texte de classe par une clause qui fournit une description concise de la classe, tout comme chaque routine débute par un commentaire d'en-tête.

"Autodocumentation", page 56.

Les clauses d'indexation et les commentaires d'en-tête sont tous les deux des applications directes du principe d'autodocumentation : autant que faire se peut, la documentation d'un module devrait apparaître dans le texte du module lui-même.

Désigner le résultat d'une fonction

Une "entité" est un nom désignant une valeur. Définition complète page 216.

Nous avons besoin d'une autre convention pour comprendre le texte des fonctions de la classe *POINT* : *rho*, *theta* et *distance*.

Tout langage qui utilise des fonctions (des routines qui renvoient des valeurs) doit offrir une notation permettant, dans le corps de la fonction, de définir la valeur qui sera renvoyée par un appel donné. La convention utilisée ici est simple : elle repose sur un identificateur prédéfini,

Result, qui désigne la valeur que l'appel renverra. Par exemple, le corps de *rho* contient une affectation à *Result* :

```
Result := sqrt (x ^ 2 + y ^ 2)
```

Result est un mot réservé et ne peut apparaître que dans les fonctions. Dans une fonction dont le résultat est déclaré de type *T*, *Result* est traité de la même manière que les autres entités et peut recevoir des valeurs via les instructions d'affectation comme ci-dessus.

Tout appel de fonction renverra, comme résultat, la valeur finale affectée à *Result* durant l'exécution de l'appel. Cette valeur existe toujours puisque les règles du langage (que nous verrons en détail plus tard) imposent que toute exécution de la routine, quand elle démarre, initialise *Result* à une valeur prédéfinie. Pour un *REAL*, la valeur d'initialisation est 0 ; ainsi une fonction de la forme :

Les règles d'initialisation seront données dans "L'instruction de création", page 233.

```
non_negative_value (x: REAL): REAL is
            -- La valeur de x si elle est positive ; zéro sinon.
      do
            if x > 0.0 then
                  Result := x
            end
      end
```

renverra toujours une valeur bien définie (décrite par le commentaire d'en-tête) bien que l'instruction conditionnelle n'ait pas de branche **else**.

La section de discussion à la fin de ce chapitre examine les motivations amenant à la convention *Result* et compare celle-ci aux autres techniques, comme les instructions de retour. Bien que cette convention concerne un problème qui se pose dans tous les langages de conception et de programmation, elle s'accorde particulièrement bien avec le reste de l'approche orientée objet.

Voir "Désigner le résultat d'une fonction", page 213.

Règles de style

Les textes des classes de ce livre suivent des conventions de style précises concernant l'indentation, les fontes (pour les impressions), le choix des noms des caractéristiques et des classes, l'utilisation des minuscules et majuscules.

L'étude mettra en avant, au fur et à mesure, ces conventions, sous l'en-tête "règles de style". Elles ne devraient pas être considérées comme simplement cosmétiques : un logiciel de qualité demande cohérence et précision, de forme comme de contenu. L'objectif de réutilisabilité rend ces observations encore plus importantes, puisqu'il sous-entend que les textes logiciels auront une longue vie, durant laquelle nombre de personnes devront les comprendre et les améliorer.

Vous devriez appliquer les règles de style dès que vous commencez à écrire une classe. Par exemple, vous ne devriez jamais écrire une routine sans inclure immédiatement son commentaire d'en-tête. Cela ne prend pas beaucoup de temps, et ce n'est pas du temps perdu ; en fait, c'est du temps que vous aurez gagné quand il s'agira, pour vous ou d'autres, de modifier cette classe, que ce soit dans la demi-heure qui suit ou dans dix ans. En utilisant une indentation régulière, une orthographe adéquate pour les commentaires et les identificateurs, des conventions lexicales appropriées — un espace avant chaque parenthèse ouvrante, mais pas après, et ainsi de suite —, votre tâche ne sera guère plus longue que si vous aviez ignoré ces règles mais, cumulée sur des mois de travail et des tonnes de logiciel, cela fera une différence majeure. Le soin apporté à de

tels détails est une condition nécessaire, quoique non suffisante, pour obtenir du logiciel de qualité (et la qualité, le thème général de ce livre, est ce qui définit le génie logiciel).

Le chapitre 26 est consacré aux règles de style.

Les règles élémentaires de style sont maintenant clairement établies au travers de l'exemple de classe précédent. Cependant, comme notre but immédiat est d'explorer les mécanismes de base de la technologie objet, nous ne verrons leur description précise que dans un prochain chapitre.

Hériter les services généraux

Un autre aspect de la classe *POINT* mérite une explication : la présence d'appels à la fonction *sqrt* (dans *rho* et *distance*). Cette fonction devrait clairement renvoyer la racine carrée d'un nombre réel, mais d'où vient-elle ?

Puisqu'il ne semble pas souhaitable de polluer un langage général avec des opérations arithmétiques spécialisées, la meilleure technique consiste à définir de telles opérations comme des caractéristiques d'une classe spécialisée — disons *ARITHMETIC* — et de demander simplement que toute classe qui a besoin de ces services hérite de cette classe spécialisée. Comme nous le verrons en détail dans un prochain chapitre, il suffit alors d'écrire *POINT* sous la forme :

```
class POINT inherit
        ARITHMETIC
feature
        ... Le reste comme auparavant ...
end
```

Voir "HÉRITAGE DE SERVICE", 24.9, page 817.

Cette technique permettant d'hériter des services généraux est quelque peu controversée ; certains argumenteront que, selon les principes orientés objet, une fonction telle que *sqrt* doit être considérée comme une caractéristique de la classe représentant l'objet auquel elle s'applique, par exemple *REAL*. Mais les opérations sur les nombres réels sont nombreuses, et toutes ne peuvent pas être incluses dans la classe. La racine carrée peut sembler suffisamment fondamentale pour être considérée comme une caractéristique de la classe *REAL* ; nous écririons alors *a*.*sqrt* plutôt que *sqrt* (*x*). Nous reviendrons, lors de l'examen des principes de conception, sur l'opportunité d'introduire des classes de "services" comme *ARITHMETIC*.

7.7 LE STYLE ORIENTÉ OBJET DE CALCUL

Concentrons-nous maintenant sur les propriétés fondamentales de la classe *POINT* en essayant de comprendre le corps d'une routine typique avec ses instructions, puis en étudiant comment la classe et ses caractéristiques peuvent être utilisées par d'autres classes— des clients.

L'instance courante

Voici à nouveau le texte de l'une des routines de notre exemple ; la procédure *translate* :

```
translate (a, b: REAL) is
                -- Déplacer de a horizontalement, b verticalement.
        do
                x := x + a
                y := y + b
        end
```

À première vue, ce texte apparaît suffisamment clair : pour décaler un point de *a* horizontalement et de *b* verticalement, nous ajoutons *a* à son *x* et *b* à son *y*. Mais, en regardant plus attentivement, cela ne paraît tout à coup plus si évident ! Nulle part dans le texte nous n'avons précisé de quel point il s'agissait. À qui appartiennent ces *x* et ces *y* auxquels nous ajoutons *a* et *b* ? C'est dans la réponse à cette question que réside l'un des aspects les plus caractéristiques du style de développement orienté objet. Avant d'être prêts à découvrir cette réponse, nous devons comprendre un certain nombre de sujets intermédiaires.

Le texte d'une classe décrit les propriétés et le comportement des objets d'un certain type, des points dans cet exemple. Il le fait en décrivant les propriétés et le comportement d'une instance typique de ce type, ce que nous pourrions appeler le "point de la rue" de la même manière que les journaux évoquent l'opinion de "l'homme ou de la femme de la rue". Nous nous limiterons à un nom plus formel : l'**instance courante** de la classe.

De temps en temps, nous aurons besoin de faire référence à l'instance courante de manière explicite. Le mot réservé :

```
Current
```

servira à cet effet. Dans le texte d'une classe, *Current* désigne l'instance courante de la classe englobante. Pour un exemple illustrant le cas où *Current* est nécessaire, supposez que nous réécrivions *distance* de façon qu'elle vérifie d'abord si l'argument *p* est le même point que l'instance courante, auquel cas le résultat est 0, et ce, sans autre calcul. *distance* apparaîtra alors sous la forme :

```
distance (p: POINT): REAL is
        -- Distance à p
do
        if p /= Current then
                Result := sqrt ((x − p.x) ^ 2 + (y − p.y) ^ 2)
        end
end
```

(/= est l'opérateur d'inégalité. Du fait de la règle d'initialisation, mentionnée plus haut, l'instruction conditionnelle n'a pas besoin d'une branche **else** : si *p = Current*, le résultat est zéro.)

Dans la plupart des cas, cependant, l'instance courante est implicite et nous n'aurons pas à faire référence à *Current* par son nom. Par exemple, les références à *x* dans le corps de *translate* et dans les autres routines correspondent simplement au "*x* de l'instance courante", si elles ne sont pas autrement précisées.

Cela ne fait que repousser le mystère, bien sûr : qui est donc réellement *Current* ? La réponse viendra lors de l'étude des appels de routines ci-dessous. Tant que nous ne regardons que le texte de la routine, il suffit de savoir que toutes les opérations sont relatives, par défaut, à un objet défini implicitement, l'instance courante.

Clients et fournisseurs

Si nous ignorons pour quelques instants l'énigme de l'identité de *Current*, nous savons comment définir des classes simples. Nous devons maintenant étudier comment utiliser leurs

définitions. De telles utilisations apparaîtront dans d'autres classes — puisque, dans une approche purement orientée objet, tout élément logiciel fait partie du texte d'une classe.

Les chapitres 14 à 16 étudient l'héritage.

Il n'y a que deux manières d'utiliser une classe comme *POINT*. L'une consiste à hériter d'elle, ce que nous étudierons dans les prochains chapitres. L'autre consiste à devenir un **client** de *POINT*.

La manière la plus simple et la plus courante de devenir un client d'une classe est de déclarer une entité ayant le type correspondant :

> ***Définition : client, fournisseur***
> Soit *S*, une classe. Une classe *C* qui contient une déclaration de la forme `a: S` est un client de *S*. *S* est alors un fournisseur de *C*.

Dans cette définition, *a* peut être un attribut ou une fonction de *C*, ou une entité locale ou un argument d'une routine de *C*.

Par exemple, les déclarations de *x*, *y*, *rho*, *theta* et *distance* ci-dessus font que la classe *POINT* est un client de *REAL*. D'autres classes peuvent alors devenir des clients de *POINT*. En voici un exemple :

```
class GRAPHICS feature
        p1: POINT

        ...

        some_routine is
                        -- Effectuer des actions sur p1.
                do
                        ... Créer une instance de POINT, attachée à p1 ...
                        p1.translate (4.0, -1.5)                          --**
                        ...
                end
        ...
end
```

Avant que l'instruction marquée `--**` soit exécutée, l'attribut *p1* aura une valeur qui désignera une certaine instance de la classe *POINT*. Supposez que cette instance représente l'origine, de coordonnées *x* = 0, *y* = 0 :

L'origine

L'entité *p1* est dite **attachée** à cet objet. Pour l'instant, peu nous importe comment l'objet a été créé (par la ligne sibylline "... `Créer une instance`...") et initialisé ; ces aspects seront abordés lors de l'étude du modèle objet dans le prochain chapitre. Supposons simplement que l'objet existe et que *p1* lui est attaché.

Appel de caractéristique

L'instruction étoilée,

```
p1.translate (4.0, -1.5)
```

mérite un examen attentif, car c'est notre premier exemple complet de ce qui peut être appelé le **mécanisme de base du calcul orienté objet** : l'appel de caractéristique. Durant l'exécution d'un système logiciel orienté objet, tout calcul est effectué en appelant certaines caractéristiques sur certains objets.

Cet appel de caractéristique particulier veut dire : appliquer à `p1` la caractéristique `translate` de la classe `POINT`, avec les arguments `4.0` et `-1.5`, correspondant à `a` et `b` dans la déclaration de `translate` donnée dans la classe. Plus généralement, un appel de caractéristiques apparaît dans sa version de base sous l'une des deux formes :

```
x.f
x.f (u, v, ...)
```

Dans un tel appel, `x`, appelé la **cible** de l'appel, est une entité ou une expression (qui sera attachée, lors de l'exécution, à un certain objet). Comme toute autre entité ou expression, `x` a un certain type, donné par une classe `C` ; `f` doit être alors une des caractéristiques de `C`. Plus précisément, dans la première forme, `f` doit être un attribut ou une routine sans arguments ; dans la seconde forme, `f` doit être une routine avec des arguments et `u`, `v`, ..., appelés les **arguments réels** de l'appel, doivent être des expressions ayant le même type et en nombre égal aux arguments formels déclarés pour `f` dans `C`.

De plus, `f` doit être accessible (exporté) au client qui contient cet appel. C'est le cas par défaut ; une section ultérieure montrera comment restreindre les droits d'exportation. Pour le moment, toutes les caractéristiques sont accessibles à tous les clients.

"EXPORTATIONS SÉLECTIVES ET RÉTENTION D'INFORMATION", 7.8, page 194.

L'effet de l'appel ci-dessus, quand il est exécuté, est défini comme suit :

Effet de l'appel d'une caractéristique `f` sur une cible `x`

Appliquer la caractéristique `f` à l'objet attaché à `x`, après avoir initialisé chaque argument formel de `f` (s'il y en a) avec la valeur de l'argument réel correspondant.

Le principe de cible unique

Quelle est la particularité d'un appel de caractéristique ? Après tout, chaque développeur logiciel sait comment écrire une procédure `translate` qui déplace un point d'une certaine distance et qu'on peut appeler, dans la forme traditionnelle (disponible, avec des variations mineures, dans tous les langages de programmation) :

```
translate (p1, 4.0, -1.5)
```

Contrairement au style orienté objet de l'appel de caractéristique, cependant, cet appel considère tous les arguments comme égaux. La forme OO n'est pas aussi symétrique : nous choisissons un certain objet (ici le point `p1`), reléguant les autres arguments, ici les nombres

réels `4.0` et `-1.5`, aux rôles de figurants. Cette manière de considérer chaque appel comme relatif à un objet cible unique caractérise l'essentiel du style du calcul orienté objet :

Principe de cible unique

Chaque opération de calcul orienté objet est relative à un certain objet.

Pour les novices, il s'agit souvent de l'aspect le plus déconcertant de la méthode. Dans la construction de logiciels orientés objet, nous ne demandons jamais vraiment : "Appliquer cette opération à ces objets". À la place, nous disons : "Appliquer cette opération à **cet** objet-là". Et, peut-être (dans la seconde forme) : "Oh, d'ailleurs, j'ai failli oublié de vous dire : vous aurez besoin de ces valeurs-là comme arguments".

Ce que nous avons vu jusqu'ici ne suffit pas vraiment à justifier cette convention ; en fait, ses inconvénients masqueront, pendant un certain temps, ses avantages. Un exemple d'effet contre-intuitif apparaît avec la fonction `distance` de la classe `POINT`, déclarée ci-dessus comme `distance (p: POINT): REAL`, impliquant qu'un appel typique sera écrit :

```
p1.distance (p2)
```

ce qui va à l'encontre de la perception d'une `distance` comme étant une opération symétrique pour les deux arguments. Le principe de cible unique ne sera complètement justifié qu'avec l'introduction de l'héritage.

L'identification module-type

"La classe comme module et type", page 174.

Le principe de cible unique est une conséquence directe de la fusion module-type, présentée précédemment comme le point de départ de la décomposition orientée objet : si chaque module est un type, chaque opération dans le module est alors relative à une certaine instance de ce type (l'instance courante). Jusqu'à présent, cependant, les détails de cette fusion sont restés un peu mystérieux. Une classe, avons-nous dit ci-dessus, est à la fois un module et un type ; mais comment pouvons-nous réconcilier la notion syntaxique de module (un regroupement de services voisins, formant une partie d'un système logiciel) avec la notion sémantique de type (la description statique de certains objets possibles à l'exécution) ? L'exemple de `POINT` rend la réponse limpide :

Comment marche la fusion module-type

Les services fournis par la classe `POINT`, considérée comme un module, sont précisément les opérations applicables aux instances de la classe `POINT`, considérée comme un type.

Cette identification entre les opérations sur les instances d'un type et les services fournis par un module réside au coeur de la discipline de structuration imposée par la méthode orientée objet.

Le rôle de *Current*

Avec l'aide du même exemple, nous sommes maintenant également à même de résoudre le mystère qui subsiste : qu'est ce que représente réellement l'instance courante ?

La forme des appels indique pourquoi le texte d'une routine (comme *translate* dans *POINT*) n'a pas besoin de spécifier "qui" est *Current* : puisque chaque appel à une routine sera relatif à une certaine cible, spécifiée explicitement dans l'appel, l'exécution traitera chaque nom de caractéristique qui apparaît dans le texte d'une routine (par exemple *x* dans le texte de *translate*) comme s'appliquant à cette cible particulière. Ainsi, pour l'exécution de l'appel :

```
p1.translate (4.0, -1.5)
```

chaque occurrence de *x* dans le corps de *translate*, comme celle de l'instruction :

```
x := x + a
```

veut dire : "le *x* de *p1*".

Le sens exact de *Current* découle de ces observations. *Current* veut dire : "la cible de l'appel courant". Par exemple, pendant la durée de l'appel ci-dessus, *Current* désignera l'objet attaché à *p1*. Dans un appel suivant, *Current* désignera la cible de ce nouvel appel. Tout ceci a un sens grâce à l'extrême simplicité du modèle de calcul orienté objet, fondé sur des appels de caractéristiques et sur le principe de cible unique :

Principe de l'appel de caractéristique

F1 • Aucun élément logiciel n'est jamais exécuté s'il ne fait pas partie d'un appel de routine.

F2 • Tout appel a une cible.

Appels qualifiés et non qualifiés

Il a été indiqué ci-dessus que tout calcul orienté objet repose sur des appels de caractéristiques. Une conséquence de cette règle est que les textes logiciels contiennent de fait plus d'appels qu'il y paraît à première vue. Les appels vus jusqu'à présent étaient de l'une des deux formes introduites ci-dessus :

```
x.f
x.f (u, v, ...)
```

De tels appels utilisent la notation dite pointée (avec le symbole ".") et sont dits **qualifiés**, car la cible de l'appel est explicitement identifiée : il s'agit de l'entité ou de l'expression (*x* dans les deux cas ci-dessus) qui apparaît avant le point.

D'autres appels, cependant, seront non qualifiés, car leurs cibles sont implicites. Par exemple, supposons que nous voulions ajouter à la classe *POINT* une procédure *transform* qui déplacera et élargira un point. Le texte de la procédure peut reposer sur *translate* et *scale* :

```
transform (a, b, factor: REAL) is
            -- Déplacer de a horizontalement, b verticalement,
            -- puis élargir d'un factor.
      do
            translate (a, b)
            scale (factor)
      end
```

Le corps de la routine contient des appels à *translate* et *scale*. Contrairement aux exemples précédents, ces appels ne montrent pas de cibles explicites et n'utilisent pas la notation pointée. De tels appels sont dits **non qualifiés**.

Les appels non qualifiés ne violent pas la propriété appelée F2 dans le principe de l'appel de caractéristique : comme les appels qualifiés, ils ont une cible. Comme vous l'avez certainement deviné, la cible, dans ce cas, est l'instance courante. Quand la procédure *transform* est appelée sur une certaine cible, son corps appelle *translate* et *scale* sur la même cible. Il aurait, en fait, pu être écrit :

```
do
        Current.translate (a, b)
        Current.scale (factor)
```

De manière stricte, l'équivalence ne s'applique que si la caractéristique est exportée.

Plus généralement, vous pouvez réécrire tout appel non qualifié comme un appel qualifié ayant *Current* pour cible. La forme non qualifiée est, bien sûr, plus simple et tout aussi claire.

Les appels non qualifiés que nous venons d'examiner sont des appels de routines. Le même raisonnement s'applique aux attributs, bien que la présence des appels soit peut-être moins évidente dans ce cas. Nous avons noté ci-dessus que, dans le corps de *translate*, l'occurrence de *x* dans l'expression *x + a* désigne le champ *x* de l'instance courante. Une autre manière d'exprimer cette propriété est de dire que *x* est en fait un appel de caractéristique, et que l'expression globale aurait pu être écrite *Current.x + a*.

Plus généralement, toute instruction ou expression de l'une des formes :

```
f
f (u, v, ...)
```

est en fait un appel non qualifié, et vous pouvez aussi l'écrire sous une forme qualifiée (respectivement) :

```
Current.f
Current.f (u, v, ...)
```

bien que les formes non qualifiées soient plus pratiques. Si vous utilisez une telle notation dans une instruction, *f* doit être une procédure (sans arguments dans la première forme, et avec le bon nombre d'arguments, de types corrects, dans la seconde). S'il s'agit d'une expression, *f* peut être un attribut (dans la première forme seulement, puisque les attributs n'ont pas d'arguments) ou une fonction.

Notez bien que cette équivalence syntaxique ne s'applique qu'à une caractéristique utilisée comme instruction ou expression. Ainsi, dans l'affectation suivante de la procédure *translate* :

```
x := x + a
```

seule l'occurrence de *x* sur le côté droit est un appel non qualifié : *a* est un argument formel, pas une caractéristique ; et l'occurrence de *x* sur la gauche n'est pas une expression (on ne peut pas affecter de valeur à une expression), et cela n'aurait aucun sens de la remplacer par *Current.x*.

Caractéristiques d'opérateurs

Une observation plus poussée de l'expression *x + a* conduit à une notion utile : les caractéristiques d'opérateurs. Cette notion (et cette section) peut être considérée comme purement cosmétique, c'est-à-dire, ne concernant qu'une facilité syntaxique qui n'apporte rien de réellement neuf à la méthode orienté objet. Mais de telles propriétés syntaxiques peuvent rendre la vie des développeurs plus agréable, si elles sont présentes, ou plus pénible, si elles sont absentes. Les caractéristiques d'opérateurs fournissent aussi un bon exemple de la

manière avec laquelle le paradigme orienté objet peut profitablement intégrer certaines approches antérieures, et cela en douceur.

Voici l'idée. Bien que vous puissiez ne pas l'avoir deviné, l'expression `x + a` ne contient pas seulement un appel — l'appel à `x` — que nous venons de voir, mais deux. Dans un calcul non OO, nous considérerions `+` comme un opérateur, appliqué ici à deux valeurs `x` et `a`, toutes deux déclarées de type `REAL`. Dans un modèle purement OO, comme on l'a vu, le seul mécanisme de calcul est l'appel de caractéristique ; ainsi, vous pouvez considérer l'addition elle-même, au moins en théorie, comme étant un appel à une caractéristique d'addition.

Pour mieux comprendre cela, considérez comment nous pourrions définir le type `REAL`. La règle de l'objet exprimée plus tôt imposait que chaque type soit basé sur une classe. Cela s'applique aux types prédéfinis comme `REAL` ainsi qu'aux types définis par le développeur comme `POINT`. Supposez qu'on vous demande d'écrire `REAL` comme une classe. Il n'est pas difficile d'en identifier les caractéristiques pertinentes : les opérations arithmétiques (addition, soustraction, négation…), les opérations de comparaison (plus petit que, plus grand que…). Ainsi une première ébauche pourrait être :

La régle de l'objet a été donnée en page 175.

```
indexing
        description: "Nombres réels (pas la version finale !)"
class REAL feature
        plus (other: REAL): REAL is
                        -- Somme de la valeur courante et d'other
                do
                        ...
                end
        minus (other: REAL) REAL is
                        -- Différence entre la valeur courante et other
                do
                        ...
                end
        negated: REAL is
                        -- Valeur courante, mais de signe opposé
                do
                        ...
                end
        less_than (other: REAL): BOOLEAN is
                        -- Est-ce que la valeur courante est strictement inférieure à other ?
                do
                        ...
                end
        ... Autres caractéristiques ...
end
```

Avec une telle forme de classe, vous ne pourriez plus écrire une expression arithmétique comme `x + a` ; à la place, vous pourriez utiliser un appel de la forme :

```
x.plus (a)
```

De même, vous devriez écrire `x.negated` à la place de l'habituel `-x`.

On pourrait essayer de justifier un tel écart avec la notation mathématique usuelle pour rester cohérent avec le modèle orienté objet, et invoquer l'exemple de Lisp pour montrer qu'il est parfois possible de convaincre une partie de la communauté du développement logiciel de renoncer à une notation standard. Mais cet argument a ses propres limites : l'usage de Lisp a

toujours été marginal. Il est plutôt dangereux de s'opposer aux notations qui existent depuis des siècles et que les gens utilisent depuis l'école primaire, en particulier quand il n'y a rien à reprocher aux dites notations.

Un simple système syntaxique réconcilie le souhait de cohérence (demandant ici un mécanisme de calcul simple fondé sur l'appel de caractéristique) et le besoin de compatibilité avec les notations traditionnelles. Il suffit de considérer qu'une expression de la forme.

```
x + a
```

est, en fait, un appel à la caractéristique d'addition de la classe *REAL* ; la seule différence avec la caractéristique *plus* suggérée ci-dessus est que nous devons réécrire la déclaration de la caractéristique correspondante pour spécifier que les appels utiliseront la notation d'opérateurs plutôt que la notation pointée.

Voici la forme d'une classe qui répond à cet objectif :

Le prochain chapitre montrera comment déclarer cette classe comme "étendue". Voir "Le rôle des types expansés", page 254.

```
indexing
        description: "Nombres réels"
class REAL feature
        infix "+" (other: REAL): REAL is
                        -- Somme de la valeur courante et de other
                do
                        ...
                end
        infix "-" (other: REAL) REAL is
                        -- Différence entre la valeur courante et other
                do
                        ...
                end
        prefix "-": REAL is
                        -- Valeur courante, mais de signe opposé
                do
                        ...
                end
        infix "<" (other: REAL): BOOLEAN is
                        -- Est-ce que la valeur courante est strictement inférieure à other ?
                do
                        ...
                end
        ... Autres caractéristiques...
end
```

Deux nouveaux mots-clés ont été introduits : **infix** et **prefix**. Une simple extension syntaxique nous permet aussi, dorénavant, de choisir des noms de caractéristiques qui, à la place d'identificateurs (comme *distance* ou *plus*), sont de l'une des deux formes :

```
infix "§"
prefix "§"
```

où § peut être un symbole d'opérateurs choisi dans une liste qui contient +, -, *, <, <= et quelques autres possibilités indiquées ci-dessous. Une caractéristique ne peut avoir un nom de la forme **infix** que si c'est une fonction à un argument, comme les fonctions appelées *plus*, *minus* et *less_than* dans la version originale de la classe *REAL* ; la caractéristique ne peut avoir un nom de la forme **prefix** que s'il s'agit d'une fonction sans arguments, ou d'un attribut.

Les caractéristiques infixes et préfixes, collectivement appelées **caractéristiques d'opérateurs**, sont traitées exactement comme les autres caractéristiques (appelées

caractéristiques d'identificateurs) à l'exception des deux propriétés syntaxiques déjà mentionnées :

- Le nom d'une caractéristique d'opérateurs, tel qu'il apparaît dans la déclaration de la caractéristique, est de la forme **infix** "§" ou **prefix** "§" plutôt qu'un identificateur.
- Les appels aux caractéristiques d'opérateurs utilisent la forme *u* § *v* (dans le cas infixe) ou § *u* (dans le cas préfixe) au lieu d'être fondés sur la notation pointée.

Du fait de la seconde propriété, les caractéristiques d'opérateurs ne permettent que des appels qualifiés. Si une routine de la classe *REAL* pouvait contenir, dans la première version donnée plus tôt, un appel non qualifié de la forme *plus* (*y*), renvoyant la somme du nombre courant et de *y*, l'appel correspondant devra être écrit *Current* + *y* dans la seconde version. Avec une caractéristique d'identificateurs, la notation correspondante, *Current*·*plus* (*y*), serait possible, mais nous ne l'utiliserons pas en pratique, car elle est lourde. Avec une caractéristique d'opérateurs, nous n'avons pas le choix.

Hormis les deux différences syntaxiques mentionnées, les caractéristiques d'opérateurs sont complètement équivalentes aux caractéristiques d'identificateurs ; par exemple, elles sont héritées de la même manière. Toute classe, et pas seulement les classes de base comme *REAL*, peut utiliser des caractéristiques d'opérateurs ; par exemple, il peut être pratique, dans une classe *VECTOR*, d'avoir une fonction d'addition de vecteurs appelée **infix** "+".

La règle suivante s'appliquera aux opérateurs utilisés dans les caractéristiques d'opérateurs. Un opérateur est une séquence d'un ou de plusieurs caractères, ne contenant pas d'espace ou de retour à la ligne, et commençant avec l'un des caractères suivants :

```
+  -  *  /  <  >  =  \  ^  @  #  |  &
```

De plus, les mots-clés suivants, utilisés pour rester compatible avec la notation booléenne habituelle, sont des opérateurs autorisés :

```
not and or xor and then or else implies
```

Si l'on met de côté le cas des mots-clés, la restriction sur le premier caractère a pour objectif de préserver la clarté des textes logiciels en permettant de reconnaitre immédiatement toute utilisation d'un opérateur infixe ou préfixe par simple lecture de son premier caractère.

Les classes de base (*INTEGER*, etc.) utilisent les opérateurs suivants, appelés opérateurs standard :

- Préfixe : + - **not**.
- Infixe : + - * / < > <= >= // \\ ^ **and or xor and then or else implies**.

> La sémantique est standard. // est utilisé pour la division entière, \\ pour le reste entier, ^ pour l'opération de puissance, **xor** pour le *ou* exclusif. Dans la classe *BOOLEAN*, **and then** et **or else** sont des variantes de **and** et **or**, la différence étant expliquée dans un prochain chapitre, et **implies** est l'opérateur d'implication tel que *a* **implies** *b* est équivalent à (**not** *a*) **or else** *b*.

Voir "Opérateurs booléens non stricts", page 439.

Les opérateurs qui ne sont pas dans la liste "standard" sont appelés opérateurs libres. Voici deux exemples de caractéristiques d'opérateurs qui utilisent des opérateurs libres :

- Quand nous introduirons, par la suite, la classe *ARRAY*, nous utiliserons la caractéristique d'opérateurs **infix** "@" pour la fonction qui renvoie un élément de tableau connaissant son index, de façon que le *i*-ème élément d'un tableau *a* puisse être simplement écrit *a* @ *i*.
- Dans la classe *POINT*, nous aurions pu utiliser le nom **infix** "|-|" au lieu de *distance*, de façon que la distance entre *p1* et *p2* soit écrite *p1* |-| *p2* au lieu de *p1*·*distance*(*p2*).

La précédence de tous les opérateurs est fixe ; les opérateurs standard ont leur précédence habituelle, et tous les opérateurs libres ont priorité sur les opérateurs standard.

L'utilisation de caractéristiques d'opérateurs est un moyen pratique pour maintenir la compatibilité avec la notation usuelle d'expressions, tout en préservant un système de type complètement uniforme (comme l'exige la règle de l'objet) et un mécanisme fondamental unique de calcul. De même que traiter *INTEGER* et les autres types de base comme des classes ne doit pas affecter les performances, traiter les opérations arithmétiques et booléennes comme des caractéristiques ne doit pas réduire l'efficacité. Conceptuellement, `a + x` est un appel de caractéristique ; mais tout bon compilateur connaîtra les types de base et leurs caractéristiques, et sera capable de traiter un tel appel en générant un code au moins aussi bon que le code généré pour `a + x` en C, Pascal, Ada ou tout autre langage dans lequel `+` est une construction spéciale câblée du langage.

Quand nous utilisons des opérateurs comme `+`, `<` et autres dans des expressions, nous pouvons oublier, la plupart du temps, qu'ils correspondent, de fait, à des appels de caractéristiques ; l'effet de ces opérateurs est celui que nous pourrions attendre des approches traditionnelles. Mais il est agréable de savoir que, grâce au contexte théorique de leurs définitions, ils ne dérogent pas aux principes orientés objet et s'intègrent parfaitement au reste de la méthode.

7.8 EXPORTATIONS SÉLECTIVES ET RÉTENTION D'INFORMATION

Voir "Rétention d'information", page 52.

Dans les exemples étudiés jusqu'à présent, les caractéristiques d'une classe étaient exportées à tous ses clients potentiels. Cela n'est, bien sûr, pas toujours acceptable ; nous savons maintenant combien il est important de pratiquer la rétention d'information pour concevoir des architectures cohérentes et flexibles.

"EXPORTATION SÉLECTIVE", 23.5, page 768.

Jetons un coup d'oeil sur la manière dont nous pouvons, en fait, restreindre l'accès des caractéristiques à aucun client ou à quelques clients seulement. Cette section ne fait qu'introduire la notation ; le chapitre sur la conception des interfaces de classe examinera son utilisation raisonnée.

Exposition complète

Par défaut, comme on l'a vu, les caractéristiques déclarées sans précautions particulières sont accessibles à tous les clients. Dans une classe de la forme :

```
class S1 feature
        f ...
        g ...
        ...
end
```

les caractéristiques `f`, `g`, … sont accessibles à tous les clients de *S1*, ce qui veut dire que, dans une classe `C`, pour une entité `x` déclarée de type *S1*, un appel :

```
x.f ...
```

est valide, sous réserve que l'appel vérifie les autres conditions de validité de `f` concernant le nombre et les types des arguments, s'il y en a. (Pour simplifier, nous utiliserons ici, comme

exemples, des caractéristiques d'identificateurs, mais le raisonnement s'appliquerait de la même manière aux caractéristiques d'opérateurs, pour lesquelles les clients utiliseront des appels sous forme infixe ou préfixe plutôt que la notation pointée.)

Restreindre l'accès aux clients

Pour restreindre l'ensemble des clients qui peuvent appeler une certaine caractéristique *h*, nous utiliserons la possibilité d'avoir deux clauses **feature**, ou plus, par classe. La classe sera alors de la forme :

```
class S2 feature
        f ...
        g ...
feature {A, B}
        h ...
        ...
end
```

Les caractéristiques *f* et *g* ont le même statut qu'auparavant : elles sont accessibles à tous les clients. La caractéristique *h* n'est accessible qu'à *A* et *B* et à tous leurs descendants (les classes qui héritent directement ou indirectement de *A* ou *B*). Cela veut dire que, si *x* est déclaré avec un type *S2*, un appel de la forme :

```
x.h ...
```

est invalide, sauf s'il apparaît dans le texte de *A*, de *B* ou d'un de leurs descendants.

Dans le cas spécial où vous désirez cacher une caractéristique *i* à tous les clients, vous pouvez la déclarer comme exportée à une liste vide de clients :

```
class S3 feature {}
        i ...
end
```

Ce n'est pas le style recommandé ; voir S5 ci-dessous.

Dans ce cas, un appel de la forme `x.i (...)` est toujours invalide. Les seuls appels permis à *i* sont les appels non qualifiés de la forme :

```
i (...)
```

qui apparaissent dans le texte d'une routine de *S3* elle-même, ou de l'un de ses descendants. Ce mécanisme assure une rétention d'information complète.

La possibilité de cacher une caractéristique à tous les clients, comme l'illustre *i*, est présente dans de nombreux langages OO. Mais la plupart n'offrent pas le mécanisme sélectif illustré par *h* : exporter une caractéristique à certains clients spécifiés et à leurs descendants propres. C'est regrettable, car de nombreuses applications auront besoin d'une telle finesse de contrôle.

"Le rôle architectural des exportations sélectives", page 212.

La section de discussion de ce chapitre explique pourquoi les exportations sélectives constituent une partie critique des mécanismes architecturaux de l'approche orientée objet, évitant d'introduire des super-modules qui fragiliseraient la simplicité de la méthode.

"EXPORTATION SÉLECTIVE", 23.5, page 768.

Nous rencontrerons divers exemples d'exportation sélective dans les chapitres suivants et nous étudierons leur rôle méthodologique dans la conception de bonnes interfaces modulaires.

Style pour déclarer des caractéristiques secrètes

Une petite note sur le style. Une caractéristique déclarée sous la forme utilisée ci-dessus pour *i* est secrète, mais cette propriété n'est peut-être pas suffisamment visible avec cette seule syntaxe. En particulier, la différence avec une caractéristique publique peut ne pas suffisamment sauter aux yeux, comme dans :

Pas le style recommandé ; voir S5 ci-dessous.

```
class S4 feature
        exported...
feature { }
        secret ...
end
```

où la caractéristique *exported* est accessible à tous les clients, tandis que *secret* n'est accessible à aucun. La différence entre **feature** { } , avec une liste vide entre parenthèses, et **feature**, sans parenthèses, est un peu subtile. Pour cette raison, la notation recommandée n'utilise pas une liste vide mais une liste constituée de la seule classe *NONE*, comme dans :

Le style recommandé.

```
class S5 feature
        ... Exported ...
feature {NONE}
        ... Secret ...
end
```

"Le bas de l'échelle", page 563.

La classe *NONE*, qui sera étudiée dans un prochain chapitre consacré à l'héritage, est une classe de la bibliothèque Base qui est définie de façon à n'avoir aucune instance et aucun descendant. Ainsi, exporter une caractéristique uniquement à *NONE* revient, en pratique, à la garder secrète. En conséquence, il n'y a pas de différence sémantique entre les formes illustrées par *S4* et *S5* ; pour des raisons de clarté et de lisibilité, cependant, la seconde forme est préférable et sera employée dans le reste de ce livre toutes les fois que nous devrons introduire une caractéristique secrète.

Exporter à vous-même

Une conséquence des règles que nous avons vues jusqu'ici est qu'une classe peut avoir à exporter une caractéristique secrète. Supposons la déclaration :

```
indexing
        note: "Invalide sous la forme présente (voir les explications ci-dessous)"
class S6 feature
        x: S6
        my_routine is do ... print (x.secret) ... end
feature {NONE}
        secret: INTEGER
end -- class S6
```

En déclarant *x* de type *S6* et en effectuant l'appel *x.secret*, la classe devient son propre client. Mais cet appel est invalide puisque *secret* n'est exportée à aucune classe ! Que le client non autorisé soit *S6* elle-même ne fait pas de différence : le statut d'exportation {*NONE*} de *secret* rend tout appel *x.secret* invalide. Autoriser des exceptions nuirait à la simplicité de la règle.

La solution est simple : au lieu de **feature** { *NONE* } , l'en-tête de la seconde clause **feature** devrait être { *S6* } , qui exporte la caractéristique à la classe elle-même et à ses descendants.

Notez bien que cela n'est nécessaire que si vous voulez utiliser la caractéristique dans un appel qualifié tel que `print (x.secret)`. Si vous utilisez simplement `secret` en lui-même, comme dans l'instruction `print (secret)`, vous n'avez, bien sûr, pas besoin de l'exporter du tout. Les caractéristiques déclarées dans une classe doivent être utilisables par les routines de la classe et ses descendants ; sinon nous ne pourrions jamais rien faire avec une caractéristique secrète ! Ce n'est que si vous utilisez la caractéristique indirectement dans un appel qualifié que vous aurez besoin de l'exporter à vous-même.

7.9 REGROUPER LE TOUT

Les commentaires précédents ont introduit les mécanismes de base du calcul orienté objet, mais il nous manque encore une vue d'ensemble : comment tout cela s'exécute-t-il en pratique ?

Répondre à cette question nous aidera à rassembler nos connaissances et à comprendre comment on peut construire des systèmes exécutables à partir de classes individuelles.

Relativité générale

Ce qui est un peu déroutant, c'est que chaque description de ce qui arrive à l'exécution a été, jusqu'à présent, relative. L'effet d'une routine comme `translate` est relatif à l'instance courante ; à l'intérieur du texte de la classe, comme on l'a vu, l'instance courante n'est pas connue. Donc, nous ne pouvons essayer de comprendre l'effet d'un appel que par rapport à une cible spécifique, comme `p1` dans :

```
p1.translate (u, v)
```

Mais cela soulève la question suivante : que désigne exactement `p1` ? Ici, à nouveau, la réponse est relative. L'appel ci-dessus doit apparaître dans le texte d'une certaine classe, comme `GRAPHICS`. Supposons que `p1` soit un attribut de la classe `GRAPHICS`. Alors l'occurrence de `p1` dans l'appel, comme on l'a vu ci-dessus, peut être considérée comme un appel : `p1` correspond à `Current.p1`. Ainsi, nous n'avons fait que repousser le problème, car nous devons connaître ce à quoi correspondait l'objet `Current` au moment de l'appel ci-dessus ! En d'autres termes, nous devons nous tourner vers le client qui a appelé la routine de la classe `GRAPHICS` contenant cet appel.

Ainsi, cette tentative de compréhension d'un appel de caractéristique déclenche une chaîne de raisonnements que nous ne serons capables de suivre jusqu'au bout que si nous savons où l'exécution a démarré.

Le big-bang

Pour comprendre ce qui se passe, généralisons l'exemple ci-dessus au cas d'un appel arbitraire. Si nous arrivons à comprendre cet appel arbitraire, nous comprendrons de fait tous les calculs OO, grâce au principe de l'appel de caractéristique qui indique que :

Voir page 189.

F1 • Aucun élément logiciel n'est exécuté s'il ne fait pas partie d'un appel de routine.

F2 • Tout appel a une cible.

Tout appel sera de l'une des deux formes suivantes (la liste d'arguments peut être absente dans chacun des cas) :

- non qualifié : `f (a, b, ...)`
- qualifié : `x.g (u, v, ...)`

L'appel apparaît dans le corps d'une routine `r`. Il ne peut être exécuté que s'il fait partie d'un appel à `r`. Supposons que nous connaissions la cible de cet appel, un objet OBJ. Alors, la cible `t` est facile à déterminer dans chaque cas :

T1 • Pour une forme non qualifiée, `t` est simplement OBJ. Les cas T2, T3 et T4 s'appliqueront à la forme qualifiée.

T2 • Si `x` est un attribut, le champ `x` de OBJ a une valeur qui doit être attachée à un objet ; `t` est cet objet.

T3 • Si `x` est une fonction, nous devons tout d'abord exécuter l'appel (non qualifié) à `x` ; le résultat nous donnera `t`.

T4 • Si `x` est une entité locale de `r`, les instructions précédentes auront donné une valeur à `x` qui, au moment de l'appel, sera attachée à un certain objet ; `t` est cet objet.

Le seul problème avec ces réponses est, bien sûr, qu'elles sont relatives : elles ne nous aident que si nous connaissons l'instance courante OBJ. Quel est cet OBJ ? Mais voyons, la cible de l'appel en cours, bien sûr ! Comme dans une chanson traditionnelle (le gamin a été mangé par le chat, le chat a été mordu par le chien, le chien a été battu par le bâton), nous ne voyons pas la fin de cette chaîne.

Pour transformer ces réponses relatives en des réponses absolues, nous devons donc savoir ce qui s'est passé quand tout a commencé, au moment du big-bang. Voici la règle :

Définition : exécution d'un système

L'exécution d'un système logiciel orienté objet comprend les deux étapes suivantes :

- créer un certain objet, appelé **objet racine** de l'exécution,
- appliquer une certaine procédure, appelée **procédure de création**, à cet objet.

Au moment du big-bang, un objet est créé et la procédure de création est démarrée. L'objet racine est une instance d'une certaine classe, la **classe racine** du système ; la procédure de création est l'une des procédures de la classe racine. Dans tous les systèmes non triviaux, la procédure de création créera elle-même de nouveaux objets et appellera des routines sur ceux-ci, déclenchant d'autres créations d'objets et appels de routines. L'exécution du système correspond,

globalement, au déploiement successif de toutes les pièces d'un feu d'artifice géant et complexe, résultant directement ou indirectement de l'allumage initial d'une minuscule étincelle.

Une fois que nous savons où tout commence, il n'est pas difficile de tracer le destin de `Current` tout le long de cette réaction en chaîne. Le premier objet courant, à la source de tout (au moment du big-bang, quand la procédure de création de la racine est appelée), est l'objet racine. Puis, à chaque étape de l'exécution du système, appelons `r` la dernière routine à avoir été appelée ; si OBJ était l'objet courant au moment de l'appel de `r`, voici ce que devient `Current` durant l'exécution de `r` :

C1 • Si `r` exécute une instruction qui n'appelle pas une routine (par exemple une affectation), nous garderons le même objet comme objet courant.

C2 • Démarrer un appel non qualifié conserve le même objet comme objet courant.

C3 • Démarrer un appel qualifié `x.f` ... fait que l'objet cible de cet appel, qui est l'objet attaché à `x` (déterminé à partir de OBJ grâce aux règles appelées T1 à T4 au début de la page précédente), devient le nouvel objet courant. Quand l'appel s'achève, OBJ reprend son rôle d'objet courant.

Dans les cas C2 et C3, l'appel peut être celui d'une routine qui inclut elle-même d'autres appels, qualifiés ou non ; ainsi, cette règle doit être comprise de manière récursive.

Il n'y a donc rien de mystérieux ou de déstabilisant dans la règle qui détermine la cible de chaque appel, même si cette règle est relative et, en fait, récursive. Ce qui est déroutant, c'est la puissance des ordinateurs, puissance que nous utilisons pour jouer à l'apprenti sorcier en écrivant un texte logiciel si petit et en l'exécutant ensuite pour créer des objets et effectuer des calculs en nombre si important — le nombre des objets, le nombre des calculs — qu'il paraît presque infini quand on le mesure à l'échelle de la compréhension humaine.

Systèmes

L'essentiel de ce chapitre porte sur les classes : les composants individuels de la construction logicielle orientée objet. Pour obtenir un code exécutable, nous devons rassembler les classes en systèmes.

La définition d'un système découle de ce qui précède. Pour fabriquer un système, nous avons besoin de trois choses :

- un ensemble `CS` de classes, appelé l'**ensemble des classes** du système,
- indiquer quelle classe de `CS` est la **classe racine**,
- indiquer quelle procédure de la classe racine est la **procédure racine de création**.

Pour obtenir un système correct, ces éléments doivent vérifier une condition de cohérence, la **fermeture du système** : toute classe dont a besoin directement ou indirectement la classe racine doit faire partie de `CS`.

Soyons un petit peu plus précis :

- Une classe `C` **a directement besoin** d'une classe `D` si le texte de `C` fait référence à `D`. Il y a essentiellement deux cas dans lesquels `C` peut avoir directement besoin de `D` : `C` est un client de `D`, comme cela a été défini plus haut dans ce chapitre, et `C` hérite de `D`, selon la relation d'héritage que nous étudierons plus tard.

- Une classe `C` **a besoin** d'une classe `E`, sans autre qualification, si `C` est `E` ou si `C` a directement besoin d'une classe `D` qui (récursivement) a besoin de `E`.

Grâce à ces définitions, nous pouvons exprimer l'existence d'une fermeture comme suit :

Définition : fermeture d'un système

Un système est fermé si et seulement si son ensemble de classes contient toutes les classes dont a besoin la classe racine.

Si un système est fermé, un outil de traitement du langage comme un compilateur sera capable de traiter toutes ses classes, en débutant par la classe racine, et en gérant récursivement toutes les classes nécessaires dès qu'il rencontre leur nom. Si l'outil est un compilateur, il pourra alors produire le code exécutable correspondant au système complet.

Le processus consistant à regrouper les classes d'un système pour générer un résultat exécutable est appelé **assemblage** et constitue la dernière étape du processus de construction logicielle.

Pas de programme principal

Au cours des chapitres précédents, nous avons souvent insisté sur le fait que les systèmes développés avec la méthode orientée objet n'ont pas de notion de programme principal. En introduisant la notion de classe racine, et en exigeant que la spécification du système indique une procédure de création particulière, n'avons nous pas fait rentrer la notion de programme principal par la petite porte ?

Pas vraiment. Le problème avec la notion traditionnelle de programme principal, c'est qu'elle combine deux concepts n'ayant aucun rapport :

- l'endroit où débute l'exécution,
- le sommet, ou composant fondamental de l'architecture du système.

Le premier est évidemment nécessaire : comme tout système débute son exécution quelque part, il faut avoir un moyen permettant aux développeurs de spécifier le point de départ ; ici, ils le feront en spécifiant une classe racine et une procédure de création. (Dans le cas d'un calcul concurrent plutôt que séquentiel, il se peut que plusieurs points de départ doivent être spécifiés, un pour chaque flot de calcul.)

Concernant le concept de sommet, nous avons suffisamment insisté sur cet abus dans les chapitres précédents pour ne pas avoir à y revenir.

Mais, indépendamment de leur mérite respectif, il n'y a aucune raison de fusionner ces deux notions, de supposer que le point de départ d'un calcul jouera un rôle particulièrement important dans l'architecture du système correspondant. L'initialisation n'est qu'un des nombreux aspects d'un système. Pour prendre un exemple typique, l'initialisation d'un système d'exploitation est sa procédure de démarrage, un composant habituellement petit et relativement marginal de l'OS ; l'utiliser comme sommet de la conception du système ne conduirait pas à une architecture élégante ou utile. La notion de système, et la technologie objet en général, reposent, en fait, sur l'hypothèse inverse : la propriété la plus importante d'un système est l'ensemble des classes qu'il contient, les capacités individuelles de ces classes et

leurs relations. Selon ce point de vue, le choix d'une classe racine est une propriété secondaire et devrait être facile à modifier durant l'évolution du système.

Comme nous l'avons vu dans un chapitre précédent, la recherche d'extensibilité et de réutilisabilité exige que nous perdions l'habitude de nous demander "quelle est la fonction principale ?" lors de l'étape initiale de la conception d'un système, et d'organiser l'architecture autour de la réponse. Au contraire, l'approche prône le développement de composants logiciels réutilisables, construits comme des implémentations de types abstraits de données — les classes. Les systèmes sont alors construits comme des assemblages reconfigurables de tels composants.

Pour une critique de la décomposition fondée sur la fonctionnalité, voir "DÉCOMPOSITION FONCTIONNELLE", 5.2, page 107

En pratique, vous ne construirez pas toujours des systèmes lors de développements logiciels OO. La méthode sert souvent à développer des **bibliothèques** de composants réutilisables — les classes. Une bibliothèque n'est pas un système et n'a pas de classe racine. Quand vous développez une bibliothèque, il se peut, de temps en temps, que vous ayez besoin de produire, compiler et exécuter un ou plusieurs systèmes, mais ces systèmes sont un moyen, pas une fin en soi : ils facilitent le test des composants et ne seront habituellement pas inclus dans la bibliothèque qui sera finalement livrée. Le produit effectivement livré est l'ensemble des classes qui composent la bibliothèque, que d'autres développeurs utiliseront ensuite pour produire leur propre système — ou, à nouveau, leurs propres bibliothèques.

Assembler un système

Le processus de mise en commun d'un certain nombre de classes (dont l'une d'elles est désignée comme la racine) pour produire un système exécutable a été appelé "assemblage" ci-dessus. Comment allons-nous, en pratique, assembler un système ?

Supposons que nous ayons un système d'exploitation classique, dans lequel se trouvent les textes des classes, stockés dans des fichiers. L'outil de traitement du langage chargé de cette tâche (compilateur, interprèteur) aura besoin des informations suivantes :

A1 • le nom de la classe racine,

A2 • un **univers**, ou ensemble de fichiers, qui puisse contenir le texte des classes dont a besoin la racine (dans le sens précis, vu ci-dessus, de "besoin").

Cette information ne devrait pas être incluse dans les textes des classes eux-mêmes. Identifier une classe comme racine dans son propre texte (A1) violerait le principe "aucun programme principal". Garder dans le texte de la classe l'information concernant les fichiers où se trouvent les classes dont elle a besoin nécessiterait que ceux-ci se trouvent à un endroit précis dans le système de fichiers d'une installation donnée ; cela nuirait à l'utilisation de la classe sur une autre installation, et est donc clairement inadéquat.

Ces observations suggèrent que le processus d'assemblage d'un système devra reposer sur une information stockée en dehors du texte des classes elles-mêmes. Pour fournir cette information, nous utiliserons un petit langage de contrôle appelé Lace. Observons le processus, non sans avoir noté, au préalable, que les détails de Lace ne sont pas essentiels à la méthode ; Lace n'est qu'un exemple de langage de contrôle qui nous permet de conserver l'autonomie et la réutilisabilité des composants OO (les classes), et d'utiliser un mécanisme séparé pour leur assemblage effectif en systèmes.

Un document typique de Lace, appelé un **fichier Ace**, ressemble à ce qui suit :

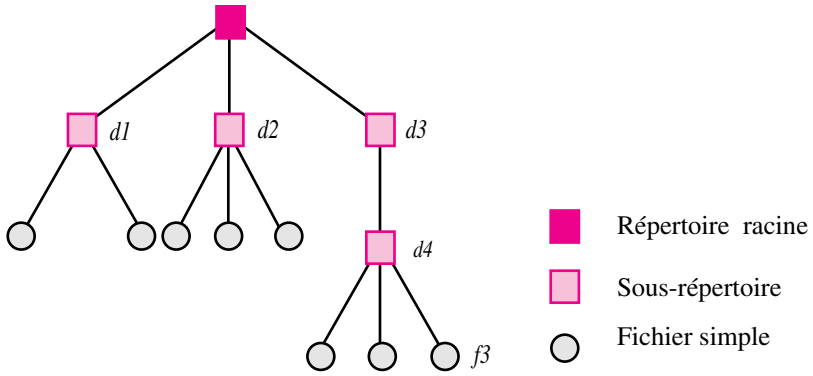
```
system painting root
        GRAPHICS ("painting_application")
cluster
        base_library: "\library\base";
        graphical_library: "\library\graphics";
        painting_application: "\user\application"
end -- system painting
```

Le chapitre 28 évoque le modèle de groupe.

La clause de groupe **cluster** définit l'univers (l'ensemble des fichiers contenant les textes de classes). Il est organisé comme une liste de groupes ; un groupe est un ensemble de classes voisines, représentant un sous-système ou une bibliothèque.

En pratique, un système d'exploitation comme Windows, VMS ou Unix fournit un mécanisme pratique pour implémenter la notion de groupe : les répertoires. Son système de fichiers est structuré comme un arbre où seuls les noeuds terminaux (les feuilles), appelés "fichiers simples", contiennent de l'information directement utilisable ; les noeuds internes, appelés répertoires, sont des ensembles de fichiers (des fichiers simples ou, à nouveau, des répertoires).

Une structure de répertoire



Nous pouvons associer à chaque groupe un répertoire. Cette convention est utilisée dans Lace comme illustré ci-dessus : chaque groupe, ayant un nom Lace comme *base_library*, possède un répertoire associé, dont le nom est donné par une chaîne entre guillemets comme "\ *library*\ *base*". Ce nom de fichier utilise les conventions Windows (nom de la forme \ *dir1*\ *dir2*\ ...), mais il s'agit simplement d'un exemple. Vous pouvez obtenir les noms Unix correspondants en remplaçant le caractère \ par /.

> Bien que, par défaut, vous puissiez utiliser la structure hiérarchique des répertoires pour représenter l'imbrication des groupes, Lace propose une notion de sous-groupe grâce à laquelle vous pouvez définir la structure logique de la hiérarchie de groupe, indépendamment des positions physiques des groupes dans le système de fichiers.

Les répertoires indiqués dans la clause **cluster** peuvent contenir des fichiers de toutes sortes. Pour déterminer l'univers, le processus d'assemblage du système devra connaître ceux qui peuvent contenir des textes de classes. Une convention simple est d'exiger que le texte de toute classe de nom *NAME* soit stocké dans un fichier de nom *name*.**e** (minuscule). Appliquons cette convention (qui peut facilement être étendue pour être plus souple) jusqu'à la fin de cet exposé. L'univers est alors l'ensemble des fichiers dont les noms sont de la forme *name*.**e** et qui se trouvent dans la liste de répertoires figurant dans la clause **cluster**.

La clause **root** de Lace sert à désigner la classe racine du système. Ici, la classe racine est *GRAPHICS* et, comme indiqué entre parenthèses, elle apparaît dans le groupe *painting_application*. S'il n'y a qu'une classe appelée *GRAPHICS* dans l'univers, il n'est pas nécessaire de spécifier le groupe.

Supposez que vous démarriez un outil de traitement du langage, par exemple un compilateur, pour traiter le système décrit par le fichier Ace ci-dessus. Supposez également qu'aucune des classes du système n'ait été compilée jusqu'à présent. Le compilateur trouve le texte de la classe racine, *GRAPHICS*, dans le fichier *graphics*.**e** du groupe *painting_application* ; ce fichier apparaît dans le répertoire *\user\application*. En analysant le texte de la classe *GRAPHICS*, le compilateur trouvera les noms des classes dont *GRAPHICS* a besoin et chargera les fichiers ayant les noms **.e** correspondants dans les trois répertoires de groupe. Il appliquera alors le même processus de recherche aux classes dont ces nouvelles classes ont besoin, en répétant le processus jusqu'à ce qu'il ait localisé toutes les classes dont la racine a besoin directement ou indirectement.

Ce processus devrait être **automatique**. En tant que développeur de logiciel, vous ne devriez pas avoir à écrire des listes de dépendance entre modules (appelées "fichiers Make") ni à indiquer dans chaque fichier les noms des fichiers qui seront nécessaires à sa compilation (ce qui se fait dans C et C++ avec des directives "include"). Devoir créer et maintenir manuellement de telles informations de dépendance est non seulement pénible, mais risque également d'engendrer des erreurs quand le logiciel évolue. Tout ce dont à besoin Ace, c'est de l'information qu'aucun outil ne peut trouver lui-même : le nom de la classe racine et la liste des positions dans le système de fichiers où peuvent se trouver les classes dont elle a besoin — ce que nous appelions plus haut l'*ensemble des classes* du système.

Pour simplifier encore davantage le travail des développeurs, un bon compilateur construira, quand il est appelé dans un répertoire où aucun fichier Ace n'est présent, un modèle Ace dont la clause **cluster** inclura les bibliothèques de base (noyau, structures de données et algorithmes fondamentaux, graphique, etc.) et le répertoire en cours, de façon que vous n'ayez plus qu'à remplir le nom du système et celui de sa classe racine, vous dispensant ainsi de vous rappeler la syntaxe de Lace.

Le résultat final du processus de compilation est un fichier exécutable dont le nom est celui donné après le mot-clé **system** dans Lace — *painting* dans l'exemple.

Le langage Lace contient quelques autres constructions simples utilisées pour commander les actions des outils de traitement du langage, en particulier les options de compilation et les niveaux de surveillance des assertions. Nous rencontrerons certaines d'entre elles lors de notre exploration des techniques OO. Lace, comme on l'a vu, propose également la notion de sous-groupe logique, qui permet de décrire des structures de systèmes complexes, utilisant les notions de sous-système et de bibliothèque à niveaux multiples.

Utiliser un langage de description de système comme Lace, séparé du langage de développement, permet aux classes de rester indépendantes du système ou des systèmes dans lesquels elles interviennent. Les classes sont des composants logiciels, semblables aux circuits en conception électronique ; un système est un assemblage particulier de classes, similaire à une carte ou à un ordinateur construit en rassemblant un certain nombre de circuits.

Imprimer votre nom

Les composants logiciels réutilisables sont intéressants, mais, parfois, tout ce que vous voulez, c'est effectuer une tâche simple, comme imprimer une chaîne. Vous vous êtes peut-être demandé comment écrire un tel programme. Maintenant que nous avons introduit la notion de système, nous pouvons répondre à cette question brûlante. (Certaines personnes ont tendance à être intimidées par l'ensemble de l'approche tant qu'elles n'ont pas vu comment faire cela, d'où cette petite digression.)

La petite classe suivante introduit une procédure qui imprime une chaîne :

```
class SIMPLE creation
      make
feature
      make is
                  -- Imprimer une chaîne exemple.
            do
                  print_line ("Hello Sarah!")
            end
end
```

Sur GENERAL, voir "Classes universelles", page 562.

La procédure *print_line* peut prendre un argument de type quelconque ; elle imprime une représentation par défaut de l'objet correspondant, ici une chaîne, sur une ligne. Est aussi disponible *print* qui ne va pas à la ligne après avoir imprimé. Ces deux procédures sont accessibles à toutes les classes, provenant d'un ancêtre universel, *GENERAL*, comme on le verra dans un prochain chapitre.

Pour obtenir un système qui imprimera la chaîne donnée, opérez comme suit :

E1 • Mettez le texte de classe ci-dessus dans un fichier appelé *simple*`.e` dans un répertoire.

E2 • Démarrez le compilateur.

E3 • Si vous n'avez pas fourni un Ace, on vous demandera d'en éditer un nouveau, généré automatiquement à partir d'un modèle ; remplissez simplement le nom de la classe racine, *SIMPLE*, le nom du système — disons *my_first* — et le répertoire du groupe.

E4 • Quittez l'éditeur ; le compilateur assemblera le système et produira un fichier exécutable appelé *my_first*.

E5 • Exécutez le résultat. Sur les plates-formes comme UNIX qui utilisent la notion de ligne de commande, une commande, de nom *my_first*, aura été créée ; tapez simplement ce nom. Sur les plates-formes graphiques comme Windows et OS/2, une nouvelle icône apparaîtra, étiquetée *my_first* ; cliquez deux fois sur cette icône.

Le résultat de la dernière étape sera d'imprimer sur votre console le message désiré.

```
Hello Sarah!
```

Structure et ordre : le développeur logiciel est un incendiaire

Nous avons maintenant une vision globale du processus de construction logicielle suivant la méthode orientée objet — assembler des classes en systèmes. Nous savons également comment reconstruire la chaîne d'événements qui amène à l'exécution d'une opération particulière. Supposez que cette opération soit :

[A]

```
x.g (u, v, ...)
```

qui apparaît dans le texte d'une routine `r` d'une classe `C` dont nous supposerons que `x` est un attribut. Comment est-elle exécutée ? Récapitulons. Nous avons dû inclure `C` dans un système et assembler ce système avec l'aide d'un Ace adéquat. Puis nous avons débuté l'exécution de ce système en créant une instance de sa classe racine. La procédure de création de la racine a forcément exécuté au moins une opération qui, directement ou indirectement, a causé la création d'une instance `C_OBJ` de `C`, et lancé l'exécution d'un appel de la forme :

[B]

```
a.r (...)
```

où `a` était, à ce moment, attaché à `C_OBJ`. Alors, l'appel montré en [A] exécutera `g`, avec les arguments indiqués, en utilisant comme cible l'objet attaché au champ `x` de `C_OBJ`.

Ainsi, dorénavant, nous savons (tout au moins, nous devrions savoir) comment déterminer la séquence exacte d'événements qui se déclencheront durant l'exécution d'un système. Mais cela suppose que nous analysions le système en entier. En général, nous ne serons pas capables, en examinant uniquement le texte d'une classe donnée, de déterminer l'ordre dans lequel les clients appelleront ces diverses routines. La seule propriété d'ordre immédiatement visible est l'ordre dans lequel une routine donnée exécute les instructions de son corps.

Même au niveau d'un système, la structure est tellement décentralisée que prévoir l'ordre précis des opérations, quoique possible en principe, est souvent difficile. Mais, et c'est plus important, ce n'est habituellement pas très intéressant. Rappelez-vous que nous traitons la classe racine comme une propriété un peu superficielle du système, un choix particulier, effectué tardivement dans le processus de développement, exprimant comment combiner un ensemble de composants individuels et ordonnancer leurs opérations disponibles.

Cette minimisation de l'importance des contraintes d'ordonnancement participe de l'effort constant de la technologie objet pour décentraliser des architectures des systèmes. L'accent n'est pas mis sur l'exécution "du" programme (comme en programmation Pascal ou C et dans de nombreuses méthodes de conception), mais sur les services fournis par un ensemble de classes via leurs caractéristiques. L'*ordre* dans lequel les services sont utilisés, durant l'exécution d'un système particulier construit à partir de ces classes, est une propriété secondaire.

La méthode va, en fait, plus loin, en indiquant que, *même si vous connaissiez* l'ordre d'exécution, vous ne devriez pas faire reposer une décision clé de conception du système sur cette connaissance. La justification de cette règle a été explorée dans les chapitres précédents : c'est une conséquence des préoccupations d'extensibilité et de réutilisabilité. Il est beaucoup plus facile d'ajouter ou de changer des services dans une structure décentralisée que de changer l'ordre des opérations si cet ordre constituait une des propriétés utilisées pour construire l'architecture. La méthode orientée objet refuse de considérer l'ordre des opérations comme une propriété fondamentale des systèmes logiciels — ce que nous avions appelé l'approche de la liste de commissions — et c'est une de ses différences majeures d'avec la plupart des autres méthodes populaires de conception de logiciels.

Voir "Ordonnancement prématuré", page 114.

Ces observations suggèrent une fois de plus l'image d'un développeur logiciel déguisé en spécialiste des feux d'artifice ou, peut-être, en incendiaire. Il prépare une déflagration géante, en s'assurant que tous les composants nécessaires sont prêts pour l'assemblage et que toutes les connexions requises sont présentes. Il allume ensuite une allumette et regarde l'incendie.

Mais, si la structure a été correctement installée et si chaque composant est correctement attaché à ses voisins, il n'est pas nécessaire de suivre ou même d'essayer de prévoir la séquence exacte de mise à feu ; il suffit de savoir que chaque partie qui doit brûler brûlera et ne le fera pas avant que son heure soit venue.

7.10 DISCUSSION

Pour conclure ce chapitre, nous allons exposer les motivations qui se trouvent derrière certaines des décisions prises lors de la conception de la méthode et de la notation, en explorant également certaines alternatives. Des sections de discussion du même genre se trouvent à la fin de la plupart des chapitres qui introduisent des constructions nouvelles ; leur but est d'aiguillonner l'esprit du lecteur en présentant un point de vue candide et libre de toute contrainte sur certaines questions délicates.

Forme des déclarations

Pour affûter notre sens critique sur quelque chose qui n'est pas trop vital, débutons avec une propriété syntaxique. Un point qui mérite d'être évoqué est la notation des déclarations de caractéristiques. Pour les routines, on ne retrouve aucun des mots-clés **procedure** ou **function** tels qu'ils apparaissent dans de nombreux langages ; la forme d'une caractéristique détermine s'il s'agit d'un attribut, d'une procédure ou d'une fonction. Le début de la déclaration d'une caractéristique est simplement le nom de la caractéristique, comme :

```
f ...
```

Une fois que vous avez lu cela, vous devez envisager toutes les possibilités. Si une liste d'arguments apparaît ensuite, comme dans :

```
g (a1: A; b1: B; ...) ...
```

vous savez que `g` est une routine ; il peut encore s'agir d'une fonction ou d'une procédure. Ensuite, peut apparaître un type :

```
f: T ...
g (a1: A; b1: B; ...): T ...
```

Dans le premier exemple, `f` peut encore être un attribut ou une fonction sans argument ; dans le second, cependant, le suspense est dissipé, puisque `g` ne peut être qu'une fonction. En revenant à `f`, l'ambiguïté sera levée par ce qui apparaît après `T` : s'il n'y a rien, `f` est un attribut, comme dans :

```
my_file: FILE
```

Mais, si un `is` est présent, suivi par un corps de routine (`do` ou les variantes `once` et `external` que nous verrons plus tard), comme dans :

```
f: T is
            -- ...
        do ... end
```

Le chapitre 18 aborde les attributs constants.

`f` est une fonction. Une autre variante est :

```
f: T is some_value
```

qui définit `f` comme un **attribut constant**, de valeur `some_value`.

La syntaxe est conçue pour faciliter la reconnaissance des différentes sortes de caractéristiques, tout en insistant sur les principales ressemblances. La notion même de caractéristique, qui recouvre les routines et les attributs, est en accord avec le principe d'accès uniforme — l'objectif consistant à fournir au client des capacités abstraites et de minimiser leurs différences de représentation. La similitude entre les déclarations de caractéristiques découle de la même idée.

Attributs et fonctions

Explorons plus avant les conséquences du principe d'accès uniforme et celles du regroupement des attributs et des routines sous la même enseigne — les caractéristiques.

"Accès uniforme", page 57 ; voir aussi, dans le présent chapitre, "Accès uniforme", page 178.

Ce principe indique que les clients d'un module devraient être capables d'utiliser tout service fourni par le module de manière uniforme, indépendamment de la façon dont le service est implémenté — via la mémoire ou par un calcul. Ici, les services sont les caractéristiques de la classe ; ce qui importe au client, c'est la disponibilité de certaines caractéristiques et leurs propriétés : qu'une caractéristique donnée soit implémentée en stockant une donnée adéquate ou en calculant le résultat à la demande est, en grande partie, sans intérêt.

Considérons, par exemple, une classe `PERSON` contenant une caractéristique `age` de type `INTEGER`, sans argument. Si l'auteur d'une classe client écrit l'expression :

```
Isabelle.age
```

la seule information importante est que `age` renverra un entier, le champ correspondant à l'âge d'une instance de `PERSON` attachée à l'exécution à l'entité `Isabelle`. En interne, `age` peut être soit un attribut, stocké avec chaque objet, soit une fonction, définie en soustrayant la valeur d'un attribut de date de naissance `birth_date` à l'année en cours. Mais l'auteur de la classe client n'a pas besoin de savoir laquelle de ces solutions a été choisie par l'auteur de `PERSON`.

La notation permettant d'*accéder* à un attribut est, donc, identique à celle utilisée pour appeler une routine ; et les notations pour *déclarer* ces deux genres de caractéristiques sont aussi proches que le concept l'autorise. Ainsi, si l'auteur d'une classe fournisseur modifie une décision d'implémentation (implémenter par une fonction une caractéristique qui était initialement un attribut, ou réciproquement), les clients ne seront pas affectés ; aucun changement, voire aucune recompilation, ne sera nécessaire.

Le contraste entre la manière dont les caractéristiques d'un module sont considérées par un fournisseur et celle utilisée par un client était apparent dans les deux figures qui, au début de ce chapitre, accompagnaient l'introduction de la notion de caractéristique. La première utilisait comme critère principal la distinction entre routine et attribut, reflétant la vue interne (implémentation), qui est aussi celle du fournisseur. Dans la seconde figure, la distinction principale concernait les commandes et les requêtes, ces dernières étant, de plus, divisées en requêtes avec ou sans arguments. C'est la vue externe — la vue du client.

Les figures se trouvent sur les pages 178 et 179.

La décision consistant à considérer les attributs et les fonctions sans arguments comme équivalents pour les clients a deux conséquences importantes que les prochains chapitres développeront :

"Documentation et assertions : la forme abrégée d'une classe", page 378.

- La première conséquence concerne la documentation logicielle. La documentation standard d'un client d'une classe, connue sous le nom de **forme abrégée** de la classe, sera conçue de manière à ne pas révéler si une caractéristique donnée est un attribut ou une fonction (dans les cas où elle pourrait être l'un ou l'autre).

"Redéclarer une fonction en un attribut", page 475.

- La seconde conséquence concerne l'héritage, la principale technique permettant d'adapter des composants logiciels à de nouvelles circonstances sans interférer avec le logiciel existant. Si une classe donnée introduit une caractéristique comme fonction sans arguments, les classes descendantes pourront **redéfinir** la caractéristique comme un attribut, substituant la mémoire au calcul.

Exporter des attributs

Le texte de la classe se trouve page 180.

Une conséquence des observations précédentes est que les classes peuvent exporter des attributs. Par exemple, la classe *POINT*, dans l'implémentation cartésienne présentée plus haut, contient les attributs *x* et *y*, et les exporte aux clients de la même manière que les fonctions *rho* et *theta*. Pour obtenir la valeur d'un attribut pour un certain objet, vous utilisez simplement la notation d'appel de caractéristique, comme *my_point*.*x* ou *my_point*.*theta*

Cette possibilité d'exporter des attributs nous éloigne des conventions qui existent dans de nombreux langages OO. Parmi ceux-ci, Smalltalk ne permet d'exporter que les routines (appelées "méthodes") ; les attributs ("variables d'instance") ne sont pas directement accessibles au client.

Une conséquence de l'approche Smalltalk est que, si vous voulez obtenir l'équivalent de l'exportation d'un attribut, vous devez écrire une petite fonction exportée dont le seul effet est de renvoyer la valeur de l'attribut. Ainsi, dans l'exemple *POINT*, nous pourrions appeler les attributs *internal_x* et *internal_y*, et écrire la classe comme suit (en utilisant la notation de ce livre plutôt que la notation exacte de Smalltalk, et en appelant les fonctions *abscissa* et *ordinate* plutôt que *x* et *y* pour éviter toute confusion) :

```
class POINT feature -- Caractéristiques publiques :
        abscissa: REAL is
                    -- Coordonnée horizontale
                do Result := internal_x end

        ordinate: REAL is
                    -- Coordonnée verticale
                do Result := internal_y end

        ... Autres caractéristiques de la version précédente ...

feature {NONE} -- Caractéristiques inaccessibles aux clients :

        internal_x, internal_y: REAL

end
```

Cette approche a deux inconvénients :

- Elle force les auteurs des classes fournisseurs à écrire de nombreuses petites fonctions comme *abscissa* et *ordinate*. Bien que, en pratique, de telles fonctions soient courtes (puisque la syntaxe de Smalltalk est concise et permet de donner le même nom à un attribut et à une fonction, évitant d'avoir à inventer des noms d'attributs spéciaux comme *internal_x* et *internal_y*), les écrire est néanmoins un gaspillage d'effort de la part de l'auteur de la classe, et les lire une distraction inutile pour le lecteur de la classe.
- La méthode entraîne une perte significative de performance : tout accès au champ d'un objet nécessite maintenant un appel de routine. Cela explique pourquoi la technologie objet a gagné, dans certains cercles, la réputation d'être inefficace. (Il est possible de développer un

compilateur optimisant qui expansera en ligne les appels aux fonctions du style *abscissa*, mais on peut alors se demander quel est le rôle de ces fonctions.)

La technique évoquée dans ce chapitre semble préférable. Elle permet d'éviter de compliquer le texte des classes avec de nombreuses petites fonctions supplémentaires et, à la place, laisse les concepteurs de la classe exporter les attributs si besoin est. Contrairement à ce qu'une analyse superficielle pourrait suggérer, cette politique ne viole pas la rétention d'information ; elle est, en fait, une implémentation directe de ce principe et du principe associé d'accès uniforme. Pour remplir ces exigences, il suffit de s'assurer que les attributs, tels qu'ils sont vus par les clients, sont indiscernables des fonctions sans arguments, et qu'ils ont les mêmes propriétés d'héritage et de documentation de classe.

Cette technique permet d'atteindre les objectifs d'accès uniforme (essentiel aux clients), de facilité d'écriture des textes des classes (essentiel au fournisseur) et d'efficacité (essentiel à tous).

Les privilèges d'un client sur un attribut

Exporter un attribut en utilisant les techniques que nous venons d'examiner permet aux clients d'accéder à la valeur d'un attribut de certains objets, comme dans *my_point*•*x*. Cela ne permet pas aux clients de modifier cette valeur. Vous ne pouvez pas affecter un attribut ; l'affectation :

Attention : construction illégale — à but d'illustration uniquement.

```
my_point•x := 3.7
```

est syntaxiquement illégale. La règle de syntaxe est simple : *a*•*attrib*, si *attrib* est un attribut (ou, d'ailleurs, une fonction), est une expression, pas une entité ; ainsi, vous ne pouvez pas l'affecter, comme vous ne pouvez pas affecter l'expression *a + b*.

Pour rendre *attrib* accessible en modification, vous devez écrire et exporter une procédure appropriée, de la forme :

```
set_attrib (v: G) is
            -- Affecter à v la valeur de attrib.
      do
            attrib := v
      end
```

Plutôt que cette convention, on aurait pu imaginer une syntaxe permettant de spécifier des droits d'accès, comme :

Attention : ce n'est pas une notation définitive. Seulement à titre d'étude.

```
class C feature [ AM]
      ...
feature [ A] { D, E}
      ...
```

où *A* voudrait dire accès et *M* modification. (Spécifier *A* pourrait être optionnel : si vous exportez quelque chose, vous devez au moins permettre au client d'y accéder en mode lecture). Cela permettrait d'éviter l'écriture répétitive de procédures semblables à *set_attrib*.

Outre qu'elle ne justifie pas une complication supplémentaire du langage, cette solution n'est pas suffisamment flexible. Dans de nombreux cas, vous voudrez exporter des méthodes *spécifiques* pour modifier un attribut. Par exemple, la classe suivante exporte un compteur et le droit de le modifier, non pas arbitrairement, mais seulement par incrément de +1 ou –1 :

```
class COUNTING feature
        counter: INTEGER
        increment is
                        -- Incrémenter le compteur.
                do
                        counter := counter + 1
                end
        decrement is
                        -- Décrémenter le compteur.
                do
                        counter := counter − 1
                end
end
```

De même, la classe *POINT* que nous avons développée dans ce chapitre n'autorise pas ses clients à positionner directement les *x* et *y* d'un point ; les clients ne peuvent changer les valeurs de ces attributs que via les mécanismes spécifiques exportés à cette intention, les procédures *translate* et *scale*.

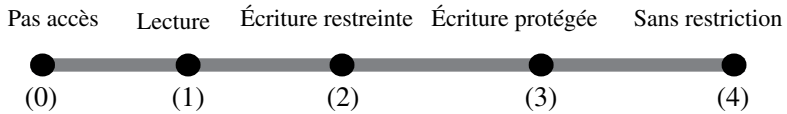
Quand nous étudierons les assertions, nous verrons une autre raison fondamentale de ne pas autoriser les clients à effectuer des affectations directes de la forme *a.attrib := some_value* : toutes les valeurs *some_value* ne sont pas acceptables. Vous pouvez définir une procédure comme :

```
set_polygon_size (new_size: INTEGER) is
                -- Positionner le nombre de noeuds d'un polygone à new_size.
        require
                new_size >= 3
        do
                size := new_size
        end
```

qui impose que tout argument réel soit supérieur ou égal à 3. Des affectations directes empêcheraient d'imposer cette contrainte ; un appel pourrait alors produire un objet incorrect.

Ces considérations montrent que l'auteur d'une classe doit avoir à sa disposition, pour chaque attribut, *cinq* niveaux possibles de droits d'accès qu'il peut offrir aux clients :

Privilèges possibles d'un client sur un attribut



Le niveau 0 correspond à une protection totale : les clients n'ont aucun moyen d'accéder à l'attribut. Aux niveaux 1 et plus, vous rendez l'attribut disponible en lecture mais, au niveau 1, vous n'autorisez pas les modifications. Au niveau 2, vous permettez aux clients de modifier l'attribut via certains algorithmes spécifiques. Au niveau 3, vous les laissez positionner la valeur, mais seulement si elle vérifie certaines contraintes, comme dans l'exemple de la taille d'un polygone. Le niveau 4 élimine toute contrainte.

La solution décrite dans ce chapitre est une conséquence de cette analyse. Exporter un attribut ne donne au client qu'un droit d'accès (niveau 1) ; le droit de modifier est spécifié en écrivant et en exportant des procédures appropriées, qui donnent au client des droits restreints comme

dans les exemples du compteur et du point (niveau 2), des droits de modification directe sous contraintes (3) ou des droits sans restriction (4).

Cette solution est une amélioration par rapport à celles que l'on trouve communément dans les langages OO :

- Dans Smalltalk, comme on l'a vu, vous devez écrire des fonctions spéciales d'encapsulation, comme `abscissa` et `ordinate`, ne serait-ce que pour laisser les clients accéder à un attribut au niveau 1 ; cela implique à la fois du travail supplémentaire pour le développeur et une pénalité en performance. Ici, il n'est pas nécessaire d'écrire des routines d'accès aux attributs ; seules les modifications d'attributs (niveau 2 et plus) nécessitent d'écrire une routine, car cela est conceptuellement nécessaire pour les raisons que nous venons d'évoquer.
- C++ et Java correspondent à l'autre extrême : si vous exportez un attribut, alors il est manipulable jusqu'au niveau 4 : les clients peuvent l'affecter directement suivant le style `my_point.x := 3.7`, tout comme ils peuvent accéder à sa valeur. La seule manière d'obtenir le niveau 2 (pas de niveau 3, car ces langages ne contiennent pas de mécanisme d'assertions OO) est de cacher complètement les attributs et d'écrire ensuite des routines d'exportation, à la fois des procédures pour modifier (niveaux 2 ou 4) et des fonctions d'accès (niveau 1). Mais vous obtenez alors le même comportement qu'avec l'approche Smalltalk.

Cette discussion d'un aspect relativement spécifique du langage illustre deux des principes généraux de conception de langage : ne tourmentez pas inutilement le programmeur ; sachez arrêter d'introduire de nouvelles constructions dans le langage quand le retour sur investissement commence à diminuer.

Optimiser les appels

Aux niveaux 2 et 3 de l'étude précédente, l'utilisation d'appels de procédure explicites comme `my_polygon.set_size (5)` pour changer la valeur d'un attribut est inévitable. Au niveau 4, on pourrait craindre les conséquences sur la performance qu'amène l'utilisation du style `set_attrib`. Le compilateur, cependant, peut générer pour `my_point.set_x (3.7)` le même code que pour `my_point.x := 3.7`, si ce type d'expression était légal.

> Le compilateur d'ISE y parvient grâce à un mécanisme général d'expansion en ligne qui élimine certains appels de routine en insérant le corps de la routine directement dans le code de l'appelant, après avoir correctement substitué les arguments.

L'expansion en ligne est, de fait, l'une des transformations que nous sommes en droit d'attendre d'un compilateur optimisant pour un langage orienté objet. Le style modulaire de développement que favorise la technologie objet produit un grand nombre de petites routines. Il serait inacceptable que les développeurs aient à se soucier de l'effet sur les performances des appels correspondants. Ils devraient simplement concevoir l'architecture la plus claire et la plus robuste possible selon les principes de modularité étudiés dans ce livre, et compter sur le compilateur pour éliminer ceux des appels qui, pertinents lors de la conception, ne sont plus nécessaires lors de l'exécution.

Dans certains langages de programmation, en particulier Ada et C++, les développeurs spécifient quelles sont les routines qu'ils souhaitent voir expanser en ligne. Je trouve préférable de laisser cette tâche à un optimiseur automatique, pour plusieurs raisons :

- Il n'est pas toujours correct d'expanser un appel en ligne ; puisque le compilateur doit, pour des raisons de correction, vérifier que l'optimisation est possible, on peut donc tout simplement éviter aux développeurs d'avoir à la demander.
- Au fur et à mesure des changements du logiciel, en particulier du fait de l'héritage, une routine qui pouvait être expansée peut ne plus le rester. Un outil logiciel est plus à même qu'un être humain de détecter de telles situations.
- Pour un grand système, les compilateurs seront toujours plus efficaces. Ils sont mieux armés pour appliquer les heuristiques permettant, en fonction de la taille des routines et du nombre d'appels, de décider quelles routines devraient être expansées en ligne. C'est, à nouveau, particulièrement critique lors de l'évolution du logiciel ; nous ne pouvons pas attendre d'un être humain qu'il suive l'évolution de chaque élément logiciel.
- Les développeurs logiciels ont mieux à faire.

"Exigences d'un ramasse-miettes", page 299, et "L'approche C++ de la liaison", page 498.

Dans la vision actuelle du génie logiciel, des optimisations aussi pénibles, automatisables et délicates devraient être effectuées par des outils logiciels, et non par des individus. La politique consistant à les laisser au développeur est l'une des critiques principales portées contre C++ et Ada. Nous rencontrerons à nouveau ce débat quand nous étudierons deux autres mécanismes clés de la technologie objet : la gestion de la mémoire et la liaison dynamique.

Le rôle architectural des exportations sélectives

La possibilité d'imposer des exportations sélectives n'est pas simplement pratique ; elle est essentielle à l'architecture orientée objet. Elle permet à un ensemble de classes conceptuellement voisines de rendre certaines de leurs caractéristiques accessibles aux autres sans avoir à les dévoiler au reste du monde, c'est-à-dire sans violer la règle de rétention d'information. Elle nous permet également de répondre à une question souvent abordée : faut-il des modules au dessus des classes ?

Sans les exportations sélectives, la seule solution (si l'on ne veut pas renoncer à la rétention d'information) serait d'introduire une nouvelle structure modulaire pour regrouper les classes. De tels supermodules, semblables aux paquetages d'Ada ou de Java, auraient leurs propres règles pour cacher et exporter. En ajoutant un niveau de module complètement neuf et partiellement incompatible avec le cadre élégant défini par les classes, on rendrait le langage plus difficile à apprendre et plus complexe.

Plutôt que d'utiliser une construction séparée de paquetage, les supermodules pourraient être eux-mêmes des classes ; c'est l'approche de Simula, qui permet l'emboîtement des classes. Cela entraîne également une complexité supplémentaire, sans apporter de bénéfices bien nets.

Nous avons vu que la simplicité de la technologie objet repose en grande partie sur l'utilisation d'un concept modulaire unique, la classe ; son apport à la réutilisabilité vient de notre capacité à extraire une classe de son contexte en ne gardant que ses dépendances logiques. Avec un concept de supermodule, nous courons le risque de perdre ces avantages. En particulier, si une classe appartient à un paquetage ou à une classe englobante, nous ne pourrons pas la réutiliser en elle-même ; si nous voulons l'inclure dans un autre supermodule, nous devrons importer le supermodule en entier ou réaliser une copie de la classe — ce qui n'est pas particulièrement exaltant.

Le besoin de regrouper les classes en collections structurées subsistera. Cela sera abordé dans un prochain chapitre avec la notion de *groupe*. Mais le groupe est une notion de gestion et d'organisation ; en faire une construction du langage mettrait en péril la simplicité de l'approche orientée objet et son avantage pour la modularité.

Chapitre 28.

Si nous voulons autoriser un groupe de classes à s'allouer entre elles des privilèges spéciaux, nous n'avons pas besoin d'un supermodule ; les exportations sélectives, une extension modeste à la rétention de base d'information, fournissent une solution simple, permettant aux classes de garder leurs statuts de composants logiciels indépendants. C'est, selon moi, un cas typique où une idée simple et peu sophistiquée peut s'avérer plus performante que l'artillerie lourde d'un mécanisme "puissant".

Répertorier les importations

Chaque classe précise, dans les en-têtes de ses clauses **feature**, les caractéristiques qu'elle veut rendre accessibles aux autres. On pourrait se demander pourquoi on ne recenserait pas également les caractéristiques obtenues des autres classes ? Le langage d'encapsulation Modula-2 fournit effectivement une clause **import**.

Dans une approche typée de la construction OO, une telle clause ne servira, cependant, à rien, si ce n'est à des fins de documentation. Pour utiliser une caractéristique `f` d'une autre classe `C`, vous devez être un client ou (par héritage) un descendant de cette classe. Dans le premier cas, le seul que nous ayons vu jusqu'ici, cela veut dire que chaque utilisation de `f` est de la forme :

```
a.f
```

où, puisque notre notation est typée, `a` doit avoir été déclaré :

```
a: C
```

indiquant sans ambiguïté que `f` vient de `C`. Dans le cas d'une descendance, l'information sera disponible dans la documentation partielle de la classe, sa "forme abrégée plate".

"La forme abrégée plate", page 525.

Ainsi, il n'est pas nécessaire d'encombrer les développeurs avec des clauses d'importation.

Il est *utile*, cependant, d'aider les développeurs avec une documentation d'importation. Un bon environnement de développement graphique devrait proposer des mécanismes vous permettant, en cliquant sur un bouton, de voir les fournisseurs et les ancêtres d'une classe, et de suivre la chaîne d'importation plus avant en explorant leurs propres fournisseurs et ancêtres.

Voir chapitre 36.

Désigner le résultat d'une fonction

Un aspect intéressant des langages, évoqué plus haut dans ce chapitre, est de savoir comment désigner les résultats d'une fonction. Il mérite d'être exploré plus à fond, bien qu'il concerne également les langages non OO.

Considérons une fonction — une routine qui renvoie une valeur. Puisque le but de tout appel à une fonction est de calculer un certain résultat et de le renvoyer à l'appelant, la question se pose de savoir comment noter ce résultat dans le texte de la fonction elle-même, en particulier dans les instructions qui initialisent et mettent à jour le résultat.

La convention introduite dans ce chapitre utilise une entité spéciale, *Result*, considérée comme une entité locale et initialisée avec une valeur par défaut appropriée ; le résultat renvoyé par un appel est la valeur finale de *Result*. Du fait des règles d'initialisation, cette valeur est toujours définie, même si le corps de la routine ne contient pas d'affectation à *Result*. Par exemple, la fonction :

```
f: INTEGER is
        do
                if some_condition then Result := 10 end
        end
```

renverra la valeur 10, si *some_condition* est vérifié au moment de l'appel, et 0 (la valeur d'initialisation par défaut pour *INTEGER*) sinon.

La technique fondée sur *Result* a été introduite, autant que je sache, avec la notation développée dans ce livre. (Depuis la première édition, elle a été réutilisée dans au moins un autre langage, Delphi de Borland.) Notez que cela ne marcherait pas dans un langage qui permettrait de déclarer des fonctions à l'intérieur d'autres fonctions, puisque le nom *Result* deviendrait alors ambigu. Parmi les techniques utilisées dans les langages antérieurs, les plus courantes sont :

A • introduire des instructions explicites de renvoi (C, C++/Java, Ada, Modula-2),

B • traiter le nom d'une fonction comme une variable (Fortran, Algol 60, Simula, Algol 68, Pascal).

La convention A repose sur une instruction de la forme **return** *e* dont l'exécution termine l'exécution en cours de la fonction englobante, renvoyant *e* comme résultat. Cette technique a le bénéfice de la clarté, puisqu'elle met clairement en exergue la valeur renvoyée dans le texte de la fonction. Mais elle présente plusieurs inconvénients :

A1 • Le résultat doit souvent, en pratique, être obtenu par un calcul : une initialisation suivie de quelques mises à jour. Cela veut dire que vous devez introduire et déclarer une variable supplémentaire (une entité, selon la terminologie de ce chapitre) dans le seul but de manipuler les résultats intermédiaires de calcul.

A2 • La technique tend à promouvoir des modules ayant plusieurs sorties, ce qui est contraire aux principes d'une bonne structuration des programmes.

A3 • La définition du langage doit spécifier ce qui arrive si la dernière instruction exécutée par un appel à la fonction n'est pas un **return**. Le résultat proposé par Ada est, dans ce cas, de lever une exception à l'exécution ! (Cela peut paraître la meilleure façon de repasser le bébé, les concepteurs du langage ayant transféré la responsabilité des problèmes de conception du langage non seulement aux développeurs logiciels mais aux *utilisateurs finaux* des programmes développés dans le langage).

Notez qu'il est possible de résoudre les deux derniers problèmes en traitant **return** non comme une instruction mais comme une clause syntaxique qui serait un élément obligatoire de tout texte de fonction :

```
function name (arguments): TYPE is
        do
                ...
        return
                expression
        end
```

Cette solution reste compatible avec l'idée d'une instruction **return**, tout en éliminant ses inconvénients les plus sérieux. Aucun langage usuel, cependant, ne l'utilise et, bien sûr, elle laisse pendant le problème A1.

La seconde technique classique, B, considère le nom d'une fonction comme étant une variable à l'intérieur du texte de la fonction. La valeur renvoyée par un appel est la valeur finale de cette variable. (Cela dispense d'introduire une variable spéciale, comme on l'a vu pour A1.)

Les trois problèmes ci-dessus ne se posent pas dans cette approche, mais d'autres difficultés surgissent, car le même nom désigne maintenant, de manière ambiguë, à la fois une fonction et une variable. Cela est particulièrement gênant dans un langage qui permet la récursion, quand le corps d'une fonction peut utiliser le nom de la fonction pour désigner un appel récursif. Puisqu'une occurrence du nom de la fonction a maintenant deux sens possibles, le langage doit définir des conventions précises pour distinguer une référence à la variable d'un appel de fonction. Habituellement, dans le corps d'une fonction *f*, une occurrence du nom *f* comme cible d'une affectation (ou dans d'autres contextes qui impliquent qu'une valeur est modifiée) désigne la variable, comme dans :

```
f := x
```

et une occurrence de *f* dans une expression (ou d'autres contextes qui impliquent qu'on accède à une valeur) désigne un appel récursif de fonction, comme dans :

```
x := f
```

qui n'est valide que si *f* n'a pas d'arguments. Mais, alors, une affectation de la forme :

```
f := f + 1
```

sera rejetée par le compilateur (si *f* a des arguments) ou, pire, comprise comme contenant un appel récursif dont le résultat est affecté à *f* (la variable). Cette dernière interprétation est sans doute différente de ce à quoi pensait le développeur : si *f* avait été une variable normale, l'instruction aurait simplement augmenté sa valeur de un. Ici, l'affectation créera vraisemblablement un calcul qui ne termine jamais. Pour obtenir l'effet désiré, le développeur devra introduire une variable supplémentaire ; cela nous ramène au problème A1 ci-dessus et élimine la raison d'être de la technique B.

La convention introduite dans ce chapitre, et qui repose sur l'entité prédéfinie *Result*, évite les inconvénients de A et B. Un avantage supplémentaire, dans un langage qui fournit des initialisations par défaut à toutes les entités, y compris *Result*, est qu'elle simplifie l'écriture des fonctions : si, comme cela se présente souvent, vous voulez que le résultat soit la valeur par défaut, sauf dans certains cas spécifiques, vous pouvez utiliser le schéma :

```
do
        if some_condition then Result := "Une valeur spécifique" end
end
```

sans introduire de clause **else**. La définition du langage doit, bien sûr, spécifier toutes les valeurs par défaut de manière non ambiguë et indépendante de la plate-forme ; le prochain chapitre introduira de telles conventions pour notre notation. *Page 234.*

Un dernier bénéfice de la convention *Result* deviendra clair quand nous étudierons la conception par contrat : nous pouvons utiliser *Result* pour exprimer une propriété abstraite du résultat d'une fonction, indépendamment de son implémentation, dans la postcondition d'une routine. Aucune des autres conventions ne nous permettrait d'écrire : *Chapitre 11.*

```
prefix "|_": INTEGER is
            -- Partie entière
      do
            ... Implémentation omise ...
      ensure
            no_greater: Result <= Current
            smallest_possible: Result + 1 > Current
      end
```

La postcondition est la clause **ensure**, qui indique deux propriétés du résultat : il n'est pas plus grand que la valeur à laquelle a été appliquée la fonction ; et lui ajouter 1 rendrait un résultat plus grand que cette valeur.

Complément : une définition précise des entités

Il sera utile, puisque nous en sommes aux problèmes de notation, de clarifier une notion qui a été souvent utilisée précédemment, mais sans avoir été définie précisément : les entités. Plutôt qu'une notion fondamentale de la technologie objet, c'est simplement une notion technique qui généralise la notion traditionnelle de variable ; nous avons besoin d'une définition précise.

Les entités, telles qu'elles sont utilisées dans ce livre, recouvrent les noms qui désignent des valeurs à l'exécution, étant éventuellement attachées elles-mêmes à des objets. Nous avons vu maintenant les trois cas possibles :

Définition : entité

Une entité est l'un des cas suivants :
E1 • un attribut d'une classe,
E2 • une entité locale d'une routine, incluant l'entité prédéfinie *Result* pour une fonction,
E3 • un argument formel d'une routine.

Le cas E2 indique que l'entité *Result* est traitée, dans tous les cas, comme une entité locale ; les autres entités locales sont introduites dans la clause **local**. *Result* et les autres entités locales d'une routine sont initialisées chaque fois que la routine est appelée.

Toutes les entités, sauf les arguments formels, sont modifiables, c'est-à-dire qu'elles peuvent apparaître comme cible *x* d'une affectation *x := some_value*.

7.11 CONCEPTS CLÉS INTRODUITS DANS CE CHAPITRE

- Le concept fondamental de la technologie objet est la notion de classe. Une classe est un type abstrait de données implémenté partiellement ou complètement.
- Une classe peut avoir des instances, appelées objets.
- Ne confondez pas les objets (des éléments dynamiques) et les classes (la description statique des propriétés communes à un ensemble d'objets à l'exécution).
- Dans une approche cohérente de la technologie objet, chaque objet est une instance d'une classe.
- La classe sert à la fois de module et de type. L'originalité et la puissance du modèle OO viennent, en partie, de la fusion de ces deux notions.

- Une classe est définie par des caractéristiques, incluant des attributs (représentant des champs des instances de la classe) et des routines (représentant des calculs sur ces instances). Une routine peut être une fonction, qui renvoie un résultat, ou une procédure, qui ne le fait pas.
- Le mécanisme de base du calcul orienté objet est l'appel de caractéristique. Un appel de caractéristique applique une caractéristique d'une classe à une instance de cette classe, éventuellement avec des arguments.
- Un appel de caractéristique utilise soit la notation de point (pour les caractéristiques d'identificateurs), soit la notation d'opérateurs, préfixe ou infixe (pour les caractéristiques d'opérateurs.)
- Toute opération est relative à l'"instance courante" d'une classe.
- Pour les clients d'une classe (d'autres classes qui utilisent les caractéristiques de celle-ci), un attribut n'est pas distinguable d'une fonction sans arguments, en accord avec le principe d'accès uniforme.
- Un assemblage exécutable de classes est appelé un système. Un système contient une classe racine et toutes les classes dont la racine a besoin directement ou indirectement (via les relations client et d'héritage). Exécuter le système revient à créer une instance de la classe racine et à appeler une procédure de création de cette instance.
- Les systèmes devraient avoir une architecture décentralisée. Les relations d'ordre entre les opérations ne sont pas essentielles à la conception.
- Un petit langage de description de systèmes, Lace, permet de spécifier la façon dont un système devrait être assemblé. Une spécification Lace, ou Ace, indique la classe racine et l'ensemble des répertoires où se trouvent les groupes du système.
- Le processus d'assemblage d'un système devrait être automatique, sans nécessiter de fichiers Make ou de directives include.
- Le mécanisme de rétention d'information doit être flexible : outre le fait d'être cachée ou accessible à tous, une caractéristique peut être exportée seulement à certains clients ; et un attribut peut être exporté en lecture uniquement, lecture et modification restreinte, ou modification complète.
- Exporter un attribut donne au client le droit d'y accéder. Le modifier impose d'appeler une procédure exportée appropriée.
- Les exportations sélectives sont nécessaires pour permettre aux groupes de classes fortement corrélées d'avoir un accès spécial aux caractéristiques des autres.
- Il n'est pas nécessaire d'introduire une construction de supermodule au-dessus des classes. Les classes devraient rester des composants logiciels indépendants.
- Le style modulaire favorisé par le développement orienté objet conduit à un grand nombre de petites routines. L'expansion en ligne, une optimisation effectuée par le compilateur, élimine toute conséquence néfaste pour l'efficacité. Détecter les appels qui peuvent être expansés devrait être du ressort du compilateur, non des développeurs logiciels.

7.12 NOTES BIBLIOGRAPHIQUES

La notion de classe vient du langage Simula 67 ; voir les références bibliographiques dans le chapitre correspondant. Une classe Simula est, à la fois, un module et un type, bien que cette propriété ne soit pas mise en avant dans la littérature Simula et ait été abandonnée chez certains de ses successeurs.

Chapitre 35, bibliographie en page 1106.

Le principe de cible unique peut être considéré comme l'équivalent logiciel d'une technique bien connue en logique mathématique et en informatique théorique : la **currification**. Currifier une fonction à deux arguments `f` revient à la remplacer par une fonction à un argument `g` renvoyant une fonction à un argument comme résultat, tel que pour tous `x` et `y` applicables :

```
(g (x)) (y) = f (x, y)
```

Currifier une fonction revient, en d'autres termes, à spécialiser son premier argument. Cela est semblable à la transformation décrite dans ce chapitre pour remplacer une routine traditionnelle à deux arguments `rotate`, appelée sous la forme :

```
rotate (some_point, some_angle)
```

par une fonction avec une cible, appelée sous la forme :

```
some_point.rotate (some_angle)
```

Chapitre 32 (discussion sur le CD).

[M 1990] décrit la currification et certaines de ses applications en informatique, en particulier l'étude formelle de la syntaxe et de la sémantique des langages de programmation. Nous rencontrerons à nouveau la currification lors de l'étude des interfaces graphiques utilisateur.

Quelques conceptions de langage ont utilisé le concept d'objet comme construction logicielle plutôt qu'une simple notion à l'exécution, telle qu'elle est décrite dans ce chapitre. Une telle approche, qui vise la programmation exploratoire, ne nécessite pas de notion de classe. Le représentant le plus significatif de cette école de pensée est le langage Self [Chambers 1991], qui utilise des prototypes plutôt que des classes.

Le détail des conventions pour les opérateurs infixe et préfixe, en particulier la table de précédence, se trouve dans [M 1992].

James McKim m'a suggéré l'argument final concernant la convention `Result` (son utilisation dans les postconditions).

EXERCICES

E7.1 Clarifier la terminologie

[Cet exercice requiert des crayons bien taillés, un *bleu* et un *rouge*.]

Voir "Que penseriez-vous de ceci ?", page 170.

Étudiez l'extrait utilisé précédemment dans ce chapitre pour illustrer la confusion entre objet et classe ; à chaque utilisation du mot "objet", "chose" ou "utilisateur" dans cet extrait, soulignez le mot en *bleu* si vous pensez que les auteurs voulaient réellement dire objet ; soulignez le mot en *rouge* si vous pensez qu'ils voulaient vraiment dire classe.

E7.2 `POINT` comme type abstrait de données

Ecrivez une spécification de type abstrait de données pour la notion de point à deux dimensions, comme cela était suggéré dans l'introduction informelle de cette notion.

E7.3 Terminer `POINT`

Page 180.

Terminer le texte de la classe `POINT` en remplissant les détails manquants et en ajoutant une procédure `rotate` (pour faire tourner un point autour de l'origine), ainsi que toutes les autres caractéristiques que vous jugez nécessaires.

E7.4 Coordonnées polaires

Écrivez le texte de la classe `POINT` de façon à utiliser une représentation polaire plutôt que cartésienne.