Conception et programmation orientées objet

Bertrand Meyer

Traduit de l'anglais par Pierre Jouvelot

 ISBN : 978-2-212-12270-1

EYROLLES

8

La structure à l'exécution : les objets

Dans le chapitre précédent, nous avons vu que les classes peuvent avoir des instances, appelées objets. Il nous faut maintenant tourner notre attention vers ces objets et, plus généralement, vers le modèle d'exécution du calcul orienté objet.

Alors que les chapitres précédents étaient essentiellement consacrés à des questions conceptuelles et structurelles, celui-ci contiendra, pour la première fois dans ce livre, des aspects d'implémentation. En particulier, il décrira comment est utilisée la mémoire lors de l'exécution d'un logiciel orienté objet — un développement qui sera poursuivi par l'étude du ramasse-miettes dans le prochain chapitre. Comme on l'a vu, un des avantages de la technologie objet est de remettre les questions d'implémentation à leur place ; ainsi, même si vous êtes essentiellement intéressé par l'analyse et la conception, vous ne devriez pas vous inquiéter de cette excursion dans le territoire des implémentations. Il est impossible de comprendre la méthode sans avoir une idée approximative de son influence sur les structures à l'exécution.

Dans ce chapitre, l'étude de la structure des objets fournit, effectivement, un exemple particulièrement éclairant de l'erreur qui consiste à séparer les aspects d'implémentation des aspects de, soit disant, plus haut niveau. Tout au long de cet exposé, chaque fois que nous aurons besoin d'une nouvelle technique OO ou d'un nouveau mécanisme, introduit initialement pour des raisons liées à l'implémentation, la vraie raison s'avérera presque toujours être plus fondamentale : nous avons également besoin de ce service pour des raisons essentiellement abstraites et aussi descriptives. Un exemple typique en sera la distinction entre les références et les valeurs expansées, qui pourrait paraître initialement comme une obscure technique de programmation, mais qui fournit, en réalité, une réponse générale à la question du partage dans les relations ensemble-élément, question qui se pose fréquemment dans de nombreux traités d'analyse orientée objet.

Cette contribution de l'implémentation est parfois difficile à admettre pour ceux qui ont été influencés par l'idée, encore fortement présente dans la littérature logicielle, que l'analyse est la seule chose qui compte. Mais elle ne devrait pas surprendre. Développer du logiciel revient à développer des modèles. Une bonne technique d'implémentation est souvent une bonne technique de modélisation ; elle peut être applicable, au-delà des systèmes logiciels, aux systèmes d'autres domaines, qu'ils soient naturels ou artificiels.

Plus que l'implémentation au sens strict du terme, le thème de ce chapitre est, donc, la modélisation : comment utiliser les structures des objets pour construire des descriptions opérationnelles, réalistes et utiles des systèmes de tous genres.

8.1 LES OBJETS

À tout moment de son exécution, un système OO aura créé un certain nombre d'objets. La structure à l'exécution est l'organisation de ces objets et de leurs relations. Explorons ses propriétés.

Qu'est-ce qu'un objet ?

La définition se trouve en page 170. Voir aussi la règle de l'objet, page 175.

Tout d'abord, nous devons rappeler ce que veut dire, dans cette étude, le mot "objet". Il n'y a rien de vague dans cette notion ; une définition technique précise a été donnée dans le chapitre précédent:

Définition : objet

Un objet est une instance à l'exécution d'une classe.

Un système logiciel qui contient une classe `C` peut, en différents points de son exécution, créer des instances de `C` (grâce aux opérations de création et de clonage, dont les détails apparaîtront plus tard dans ce chapitre) ; une telle instance est une structure de données construite selon le modèle défini par `C` ; par exemple, une instance de la classe `POINT` introduite dans le chapitre précédent est une structure de données formée de champs associés aux deux attributs `x` et `y` déclarés dans la classe. Les instances de toutes les classes possibles constituent l'ensemble des objets.

La définition ci-dessus est la définition officielle du logiciel orienté objet. Mais, "objet" a aussi un sens plus général, qui vient du langage de tous les jours. Tout système logiciel est associé à un système externe, qui peut contenir des "objets" : des points, des lignes, des angles, des surfaces et des solides dans un système graphique : des employés, des chèques de paye, des échelles de salaire dans un système de paye ; et ainsi de suite. Certains objets créés par le logiciel seront en correspondance directe avec de tels objets externes, comme dans un système de paye qui contient une classe `EMPLOYEE`, dont les instances à l'exécution sont les modèles informatiques des employés.

"Correspondance directe", page 48.

L'utilisation duale du mot "objet" a des conséquences bénéfiques, qui découlent de la puissance de la méthode orientée objet considérée comme outil de modélisation. Plus adaptée que les autres méthodes, la technologie objet met en valeur et facilite la partie modélisation du développement logiciel. Cela explique en partie l'impression de naturel qui s'en dégage, l'attraction qu'elle exerce sur tant de gens et ses succès initiaux — qui restent parmi les plus visibles — dans des domaines comme la simulation et les interfaces utilisateur. La méthode possède ici la propriété de *correspondance directe* qu'un chapitre précédent a présentée comme une exigence principale d'une bonne conception modulaire. Puisque les systèmes logiciels sont considérés comme des modèles directs ou indirects des systèmes réels, il n'est pas surprenant que certaines classes soient des modèles de types d'objets externes appartenant au domaine du problème, et donc que les objets logiciels (les instances de ces classes) soient eux-mêmes des modèles des objets externes correspondants.

Ne nous laissons pas trop emporter par le mot “objet”. Comme toujours en science et en technologie, il est un peu dangereux d’emprunter des mots au langage de tous les jours et de leur donner un sens technique. (La seule discipline qui semble avoir réussi dans cet art délicat est les mathématiques, qui s’accaparent sans vergogne des mots innocents comme “voisinage”, “variété” ou “cylindre”, et les utilisent avec des sens complètement inattendus — ce qui est peut-être la raison pour laquelle cela ne semble gêner personne.) Le terme “objet” est si surchargé de sens commun que, en dépit des avantages que nous venons de mentionner, son utilisation dans un sens logiciel technique a été la source d’une certaine confusion. En particulier :

- Comme nous l’avons signalé lors de la discussion sur la correspondance directe, toutes les classes ne correspondent pas à des types d’objets du domaine du problème. Les classes introduites lors de la conception et de l’implémentation n’ont pas d’équivalent immédiat dans le système modélisé. Ce sont souvent les plus importantes en pratique et les plus difficiles à trouver.
- Certains concepts du domaine du problème peuvent correspondre à des classes du logiciel (et à des objets lors de l’exécution du logiciel) tout en n’étant pas nécessairement considérés comme des objets dans le sens usuel du terme, si nous insistons pour avoir une vision concrète des objets. Les classes comme l’état *STATE* dans l’étude d’un système interactif fondé sur des masques de saisie ou *COMMAND* (que nous étudierons dans un prochain chapitre dans le cadre des mécanismes défaire-refaire) tombent dans cette catégorie.

Voir le chapitre 20 sur les systèmes utilisant des formulaires. À propos de la notion de commande, voir le chapitre 21.

Quand le mot “objet” est utilisé dans ce livre, le contexte indiquera clairement s’il s’agit du sens commun ou (plus fréquemment) du terme logiciel technique. Quand il nous faudra distinguer entre eux, nous parlerons d’*objets externes* et d’*objets logiciels*.

Forme de base

Un objet logiciel est un animal relativement simple une fois que vous savez de quelle classe il vient.

Soit `o` un objet. La définition de la page précédente indique qu’il est instance d’une certaine classe. Plus précisément, c’est une **instance directe** d’une seule classe, disons *C*.

> Du fait de l’héritage, `o` sera donc une instance, directe ou non, d’autres classes, les ancêtres de *C* ; mais cela fera l’objet d’un prochain chapitre et, dans l’étude présente, nous avons seulement besoin de la notion d’instance directe. Le mot “direct” sera omis quand il n’y a pas de risque de confusion

C est appelée la classe génératrice, ou simplement **générateur**, de `o`. *C* est un texte logiciel ; `o` est une structure de données à l’exécution, produite par l’un des mécanismes de création d’objets étudiés ci-dessous.

Parmi ses caractéristiques, *C* a un certain nombre d’attributs. Ces attributs déterminent entièrement la forme de l’objet : `o` est simplement une collection de composants, ou **champs**, un par attribut.

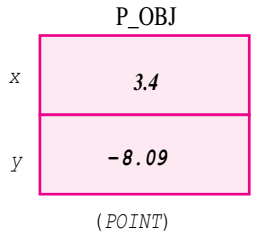
Considérons la classe *POINT* du chapitre précédent. Le texte de la classe était de la forme :

```
class POINT feature
        x, y: REAL
        ... Déclarations de routines ...
end
```

Pour le texte de la classe POINT, voir page 180.

Les routines ont été omises, et pour de bonnes raisons : la forme des objets correspondants (les instances directes de la classe) ne dépend que des attributs, quoique les *opérations* applicables aux objets dépendent des routines. Ici, la classe a deux attributs, *x* et *y*, tous deux de type *REAL* ;

ainsi une instance directe de *POINT* est un objet à deux champs contenant des valeurs de ce type, par exemple :



Voir "Conventions graphiques", page 267.

Notez les conventions utilisées ici et dans le reste de ce livre pour représenter un objet par un ensemble de champs, indiqués par des rectangles adjacents, qui contiennent les valeurs associées. En dessous de l'objet, le nom de la classe génératrice, ici *POINT*, apparaît entre parenthèses et en italique ; à côté de chaque champ, également en italique, apparaît le nom de l'attribut correspondant, ici *x* et *y*. Parfois, un nom en caractères romains (ici P_OBJ) apparaîtra au-dessus de l'objet ; il n'a pas d'équivalent dans le logiciel, mais identifie l'objet dans l'exposé.

Dans les diagrammes utilisés pour illustrer la structure d'un système orienté objet ou, plus souvent, d'une partie d'un tel système, les classes apparaissent sous forme d'ellipse. Cette convention, déjà utilisée dans les figures du chapitre précédent, évite toute confusion entre classe et objet.

Champs simples

Les deux attributs de la classe *POINT* sont de type *REAL*. En conséquence, chaque champ correspondant d'une instance directe de *POINT* contient une valeur réelle.

C'est un exemple d'un champ correspondant à un attribut ayant un "type de base". Bien que ces types soient formellement définis comme des classes, leurs instances prennent leur valeur dans des ensembles prédéfinis, implémentés efficacement sur les ordinateurs. Parmi ceux-ci, on trouve :

- *BOOLEAN*, qui n'a que deux instances, représentant les valeurs booléennes vrai et faux ;
- *CHARACTER*, dont les instances représentent des caractères ;
- *INTEGER*, dont les instances représentent des entiers ;
- *REAL* et *DOUBLE*, dont les instances représentent des nombres flottants en précision simple et double.

"CHAÎNES", 13.5, page 441.

Un autre type qui, pour l'instant, sera traité comme un type de base, bien que, comme nous le verrons plus tard, il rentre en fait dans une autre catégorie, est celui des chaînes *STRING*, dont les instances représentent des séquences finies de caractères.

Pour chaque type de base, il nous faut pouvoir indiquer les valeurs correspondantes dans les textes logiciels et sur les figures. Les conventions sont évidentes :

- Pour *BOOLEAN*, les deux instances sont écrites `True` et `False`.
- Pour désigner une instance de *CHARACTER*, écrivez un caractère entouré de simples guillemets, comme `'A'`.
- Pour désigner une instance de *STRING*, écrivez une séquence de caractères entre doubles guillemets, comme `"UNE CHAINE"`.
- Pour désigner une instance de *INTEGER*, écrivez un nombre en notation décimale classique avec un signe facultatif, comme `34`, `-675` et `+4`.

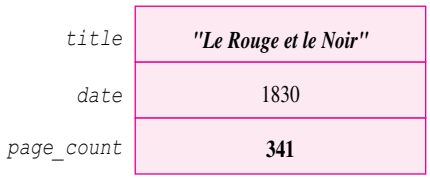
- Vous pouvez aussi écrire une instance de `REAL` ou `DOUBLE` en notation classique, comme `3.5` ou `-0.05`. Utilisez la lettre `e` pour introduire un exposant décimal, comme dans `-5.e-2` qui désigne la même valeur que dans l'exemple précédent.

Une notion simple de livre

Voici une classe ayant des types d'attributs appartenant à l'ensemble précédent :

```
class BOOK1 feature
        title: STRING
        date, page_count: INTEGER
end
```

Une instance typique de la classe `BOOK1` peut apparaître comme suit :



Un objet représentant un livre

Puisque, pour l'instant, nous ne nous intéressons qu'à la structure des objets, toutes les caractéristiques de cette classe et des prochains exemples seront des attributs — pas des routines.

Cela veut dire que nos objets sont, à ce stade, semblables aux enregistrements ou types structures des langages non orientés objet comme Pascal et C. Mais, contrairement à ces langages, nous ne pouvons pas, dans un bon langage OO, faire grand-chose avec une telle classe : du fait des mécanismes de rétention d'information, une classe client n'a aucun moyen d'affecter des valeurs aux champs de ces objets. En Pascal, ou en C avec une syntaxe légèrement différente, un type enregistrement de structure similaire permettrait à un client d'inclure la déclaration et l'instruction :

```
b1: BOOK1
...
b1.page_count := 355
```

Attention : pas autorisé dans la notation OO ! Pour étude uniquement.

qui, à l'exécution, affecterait la valeur 355 au champ `page_count` de l'objet attaché à `b1`. Avec les classes, cependant, nous ne fournirons pas ce type de service : laisser les clients changer les champs des objets comme ils le veulent seraient une infraction à la règle de rétention d'information, qui implique que l'auteur de chaque classe contrôle l'ensemble précis des opérations que les clients peuvent exécuter sur ses instances. Aucune affectation directe d'un champ n'est possible dans un contexte OO ; les clients effectueront des modifications de champs via les procédures de la classe. Plus tard dans ce chapitre, nous ajouterons à `BOOK1` une procédure qui donne aux clients un moyen d'obtenir l'effet de l'affectation ci-dessus, si l'auteur de la classe souhaite effectivement leur offrir de tels privilèges.

> Nous avons déjà vu que C++ et Java autorisent les affectations de la forme `b1.page_count := 355`. Mais cela ne fait que refléter les limites contre lesquelles butent les efforts tendant à intégrer la technologie objet dans un contexte C.

[Arnold 1996], *page 40*.

Comme l'écrivent eux-mêmes les concepteurs de Java dans leur livre sur ce langage : *"Un programmeur peut néanmoins toujours polluer un objet en modifiant un champ* [*public*]*, car le champ* [*peut être*] *modifié"* par des instructions d'affectation directe. Trop de langages nécessitent de tels avertissements de la forme "ne faites pas ceci". Plutôt que de spécifier un langage pour, ensuite, expliquer en long et en large comment ne pas l'utiliser, il est préférable de définir parallèlement la méthode et une notation qui lui convient.

Voir aussi "Si c'est baroque, réparez-le", page 651.

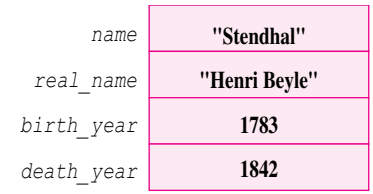
Dans un vrai développement OO, les classes sans routine, comme `BOOK1`, ont peu d'intérêt pratique (sauf en tant qu'ancêtres dans une hiérarchie d'héritage où les descendants hériteraient des attributs et fourniraient leurs propres routines ; ou pour représenter des objets externes auxquels l'environnement OO peut accéder mais qu'il ne peut modifier, par exemple les données d'un capteur dans un système temps réel). Mais, elles nous aideront à introduire les concepts de base ; puis nous y ajouterons les routines.

Écrivains

En utilisant les types mentionnés ci-dessus, nous pouvons aussi définir une classe `WRITER` qui décrit une notion simple d'auteur de livre :

```
class WRITER feature
        name, real_name: STRING
        birth_year, death_year: INTEGER
end
```

Un objet "écrivain"



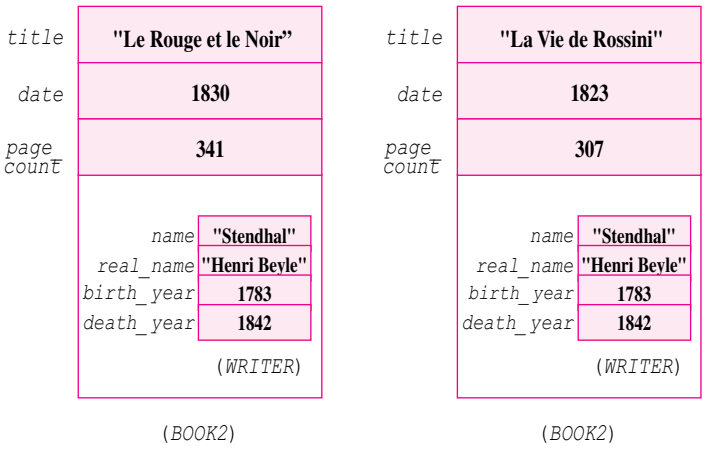
Références

Les objets dont les champs ont tous des types de base ne vont pas nous mener très loin. Nous avons besoin d'objets ayant des champs qui représentent d'autres objets. Par exemple, nous voudrons représenter la propriété selon laquelle un livre a un auteur — indiqué par une instance de la classe `WRITER`.

Nous pourrions envisager d'introduire une notion de sous-objet. Par exemple, nous pouvons penser à un objet livre, dans une nouvelle version `BOOK2` de la classe des livres, comme ayant un champ `author` qui est lui-même un objet, ainsi que le suggère la figure suivante.

Une telle notion de sous-objet est certes utile et nous verrons, plus loin dans ce chapitre, comment écrire les classes correspondantes.

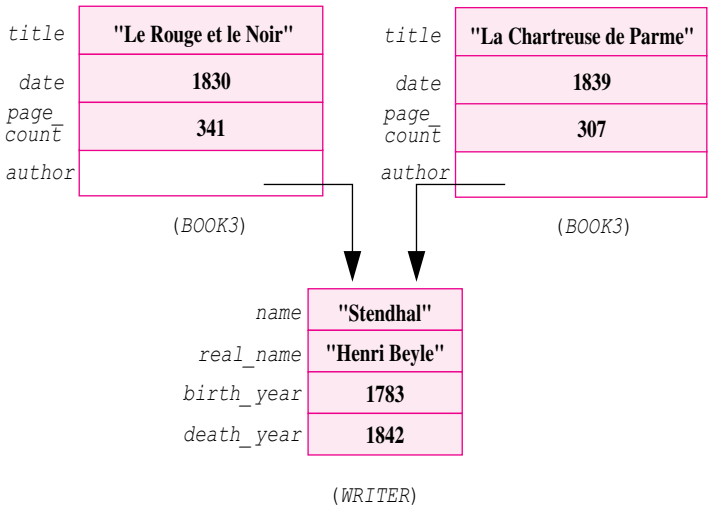
Mais ce n'est pas exactement ce dont nous avons besoin ici. L'exemple présente des livres ayant le même auteur ; cela nous a amené à dupliquer l'information d'auteur, qui apparaît maintenant comme deux objets, un dans chaque instance de `BOOK2`. Cette duplication n'est probablement pas acceptable :



Deux objets "livre" faisant référence au même objet "écrivain"

- Elle gaspille l'espace mémoire. D'autres exemples auraient rendu un tel gaspillage encore plus inacceptable : imaginez, par exemple, un ensemble d'objets représentant des individus, chacun ayant un sous-objet représentant le pays d'appartenance, avec un nombre important de personnes représentées, mais un nombre limité de pays.
- Plus grave encore, cette technique ne permet pas d'exprimer le **partage**. Indépendamment des choix de représentations, les champs `author` des deux objets font référence à la même instance de `WRITER` ; si vous mettez à jour l'objet `WRITER` (par exemple, pour enregistrer la mort de l'auteur), il faudrait que ce changement affecte tous les objets livres associés à un auteur donné.

Voici, donc, une meilleure représentation de la situation souhaitée, en supposant une troisième version de la classe des livres, `BOOK3` :



Deux objets "livre" ayant des sous-objets "écrivain".

Le champ `author` de chaque instance de `BOOK3` contient ce qu'on appelle une **référence** à un possible objet de type `WRITER`. Il n'est pas difficile de définir précisément cette notion :

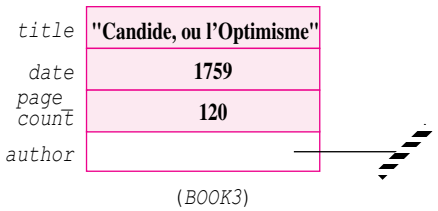
Définition : référence

Une référence est une valeur à l'exécution qui est **vide** ou **attachée**. Si elle est attachée, une référence identifie un objet unique. (On dit qu'elle est attachée à cet objet particulier.)

Dans la dernière figure, les champs de référence `author` dans les instances de `BOOK3` sont tous les deux attachés à l'instance de `WRITER`, comme le montrent les flèches qui sont conventionnellement utilisées dans de tels diagrammes pour représenter une référence attachée à un objet. La figure suivante montre une référence vide (peut-être pour indiquer un auteur inconnu), introduisant la représentation graphique des références vides :

Un objet avec un champ référence vide

(*"Candide" a été publié anonymement.*)



La définition des références ne mentionne aucune propriété d'implémentation. Une référence, si elle n'est pas vide, est un moyen d'identifier un objet ; un *nom* abstrait pour cet objet. Elle est semblable au numéro de Sécurité Sociale qui identifie de manière unique une personne, ou un code postal qui identifie une zone de courrier. Rien n'est ici spécifique à une implémentation ou à l'informatique.

Le concept de référence, bien sûr, a son pendant dans les implémentations informatiques. Dans la programmation au niveau machine, il est possible de manipuler des adresses ; de nombreux langages de programmation offrent une notion de pointeur. La notion de référence est plus abstraite. Bien qu'une référence puisse finalement être représentée par une adresse, cela n'est pas obligatoire ; et même quand la représentation d'une référence inclut une adresse, elle peut inclure d'autres informations.

Une autre propriété différencie également les références des adresses, quoiqu'elle soit aussi l'apanage des pointeurs dans les langages typés comme Pascal et Ada (pas C) : comme nous l'expliquerons ci-dessous, une référence, dans l'approche décrite ici, est typée. Cela veut dire qu'une référence donnée ne peut devenir attachée qu'à des objets ayant un type appartenant à un ensemble spécifique, déterminé par une déclaration dans le texte logiciel. Cette idée, à nouveau, trouve son pendant dans le monde non informatique : un numéro de Sécurité Sociale ne s'applique qu'aux personnes, tandis que des codes postaux ne s'appliquent qu'à des zones de courrier. (Il se peut qu'ils ressemblent à des nombres normaux, mais vous n'*additionneriez* pas des codes postaux.)

Identité des objets

La notion de référence introduit le concept d'identité d'objet. Chaque objet créé durant l'exécution d'un système orienté objet possède une identité unique, indépendante de la valeur de l'objet définie par ses champs. En particulier :

I1 • Deux objets ayant des identités différentes peuvent avoir des champs identiques.

I2 • Réciproquement, les champs d'un objet donné peuvent changer durant l'exécution d'un système ; mais cela n'affecte en rien l'identité de l'objet.

Ces observations indiquent qu'une phrase comme "*a* désigne le même objet que *b*" peut être ambiguë : parlons-nous d'objets ayant des identités différentes mais le même contenu (I1) ou des états d'un objet avant et après un changement effectué sur ses champs (I2) ? Nous utiliserons la seconde interprétation : un objet donné peut recevoir de nouvelles valeurs pour ses champs durant une exécution, tout en restant "le même objet". Chaque fois qu'une confusion est possible, il faudra être plus explicite. Dans le cas I1, nous pourrons parler d'objets égaux (mais distincts) ; l'égalité sera définie plus précisément ci-dessous.

> Un point de terminologie peut avoir attiré votre attention. Ce n'est pas une erreur de dire (comme dans la définition de I2) que les champs d'un objet peuvent changer. Le mot "champ", tel qu'il est défini ci-dessus, désigne l'une des valeurs qui constituent un objet, pas l'identificateur du champ correspondant, qui est le nom d'un des attributs de la classe génératrice de l'objet.
>
> Pour chaque attribut de la classe, par exemple *date* dans la classe *BOOK3*, l'objet a un champ, par exemple *1832* dans l'objet de la dernière figure. Durant l'exécution, les attributs ne changeront jamais, de façon que la composition d'un objet en champs reste la même ; mais les champs eux-mêmes pourront changer. Par exemple, une instance de *BOOK3* aura toujours quatre champs, correspondant aux attributs *title*, *date*, *page_count*, *author* ; ces champs — les quatre valeurs qui constituent un objet donné de type *BOOK3* — peuvent changer.

L'étude de la manière permettant de rendre des objets *persistants* nous amènera à explorer plus loin les propriétés de l'identité d'un objet.

"Identité d'objet", page 1018.

Déclarer des références

Regardons maintenant comment étendre la classe initiale des livres, *BOOK1*, qui n'avait que des attributs ayant un type de base, pour obtenir la nouvelle variante *BOOK3*, qui possède un attribut représentant des références aux auteurs potentiels. Voici le texte de la classe, ne montrant que les attributs ; la seule différence est la déclaration supplémentaire d'un attribut à la fin :

```
class BOOK3 feature
        title: STRING
        date, page_count: INTEGER
        author: WRITER                  -- C'est le nouvel attribut.
end
```

Le type utilisé pour déclarer *author* est simplement le nom de la classe correspondante : *WRITER*. Il s'agit d'une règle générale : chaque fois qu'une classe est déclarée sous la forme standard :

```
class C feature ... end
```

toute entité déclarée de type *C* grâce à une déclaration de la forme :

```
x: C
```

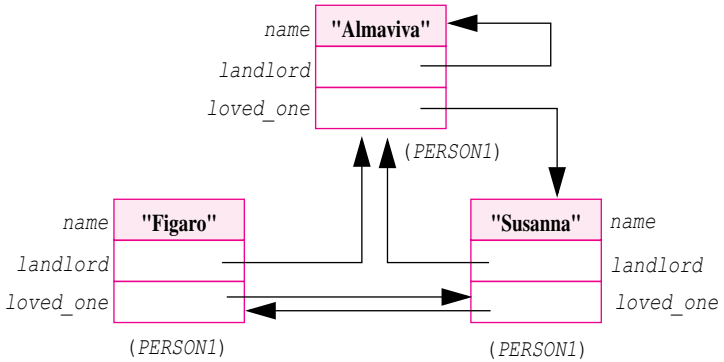
désigne des valeurs qui sont des **références** aux objets potentiels de type *C*. Cette convention se justifie par la souplesse d'utilisation des références, qui sont donc plus appropriées dans la grande majorité des cas. Vous trouverez un examen plus approfondi de cette règle (et des autres conventions possibles) dans la section de discussion de ce chapitre.

Voir page 268.

Autoréférence

Rien, dans l'exposé précédent, n'empêche qu'un objet O1 ne contienne une référence qui (à un moment de l'exécution du système) soit attachée à O1 lui-même. Cette sorte d'autoréférence peut être aussi indirecte. Dans la situation indiquée ci-dessous, l'objet ayant "Almaviva" dans son champ *name* est son propre propriétaire (cycle de référence directe) ; l'objet "Figaro" aime "Susanna" qui aime "Figaro" (cycle de référence indirecte) :

Autoréférence directe et indirecte



De tels cycles dans la structure dynamique ne peuvent exister que si la relation client entre classes présente également des cycles directs ou indirects. Dans l'exemple ci-dessus, la déclaration de classe est de la forme :

```
class PERSON1 feature
        name: STRING
        loved_one, landlord: PERSON1
end
```

exhibant un cycle direct (*PERSON1* est un client de *PERSON1*).

La propriété inverse n'est pas vraie : la présence d'un cycle dans la relation client n'implique pas que la structure à l'exécution aura des cycles. Par exemple, vous pouvez déclarer une classe :

```
class PERSON2 feature
        mother, father: PERSON2
end
```

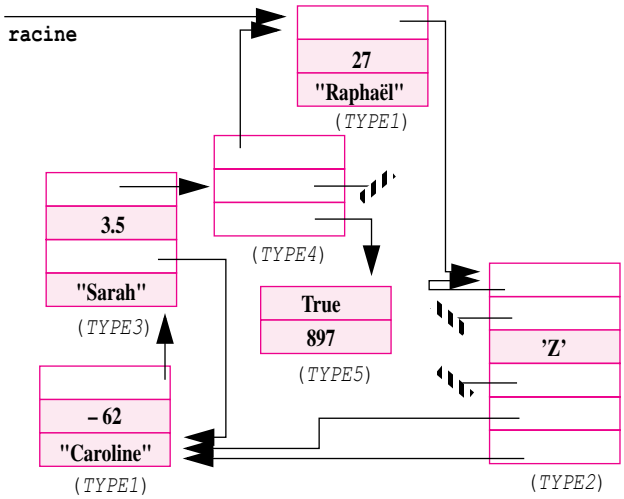
qui est cliente d'elle-même ; mais, si cela modélise les relations entre les personnes que suggèrent les noms des attributs (mère et père), il ne peut y avoir de cycle de référence dans la structure à l'exécution, car cela impliquerait qu'une certaine personne est son propre parent ou ancêtre indirect.

Un aperçu de la structure à l'exécution d'un objet

Une première vision de la structure à l'exécution d'un système orienté objet émerge de ce que nous avons vu jusqu'ici.

Le système est formé d'un certain nombre d'objets, ayant des champs divers. Certains champs sont des valeurs ayant des types de base (des champs entiers comme *27*, des champs caractères comme *'Z'* et ainsi de suite) ; d'autres sont des références, certaines vides, d'autres attachées à des objets. Chaque objet est une instance d'un certain type, toujours fondé sur une classe,

indiquée sous l'objet dans la figure. Certains types peuvent être représentés par une seule instance, mais, plus fréquemment, il y aura plusieurs instances d'un type donné ; ici, `TYPE1` a deux instances, les autres seulement une. Un objet peut avoir uniquement des champs références ; c'est le cas ici avec l'instance `TYPE4`, ou uniquement des champs de base, comme avec l'instance `TYPE5`. On peut trouver des autoréférences : directes, comme avec le champ supérieur de l'instance `TYPE2`, ou indirectes, comme le cycle de références dans le sens des aiguilles d'une montre qui débute et revient à l'instance `TYPE1` au sommet.



Une structure possible d'objets à l'exécution

Ce genre de structure peut sembler bien compliqué au début — une impression renforcée par la dernière figure, qui a pour objectif de montrer bon nombre de possibilités envisageables et qui ne prétend pas modéliser un quelconque système réel. L'expression "plat de spaghettis" vient rapidement à l'esprit.

Mais cette impression n'est pas justifiée. L'accent mis sur la simplicité s'applique aux textes logiciels et pas nécessairement à la structure des objets à l'exécution. Le texte d'un système logiciel implémente certaines relations (comme "est fils de (*child*)", "aime (*loves*)", "a comme propriétaire (*landlord*)") ; une structure particulière d'objets à l'exécution implémente ce que nous pouvons appeler une instance de ces relations — la manière dont les relations opèrent entre les membres d'un ensemble donné d'objets. Les relations modélisées par le logiciel peuvent être simples même si leurs instances, pour un ensemble particulier d'objets, sont complexes. Quelqu'un qui considère comme simple l'idée de base derrière la relation "aime" peut trouver particulièrement complexe l'instance de cette relation — la liste de qui aime qui — dans un groupe particulier de personnes.

Ainsi, il est souvent impossible d'empêcher que les structures des objets gagnent en importance (mettant en jeu un grand nombre d'objets) et en complexité (mettant en jeu de nombreuses références dans une structure tarabiscotée) durant l'exécution de nos systèmes OO. Un bon environnement de développement logiciel fournira des outils qui faciliteront l'exploration des structures d'objets lors des tests et de la mise au point.

Une telle complexité à l'exécution n'a pas à affecter la vision statique. Nous devrions essayer de conserver au logiciel lui-même — l'ensemble des classes et de leurs relations — la plus grande simplicité.

Le fait que des modèles simples puissent avoir des instances complexes est, en partie, une conséquence de la puissance des ordinateurs. Un petit texte logiciel peut décrire un calcul gigantesque ; un simple système OO peut, au moment de l'exécution, fournir des millions d'objets connectés par de nombreuses références. Un objectif primordial du génie logiciel est de maintenir la simplicité du logiciel, même quand ses instances ne le sont pas.

8.2 LES OBJETS COMME OUTILS DE MODÉLISATION

Nous pouvons utiliser les techniques introduites jusqu'à présent pour améliorer notre compréhension de la puissance de modélisation de la méthode. En particulier, il est important de clarifier deux aspects : d'une part, les divers mondes concernés par le développement logiciel et, d'autre part, la relation entre notre logiciel et la réalité externe.

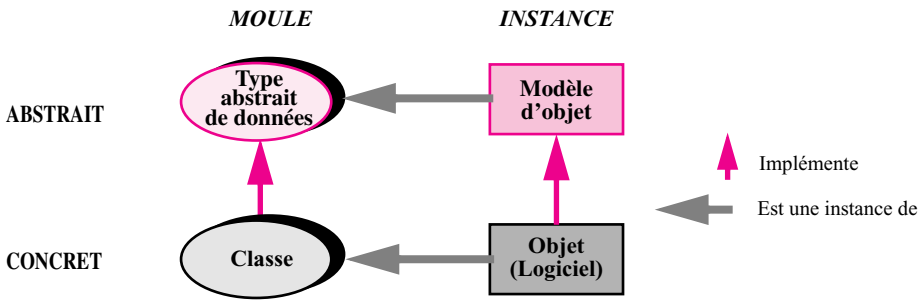
Les quatre mondes du développement logiciel

Des exposés précédents, il apparaît que, quand nous parlons de développement logiciel orienté objet, nous devrions différencier quatre mondes :

- le système modélisé, aussi appelé système externe (en opposition au système logiciel) et décrit par les types d'objets et les relations abstraites ;
- une instance particulière du système externe, faite d'objets entre lesquels des relations peuvent exister ;
- le système logiciel, fait de classes connectées par les relations de la méthode orientée objet (client et héritage) ;
- une structure d'objets, comme celle qui peut exister durant l'exécution du système logiciel, faite d'objets logiciels connectés par des références.

La figure suivante suggère les relations qui existent entre ces mondes :

Les moules et leurs instances



Il est important, à la fois au niveau logiciel (partie inférieure de la figure) et au niveau externe (la partie supérieure), de distinguer entre les notions générales (les classes et les relations abstraites, qui apparaissent à gauche) et leurs instances spécifiques (objets et instances de relation, à droite). Cela a déjà été mis en exergue lors du développement du chapitre précédent

consacrée aux rôles respectifs des classes et des objets. Cela s'applique également aux relations : nous devons distinguer entre la relation abstraite `loved_one` et l'ensemble des liens `loved_one` qui existent entre les éléments d'un ensemble donné d'objets.

Cette distinction n'est mise en évidence ni par les définitions mathématiques standard des relations ni, dans le domaine logiciel, par la théorie des bases de données relationnelles. En nous limitant aux relations binaires, une relation est définie, à la fois en mathématiques et dans les bases de données relationnelles, comme un ensemble de paires, toutes de la forme `<x, y>`, où chaque `x` est membre d'un ensemble donné `TX` et tout `y` est membre d'un ensemble donné `TY`. (Dans la terminologie logicielle : tous les `x` sont de type `TX` et tous les `y` sont de type `TY`.) Aussi appropriée que soit une telle définition d'un point de vue mathématique, elle n'est pas satisfaisante pour modéliser des systèmes, car elle ne permet pas de faire la distinction entre une relation abstraite et une de ses instances particulières. Dans la modélisation des systèmes, et même en mathématiques et bases de données relationnelles, la relation `loves` a ses propres propriétés générales et abstraites, tout à fait indépendantes de la mention de qui aime qui dans un groupe particulier de personnes à un moment donné.

> Cette réflexion sera poursuivie dans un prochain chapitre quand nous examinerons les *transformations* opérées sur les objets abstraits et concrets, et donnerons un nom aux flèches verticales de la figure précédente : la *fonction d'abstraction*.

"La fonction d'abstraction", page 365.

La réalité : un cousin doublement éloigné

Vous avez peut-être remarqué combien la présentation ci-dessus (et les précédentes sur des sujets voisins) s'efforce de ne faire aucune référence au "monde réel". Au contraire, l'expression utilisée ci-dessus pour faire référence à ce que le logiciel représente est simplement le "système modélisé".

Cette distinction n'est pas toujours faite. De nombreuses études sur la modélisation de l'information parlent de la "modélisation du monde réel", et des expressions similaires fleurissent dans les livres consacrés à l'analyse OO. Aussi nous devrions prendre un moment pour réfléchir à cette notion. Parler de la "réalité" derrière un système logiciel est trompeur pour au moins quatre raisons.

Tout d'abord, la réalité n'existe que dans les yeux du croyant. Sans être accusé de favoritisme exagéré envers sa profession, un ingénieur logiciel peut, de manière relativement justifiée, demander à ses clients en quoi *leur* système est plus réel que le sien. Considérez un programme qui effectue des calculs mathématiques — pour prouver la conjecture des quatre couleurs en théorie des graphes, pour intégrer des équations différentielles ou pour résoudre des problèmes géométriques sur une surface de Riemann à quatre dimensions. Faut-il que nous, développeurs logiciels, nous querellions avec nos amis (et clients) mathématiciens pour savoir lequel de ces artefacts est le plus réel : un morceau de logiciel écrit dans un langage de programmation ou un sous-espace complet de courbure négative ?

Deuxièmement, la notion de monde réel s'écroule dans le cas, pas si rare, de logiciels traitant de problèmes logiciels — des applications réflexives, comme on les appelle parfois. Prenez un compilateur C écrit en Pascal. Les objets réels qu'il traite sont des programmes C. Pourquoi devrions-nous considérer ces programmes comme plus réels que le compilateur lui-même ? La même observation s'applique aux autres systèmes qui traitent d'objets qui n'existent que dans un ordinateur : un éditeur, un outil CASE ou même un système de traitement de documents

(puisque les documents qu'il manipule sont des objets informatiques, les versions imprimées n'en étant que la forme finale).

*Voir aussi "DISCUSSION", 20.6, page 674 sur les dangers de rester **trop** proche de la réalité.*

La troisième raison est une généralisation de la seconde. Au début de l'informatique, il pouvait paraître légitime de considérer que les systèmes logiciels étaient superposés à une réalité existante et indépendante. Mais, de nos jours, les ordinateurs et leurs logiciels font de plus en plus partie de cette réalité. De même qu'un physicien quantique se trouve incapable de séparer la mesure de l'objet mesuré, nous pouvons rarement considérer que "le monde réel" et "le logiciel" sont des entités indépendantes. Le domaine des SGI (systèmes de gestion de l'information, c'est-à-dire le traitement des données de l'entreprise) en fournit une illustration des plus marquantes : bien que les premières applications SGI aient pu être introduites, il y a quelques décennies, par les entreprises, pour automatiser, grâce aux ordinateurs et à leurs logiciels associés, des procédures existantes, la situation actuelle est radicalement différente, car de nombreuses procédures existantes mettent déjà en jeu des ordinateurs et leurs logiciels. Décrire les opérations d'une banque moderne revient à décrire des mécanismes où le logiciel joue un rôle fondamental. Il en est de même dans la plupart des domaines d'application ; nombreuses sont les activités des physiciens et autres scientifiques de la nature, par exemple, qui s'appuyent sur les ordinateurs et le logiciel, vus non pas comme des outils auxiliaires mais comme faisant partie intégrante du processus opérationnel. On peut méditer, à ce stade, sur l'expression de "réalité virtuelle", et ce qu'elle implique : ce que produit le logiciel n'est pas moins réel que ce qui vient du monde extérieur. Dans tous ces cas, le logiciel n'est pas séparé de la réalité, et tout se passe comme si nous avions une boucle de rétroaction par laquelle le logiciel injecte des données nouvelles et importantes dans le modèle.

La dernière raison est encore plus fondamentale. Un système logiciel n'est pas un modèle de la réalité ; c'est, au mieux, un modèle d'un modèle d'une partie d'une réalité. Un système de contrôle d'un patient d'hôpital n'est pas un modèle de l'hôpital, mais l'implémentation de la vision de quelqu'un concernant certains aspects de la gestion de l'hôpital — un *modèle* d'un *modèle* d'un *sous-ensemble* de la réalité de l'hôpital. Un programme d'astronomie n'est pas un modèle de l'univers ; c'est un modèle logiciel du modèle de quelqu'un concernant certaines propriétés d'une certaine partie de l'univers. Un système d'information financière n'est pas un modèle de la bourse ; c'est une transposition logicielle d'un modèle conçu par une certaine compagnie pour décrire les aspects de la bourse qui sont pertinents par rapport aux objectifs de la compagnie.

Voir "AU-DELÀ DU LOGICIEL", 6.6, page 151.

Le thème général de la méthode orientée objet, les types abstraits de données, aident à comprendre pourquoi nous n'avons pas besoin de nous bercer de l'idée flatteuse mais illusoire que nous traitons avec le monde réel. La première étape de l'orientation objet, telle qu'elle est exprimée par la théorie des ADT, est de se détacher de la réalité en faveur de quelque chose de moins grandiose mais de plus maniable : un ensemble d'abstractions caractérisées par les opérations offertes aux clients, et leurs propriétés formelles. (Ceci nous a donné le slogan du modélisateur ADT — ne me dites pas ce que vous êtes, mais ce que vous avez.) N'allons jamais prétendre que ce sont les seules opérations et propriétés possibles : nous choisissons celles qui correspondent à nos objectifs du moment et rejetons les autres. *Modéliser, c'est éliminer.*

Pour un système logiciel, la réalité qu'il prend en compte est, au mieux, un cousin issu de germain.

8.3 MANIPULER LES OBJETS ET LES RÉFÉRENCES

Revenons à des questions plus terre-à-terre et regardons comment nos systèmes logiciels vont manipuler les objets pour créer et utiliser des structures de données flexibles.

Création dynamique et rattachement

Ce que la description de la structure des objets à l'exécution n'a pas encore montré, c'est la nature très dynamique d'un vrai modèle orienté objet. Contrairement aux politiques statique et à base de pile de la gestion des objets, illustrées au niveau des langages de programmation par, respectivement, Fortran et Pascal, la politique d'un environnement OO consiste à laisser les systèmes créer les objets au fur et à mesure des besoins au cours de l'exécution, selon un schéma qu'il est habituellement impossible de prévoir par un simple examen statique du texte logiciel.

À partir d'un état initial dans lequel (comme nous l'avons décrit dans le chapitre précédent) un seul objet a été créé — l'objet racine —, un système exécutera de manière répétitive sur la structure des objets des opérations telles que créer un nouvel objet, attacher une référence précédemment vide à un objet, créer une référence vide ou rattacher à un objet différent une référence précédemment attachée. La nature dynamique et imprévisible de ces opérations explique, en partie, la flexibilité de l'approche et sa capacité à prendre en compte les structures de données dynamiques qui sont nécessaires si nous voulons utiliser des algorithmes sophistiqués et modéliser les propriétés volatiles des systèmes externes.

Les prochaines sections explorent les mécanismes nécessaires pour créer des objets et manipuler leurs champs, en particulier les références.

L'instruction de création

Regardons comment nous allons créer une instance d'une classe comme *BOOK3*. Cela n'est possible que dans une routine d'une classe qui est client de *BOOK3*, comme :

```
class QUOTATION feature
        source: BOOK3
        page: INTEGER
        make_book is
                        -- Créer un objet BOOK3 et lui attacher source.
                do
                        ... Voir ci-dessous ...
                end
end
```

qui peut servir à décrire une citation d'un livre, telle qu'elle apparaît dans une autre publication, identifiée par deux champs : une référence au livre cité et le numéro de la page qui cite le livre.

Le mécanisme (bientôt expliqué) qui permet de créer une instance de type *QUOTATION* initialisera aussi par défaut tous ses champs. Une part importante de la règle d'initialisation par défaut est que tout champ référence, comme celui associé à l'attribut *source*, sera initialisé avec une référence vide. En d'autres termes, créer un objet de type *QUOTATION* ne crée pas en lui-même un objet de type *BOOK3*.

La règle générale est que, à défaut d'une intervention, une référence reste vide. Pour changer cela, vous pouvez créer un nouvel objet avec l'instruction de création. Cela peut être fait par la procédure *make_book*, qui devrait alors se lire comme suit :

```
make_book is
            -- Créer un objet BOOK3 et l'attacher à source.
      do
            !! source
      end
```

Ceci illustre la forme la plus simple de l'instruction de création : `!! x`, où *x* est un attribut de la classe englobante ou (comme nous le verrons plus tard) une entité locale de la routine englobante. Nous verrons quelques extensions de cette notation de base par la suite.

Le symbole `!` est habituellement lu "bang", et, donc, `!!` est "bang bang". L'entité *x* nommée dans l'instruction (*source* dans l'exemple ci-dessus) est appelée la **cible** de l'instruction de création.

Cette forme de l'instruction de création est appelée "instruction de création de base". (Une autre forme, mettant en jeu un appel à une procédure de la classe, apparaîtra sous peu.) Voici l'effet précis d'une instruction de création de base :

Effet d'une instruction de création de base

L'effet d'une instruction de création de la forme `!! x`, où le type de la cible *x* est un type référence fondé sur une classe *C*, est d'exécuter les trois étapes suivantes :

C1 • créer une nouvelle instance de *C* (faite d'une collection de champs, un pour chaque attribut de *C*). Soit *OC* cette nouvelle instance,

C2 • initialiser chaque champ de *OC* selon les valeurs standards par défaut,

C3 • attacher la valeur de *x* (une référence) à *OC*.

Les "valeurs standard par défaut" mentionnées dans C2 apparaissent dans la boîte suivante.

L'étape C1 créera une instance de *C*. L'étape C2 positionnera les valeurs de chaque champ à une valeur prédéterminée, qui dépend du type de l'attribut correspondant. En voici les valeurs :

Valeurs d'initialisation par défaut

Pour une référence, la valeur par défaut est une référence vide.

Pour un *BOOLEAN*, la valeur par défaut est *False*.

Pour un *CHARACTER*, la valeur par défaut est le caractère nul.

Pour un nombre (de type *INTEGER*, *REAL* ou *DOUBLE*), la valeur par défaut est zéro (c'est-à-dire la valeur zéro de type approprié).

Ainsi, pour une cible *source* de type *BOOK3*, dont la déclaration de classe ci-dessus était :

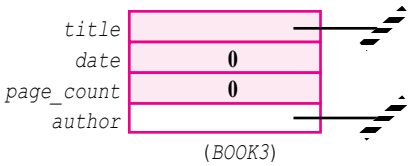
```
class BOOK3 feature
      title: STRING
      date, page_count: INTEGER
      author: WRITER
end
```

"CHAÎNES", 13.5, page 441.

l'instruction de création `!! source`, exécutée comme partie d'un appel à la procédure *make_book* de la classe *QUOTATION*, donnera un objet de la forme suivante (voir figure suivante).

Les champs entiers ont été initialisés à zéro. Le champ référence pour *author* a été initialisé à une référence vide. Le champ pour *title*, une chaîne *STRING*, indique également une référence

vide. Cela découle du fait que le type `STRING` (dont nous n'avons pas parlé dans les règles d'initialisation ci-dessus) est, en fait, également un type référence, bien que, comme nous l'avons noté, nous puissions souvent, en pratique, le traiter comme un type de base.



Un objet nouvellement créé et initialisé

La vue globale

Il est important de ne pas perdre le fil de l'ordre dans lequel les événements se produisent. Pour que l'instance ci-dessus de `BOOK3` soit créée, les deux événements suivants ont dû avoir lieu :

B1 • Une instance de `QUOTATION` est créée. Soit Q_OBJ cette instance et soit `a` une entité dont la valeur est une référence attachée à Q_OBJ.

B2 • Quelque temps après l'étape B1, un appel de la forme `a.make_book` exécute la procédure `make_book` en ayant Q_OBJ comme cible.

Il est, bien sûr, légitime de se demander comment nous allons arriver à l'étape B1 — comment Q_OBJ lui-même sera créé. Cela ne fait que repousser le problème. Mais, maintenant, vous connaissez la réponse à cette question : tout revient au big-bang. Pour exécuter un système, vous devez fournir une classe racine et le nom d'une procédure de cette classe — la procédure de création. Au début de l'exécution, on vous fournit automatiquement un objet, l'objet racine, une instance de la classe racine. L'objet racine est le seul qu'il n'est pas nécessaire de créer dans le texte logiciel lui-même ; il vient de l'extérieur, comme un *objectus ex machina*. En débutant avec cet objet providentiel, le logiciel peut maintenant créer d'autres objets de manière normale, via les routines qui exécutent les instructions de création. La première routine à être exécutée est la procédure de création, appliquée automatiquement à l'objet racine ; dans tous les cas non triviaux, elle contiendra au moins une instruction de création, de façon à démarrer ce que nous comparions, dans le chapitre précédent, à un feu d'artifice géant : le processus de production de tous les nouveaux objets qu'une exécution particulière demandera.

Voir "REGROUPER LE TOUT", 7.9, page 197.

Pourquoi une création explicite ?

La création d'objet est explicite. Déclarer une entité comme :

```
b: BOOK3
```

ne déclenche pas la création d'un objet à l'exécution : la création n'aura lieu que quand un élément du système exécutera une opération :

```
!! b
```

Pourquoi ? Est-ce que la déclaration de `b` ne devrait pas suffire si nous avons besoin d'un objet à l'exécution ? À quoi cela sert-il de déclarer une entité si nous ne créons pas un objet ?

Réflexion faite, la distinction entre déclaration et création est la seule solution raisonnable.

Le premier argument est par *reductio ad absurdum*. Supposons que nous démarrions le traitement de la déclaration de `b` en créant immédiatement l'objet livre correspondant. Mais cet objet est une instance de la classe `BOOK3`, qui a un attribut `author`, lui-même de type référence `WRITER`, de sorte que le champ `author` est une référence, pour laquelle nous devons créer un objet sur le champ. Maintenant, cet objet a des champs références (rappelez-vous que `STRING` est, en fait, un type référence) qui devront subir le même traitement : nous sommes sur le point de démarrer un long processus de création récursive d'objets avant même d'avoir débuté un quelconque traitement utile !

Cet argument serait encore plus évident avec une classe autoréférente, comme `PERSON1` que nous avons vue ci-dessus :

```
class PERSON1 feature
      name: STRING
      loved_one, landlord: PERSON1
end
```

Traiter chaque déclaration comme si elle renvoyait un objet voudrait dire que chaque création d'une instance de `PERSON1` causerait la création de deux objets supplémentaires (correspondant à `loved_one` et `landlord`), déclenchant une boucle infinie. Cependant, nous avons vu que de telles définitions autoréférentes, directes, comme ici, ou indirectes, sont fréquentes et nécessaires.

Un autre argument découle simplement d'un thème qui surgit tout au long de ce chapitre : l'utilisation de la technologie objet comme puissante technique de modélisation. Si chaque champ référence était initialisé par un objet nouvellement créé, nous n'aurions pas de place pour les références vides ou des références multiples attachées à un objet unique. Ces deux notions sont nécessaires pour modéliser de manière réaliste des systèmes pratiques :

- Dans certains cas, le modèle peut imposer qu'une certaine référence soit laissée non attachée à un objet. Nous avons utilisé cette technique quand nous avons laissé vide le champ `author` pour indiquer qu'un livre est d'un auteur inconnu.

Voir la figure page 228.

- Dans d'autres cas, des références devraient être attachées, à nouveau pour des raisons conceptuelles venant du modèle, au même objet. Dans l'exemple d'autoréférence, nous avons vu que les champs `loved_one` des deux instances `PERSON1` sont attachés au même objet. Créer un objet pour chaque champ au moment de la création n'aurait, dans ce cas, pas de sens ; il nous faut, à la place d'une instance de création, une opération d'affectation (étudiée plus tard dans ce chapitre) qui attache une référence à un objet déjà existant. Cette observation s'applique encore plus clairement aux champs autoréférents du même exemple (champ `landlord` de l'objet en haut).

Le mécanisme de gestion d'objets n'attache jamais implicitement de références. Il crée des objets via des instructions de création (ou des opérations `clone`, voir ci-dessous, qui sont également explicites), en initialisant leurs champs références avec des références vides ; ce n'est qu'avec des instructions explicites que ces champs, à leur tour, deviendront attachés à des objets.

"Création polymorphe", page 464.

> Dans l'étude sur l'héritage, nous verrons qu'une instruction de création peut utiliser la syntaxe `! T ! x` pour créer un objet dont le type `T` est un descendant du type déclaré pour `x`.

8.4 PROCÉDURES DE CRÉATION

Toutes les instructions de création que nous avons vues jusqu'à présent utilisaient des initialisations par défaut. Dans certains cas, il se peut que vous n'appréciiez pas les

initialisations définies dans le langage, désirant plutôt fournir des informations spécifiques pour initialiser l'objet créé. Les procédures de création répondent à ce besoin.

Outrepasser les initialisations par défaut

Pour utiliser une initialisation autre que celle effectuée par défaut, ajoutez à la classe une procédure de création, ou plusieurs. Une procédure de création est une procédure de la classe qui est répertoriée, dans une clause débutant avec le mot-clé **creation**, au début de la classe, avant la première clause de caractéristique. Le schéma est le suivant :

```
indexing
        ...
class C creation
        p1, p2, ...
feature
        ... Déclarations de caractéristiques, y compris celles des procédures p1, p2, ...
end
```

Une suggestion de style : le nom recommandé pour les procédures de création, dans les cas simples, est *make*, pour une classe qui n'a qu'une procédure de création ; pour une classe ayant plusieurs procédures de création, il est généralement conseillé de leur donner un nom commençant par *make_* et se poursuivant avec un mot qualificatif, comme dans l'exemple *POINT* qui suit.

"CHOISIR LES BONS NOMS", 26.2, page 849.

L'instruction de création correspondante n'est plus simplement `!! x`, mais est de la forme :

```
!! x.p (...)
```

où *p* est l'une des procédures de création répertoriées dans la clause **creation**, et `(...)` une liste d'arguments réels valides pour *p*. L'effet d'une telle instruction est de créer l'objet en utilisant les valeurs par défaut de la forme précédente, et d'appliquer *p*, avec les arguments fournis, au résultat. L'instruction est appelée un **appel de création** ; c'est la combinaison d'une instruction de création et d'un appel de procédure.

Nous pouvons, par exemple, ajouter des procédures de création à la classe *POINT* pour permettre aux clients de spécifier des coordonnées initiales, cartésiennes ou polaires, au moment de la création d'un objet. Nous aurons deux procédures de création, *make_cartesian* et *make_polar*. Voici le schéma :

Version originale de POINT dans "La classe", page 180.

```
class POINT1 creation
        make_cartesian, make_polar
feature
        ... Les caractéristiques étudiées dans la version précédente de la classe :
            x, y, ro, theta, translate, scale, ...
feature {NONE} -- Voir les explications ci-dessous à propos du statut d'exportation.
        make_cartesian (a, b: REAL) is
                        -- Initialiser un point avec les coordonnées cartésiennes a et b.
                do
                        x := a; y := b
                end
        make_polar (r, t: REAL) is
                        -- Initialiser un point avec les coordonnées polaires r et t.
                do
                        x := r * cos (t); y := r * sin (t)
                end
end -- class POINT1
```

Avec ce texte de classe, un client créera un point grâce à des instructions comme :

```
!! my_point.make_cartesian (0, 1)
!! my_point.make_polar (1, Pi/2)
```

ayant toutes deux le même effet, si `Pi` a la valeur suggérée par son nom.

Voici la règle définissant l'effet de ces appels de création. Les trois premières étapes sont les mêmes que pour la forme de base vue précédemment :

Effet d'un appel de création

Un appel de création de la forme `!! x.p (...)`, où le type de la cible `x` est un type référence fondé sur une classe `C`, `p` est une procédure de création de la classe `C`, et `(...)` représente une liste valide d'arguments réels pour cette procédure, si besoin est, a pour effet d'exécuter les quatre étapes suivantes :

C1 • créer une nouvelle instance de `C` (faite d'un ensemble de champs, un pour chaque attribut de `C`). Soit `OC` la nouvelle instance ;

C2 • initialiser chaque champ de `OC` avec les valeurs par défaut standards ;

C3 • attacher la valeur de `x` (une référence) à `OC` ;

C4 • appeler la procédure `p`, avec les arguments donnés, sur `OC`.

La nouvelle étape

Le statut d'exportation des procédures de création

À propos de la construction { NONE} , voir "Style pour déclarer des caractéristiques secrètes", page 196.

Dans `POINT1`, les deux procédures de création ont été déclarées dans une clause de caractéristiques qui débute par **`feature`** `{ NONE}` . Cela indique qu'elles sont secrètes, mais seulement pour les appels normaux, pas pour les appels de création. Ainsi, les deux appels de création de l'exemple que nous venons de voir sont valides ; les appels normaux de la forme `my_point.make_cartesian (0, 1)` ou `my_point.make_polar (1, Pi/2)` sont invalides puisque les caractéristiques n'ont pas été rendues accessibles aux clients.

La décision de rendre les deux procédures secrètes montre que nous ne voulons pas que les clients, une fois qu'un objet existe, puissent positionner ses coordonnées directement, bien qu'ils puissent le faire indirectement via les autres procédures de la classe comme `translate` et `scale`. Bien sûr, ce n'est qu'une politique parmi d'autres ; vous pouvez très bien décider d'exporter `make_cartesian` et `make_polar` tout en les considérant comme des procédures de création.

Il est possible de donner à une procédure un statut sélectif de création en introduisant un ensemble de classes entre parenthèses dans sa clause `creation`, comme dans :

```
class C creation {A, B, ...}
        p1, p2,
...
```

bien que cela soit moins fréquent que de limiter le statut d'exportation d'une caractéristique via la syntaxe similaire **`feature`** `{ A, B, ...}` ou **`feature`** `{ NONE}` . Dans tous les cas, rappelez-vous que le statut de création d'une procédure est indépendant de son statut d'exportation d'appel.

Règles concernant les procédures de création

Les deux formes d'instructions de création, la forme de base `!! x` et l'appel de création `!! x.p (...)`, sont mutuellement exclusives. Dès qu'une classe a une clause **`creation`**, seul l'appel de création est permis ; la forme de base sera considérée comme invalide et sera rejetée par le compilateur.

Cette convention peut paraître étrange à première vue, mais elle est justifiée par des considérations de cohérence d'objets. Un objet n'est pas seulement une collection de champs ; c'est surtout l'implémentation d'un type abstrait de données, ce qui peut imposer des contraintes de cohérence aux champs. En voici un exemple typique. Supposez qu'un objet représente une personne, avec un champ pour la date de naissance et un autre pour l'âge. Vous ne pouvez pas indépendamment positionner ces deux champs à des valeurs arbitraires, mais vous devez assurer une contrainte de cohérence : la somme du champ d'âge et de l'année de naissance doit être égale à l'année courante ou la précédente. (Dans un prochain chapitre, nous apprendrons à exprimer de telles contraintes, qui reflètent souvent des axiomes de l'ADT sous-jacent, par des **invariants de classes**.) Une instruction de création doit *toujours* renvoyer un objet cohérent. La forme de base de l'instruction de création — `!! x`, sans appel — n'est acceptable que si l'on obtient un objet cohérent en positionnant tous les champs avec leurs valeurs par défaut. Si cela n'est pas le cas, il vous faudra introduire des procédures de création et vous devrez interdire la forme de base de l'instruction de création.

Voir "INVARIANTS DE CLASSE", 11.8, page 354, en particulier "Le rôle des procédures de création", page 361.

Dans certains cas rares, il se peut que vous vouliez accepter les initialisations par défaut (puisqu'elles vérifient les invariants de classe) tout en définissant au moins une autre procédure de création. La technique à appliquer dans ce cas consiste à mettre *nothing* parmi les procédures de création. La caractéristique *nothing* est une procédure sans arguments, héritée de la classe universelle *ANY*, qui a un corps vide (la déclaration de la caractéristique est simplement : *nothing* **is do end**) de manière à faire exactement ce que son nom indique.Vous pouvez alors écrire :

```
class C creation
        nothing, some_creation_procedure, some_other_creation_procedure...
feature
        ...
```

Bien que la forme `!! x` soit toujours invalide dans ce cas, les clients peuvent obtenir l'effet escompté en écrivant l'instruction `!! x.nothing`.

Enfin, cas spécial, notez que la règle sur les instructions de création nous donne un moyen de définir une classe qu'*aucun client* n'aura le droit d'instancier. Une déclaration de la forme :

```
class C creation
        -- Il n'y a rien ici !
feature
        ... Reste du texte de la classe ...
end
```

possède une clause de création — une clause vide. La règle ci-dessus indique que, s'il y a une clause **creation**, les seules instructions de création autorisées sont les appels de création qui utilisent une procédure de création ; ici, puisqu'il n'y a pas de procédure de création, aucun appel de création n'est permis.

Interdire l'instantiation de classes est de peu d'intérêt si nous nous limitons aux mécanismes orientés objet que nous avons vus jusqu'à présent. Mais, quand nous aborderons l'héritage, ce petit truc pourra s'avérer pratique si nous voulons spécifier qu'une certaine classe ne devrait être utilisée que comme ancêtre d'autres classes, jamais directement pour créer des objets.

Voir "Que faire avec les classes retardées", page 472 et l'exercice E14.5, page 502.

> Une autre manière d'obtenir ceci est de *retarder* la classe, mais une classe retardée doit avoir au moins une caractéristique retardée, et nous n'aurons pas toujours besoin d'une telle caractéristique.

Création multiple et surcharge

Avant d'y revenir dans la section de discussion, il est intéressant de comparer le mécanisme de procédures multiples de création avec l'approche de C++/Java. Le besoin est le même : fournir différentes manières d'initialiser un objet au moment de la création. C++ et Java utilisent, cependant, une technique différente, la surcharge de nom.

Dans ces langages, toutes les procédures de création d'une classe (ses "constructeurs") ont le même nom, qui est, de fait, le nom de la classe ; si une classe *POINT* contient un constructeur ayant deux arguments réels correspondant à *make_cartesian*, l'expression **new** *POINT (0, 1)* créera une nouvelle instance. Pour différencier deux constructeurs, le langage s'appuie sur les signatures (le type des arguments).

Voir "Surcharge syntaxique", page 96.

Le problème, bien évidemment, comme nous l'avons vu lors de l'étude de la surcharge, est que la signature des arguments n'est pas un critère approprié : si nous voulons également fournir un constructeur équivalent à *make_polar*, nous sommes coincés, car les arguments seraient les mêmes, deux nombres réels. C'est le problème général que présente la surcharge : utiliser le même nom pour des opérations différentes, créant ainsi une ambiguïté potentielle — aggravée ici par l'utilisation de ce nom comme nom de classe et aussi comme nom de procédure.

La technique développée précédemment semble préférable à tous égards : limiter l'intervention au minimum (pas de procédure de création), si l'initialisation par défaut suffit ; empêcher la création, si cela est souhaité, grâce à une clause **creation** vide ; fournir plusieurs formes de création ; définir autant de procédures de création qu'il est nécessaire ; ne pas introduire de confusion entre les noms des classes et les noms des caractéristiques ; permettre que l'effet de chaque opération soit clairement visible à partir de son nom, comme dans *make_polar*.

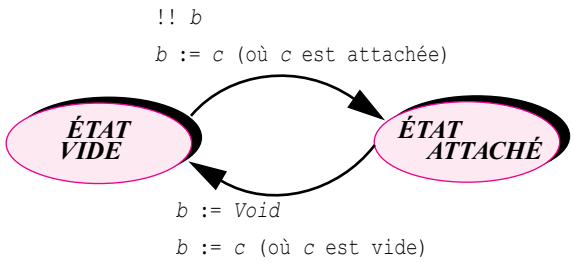
8.5 APPROFONDIR LES RÉFÉRENCES

Le modèle à l'exécution fait jouer un rôle important aux références. Examinons certaines de leurs propriétés, en particulier la notion de référence vide, ainsi que certaines interrogations qu'elles soulèvent.

Etats d'une référence

Une référence peut se trouver dans l'un des deux états suivants : vide et attachée. Nous avons vu qu'une référence est toujours initialement vide et qu'elle peut devenir attachée grâce à une création. Voici une figure plus complète.

Les états possibles d'une référence



Outre lors d'une création, une référence peut changer d'état grâce à une affectation, comme nous le verrons bientôt. Pour l'instant, assurez-vous que vous comprenez la différence entre les trois notions — objet, référence et entité — qui apparaissent tout au long de ce chapitre :

- Un "objet" est une notion à l'exécution ; tout objet est une instance d'une certaine classe, est créé au moment de l'exécution et est formé d'un certain nombre de champs.
- Une "référence" est aussi une notion à l'exécution : une référence est une valeur qui est soit vide, soit attachée à un objet. Nous avons vu une définition précise d'"attaché" : une référence est attachée à un objet si elle identifie cet objet de manière non ambiguë.
- A contrario, une "entité" est une notion statique — c'est-à-dire qui s'applique au texte du logiciel. Une entité est un identificateur apparaissant dans le texte d'une classe et représentant une valeur à l'exécution ou un ensemble de valeurs successives à l'exécution. (Les lecteurs habitués aux formes traditionnelles de développement logiciel peuvent considérer que la notion d'entité correspond aux variables, constantes symboliques, arguments des routines et résultats de fonction.)

Définition complète d'"entité" : page 216.

Si `b` est une entité de type référence, sa valeur à l'exécution est une référence qui peut être attachée à un objet O. Par abus de langage, nous pouvons dire que `b` elle-même est attachée à O.

Références vides et appels

Dans la plupart des cas, nous nous attendons qu'une référence soit attachée à un objet, mais les règles permettent également qu'une référence soit vide. Les références vides jouent un rôle important — ne serait-ce qu'en nous embêtant — dans le modèle de calcul orienté objet. Comme nous l'avons longuement expliqué dans le chapitre précédent, l'opération fondamentale de ce modèle est l'appel de caractéristique : appliquer à une instance d'une classe une caractéristique de cette classe. Cela s'écrit :

```
some_entity.some_feature (arg1, ...)
```

où `some_entity` est attaché à l'objet cible désiré. Pour que l'appel réussisse, `some_entity` doit, de fait, être attaché à un objet. Si `some_entity` a un type référence et une valeur vide au moment de l'appel, cet appel ne pourra pas être effectué, puisque `some_feature` a besoin d'un objet cible.

Pour être correct, un système orienté objet ne doit jamais essayer d'appeler, au moment de l'exécution, une caractéristique dont la cible est vide. L'effet en sera une **exception** ; la notion d'exception et la manière permettant de récupérer cette exception seront étudiées dans un prochain chapitre.

Voir le chapitre 12, en particulier "Sources d'exceptions", page 400.

Il serait préférable de laisser au compilateur le soin de vérifier le texte d'un système pour garantir qu'aucun événement de ce genre ne se produira à l'exécution, de la même manière qu'il peut vérifier l'absence d'incompatibilités de type en imposant des règles de typage. Malheureusement, un tel objectif général est, à l'heure actuelle, au-delà des possibilités des compilateurs (sauf à imposer des restrictions inacceptables au langage). Il appartient donc aux développeurs logiciels de s'assurer que l'exécution n'essaiera jamais d'appeler une caractéristique sur une cible vide. Il y a, bien sûr, un moyen simple de le faire : écrire toujours `x.f(...)` sous la forme :

```
if "x is not void" then -- x n'est pas vide
        x.f (...)
else
        ...
end
```

Le test `"x is not void"` peut être simplement écrit `x /= Void`. Voir ci-dessous.

mais c'est une exigence trop lourde pour être universellement acceptable. Parfois (comme quand un appel `x...f` suit immédiatement une création `!! x`), il est clair, à partir du contexte, que `x` n'est pas vide, et vous ne voulez pas le tester.

La question de la vacuité des références renvoie à une question plus générale de correction logicielle. Pour prouver qu'un système est correct, il est nécessaire de prouver qu'aucun appel n'est jamais effectué avec une référence vide et que toutes les assertions du logiciel (que nous étudierons dans un prochain chapitre) sont vérifiées au moment approprié de l'exécution. Pour la non-vacuité comme pour la correction des assertions, il serait préférable d'avoir un mécanisme automatique (un démonstrateur de programmes intégré au compilateur ou conçu comme un outil logiciel séparé) pour assurer qu'un système logiciel est correct. En l'absence de tels outils, le résultat d'une violation est une erreur à l'exécution — une exception. Les développeurs peuvent protéger leurs logiciels contre de telles situations de deux manières :

- Lors de l'écriture du logiciel, essayer d'éviter les situations erronées qui peuvent survenir à l'exécution en utilisant tous les moyens possibles : développement systématique et soigné, inspection des classes, utilisation d'outils effectuant des vérifications au moins partielles.
- Si un doute quelconque subsiste et que des échecs à l'exécution ne sont pas acceptables, équiper le logiciel de moyens de gestion des exceptions.

8.6 OPÉRATIONS SUR LES RÉFÉRENCES

Nous avons vu une manière de changer la valeur d'une référence `x` : utiliser une instruction de création de la forme `!! x`, qui crée un nouvel objet et l'attache à `x`. Un certain nombre d'autres opérations intéressantes sont disponibles sur les références.

Attacher une référence à un objet

Jusqu'à présent, les classes de ce chapitre avaient des attributs, mais pas de routines. Comme on l'a vu, cela les rend presque inutiles : il n'est pas possible de changer un champ dans un objet existant. Il nous faut des moyens de modifier la valeur des références, sans pour autant utiliser des instructions de la forme `my_beloved.loved_one := me` à la Pascal-C-Java-C++ (pour positionner directement le champ `loved_one` d'un objet), qui violent la rétention d'information et sont syntaxiquement illégales dans notre notation.

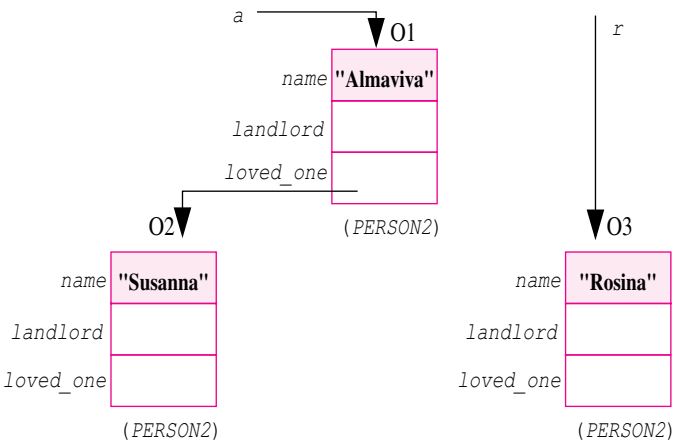
Pour modifier les champs des objets, une routine devra appeler d'autres routines, que les auteurs de la classe correspondante auront spécifiquement conçues dans à cette intention. Adaptons la classe `PERSON1` pour inclure une telle procédure, qui changera le champ `loved_one` pour l'attacher à un nouvel objet. Voici le résultat :

```
class PERSON2 feature
        name: STRING
        loved_one, landlord: PERSON2
        set_loved (l: PERSON2) is
                        -- Attacher le champ loved_one de l'objet courant à l.
                do
                        loved_one := l
                end
end
```

La procédure `set_loved` affecte au champ `loved_one` de l'instance courante de `PERSON2`, un champ référence, la valeur d'une autre référence, `l`. L'affectation de référence (comme l'affectation

de valeurs simples, tels les entiers) utilise le symbole `:=`, en plaçant la source de l'affectation à droite et la cible à gauche. Dans ce cas, puisque les source et cible ont des types références, une telle affectation est appelée affectation de référence.

L'effet d'une affectation de référence est exactement ce que son nom suggère : la référence cible devient rattachée à l'objet auquel la référence source est attachée, ou devient vide si la source était vide. Supposez, par exemple, que nous débutions dans la situation illustrée au sommet de la page suivante ; pour éviter de surcharger la figure, les champs `landlord` et les champs `loved_one`, non concernés, ont été laissés en blanc.



Avant l'affectation de référence

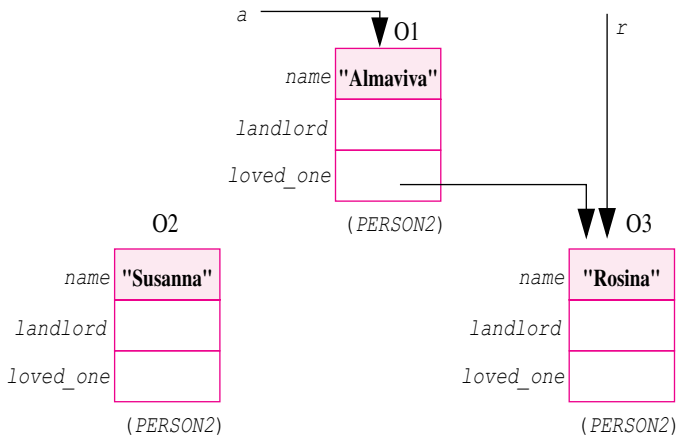
Supposez que nous exécutions l'appel de procédure :

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

où `a` est attaché à l'objet en haut (O1) et `r` à l'objet en bas à droite (O3). Étant donné la manière dont est écrit `set_loved`, cela exécutera l'affectation :

```
loved_one := l
```

avec O1 comme objet courant et `l` ayant la même valeur que `r`, une référence à O3. Le résultat est de rattacher le champ `loved_one` de O1 :



Après l'affectation de référence

Si *r* avait été une référence vide, l'affectation aurait rendu le champ *loved_one* de O1 également vide.

> Une question naturelle se pose ici : qu'arrive-t-il à l'objet auquel le champ modifié était attaché initialement — O2 dans la figure ? Est-ce que l'espace qu'il occupe sera automatiquement recyclé pour être utilisé par les instructions de création futures ?
>
> Il s'avère que cette question est tellement importante que nous lui consacrerons un chapitre entier — le prochain, sur la gestion de la mémoire et le ramasse-miettes. Aussi, retenez votre souffle jusque-là. Mais, il n'est pas trop tôt pour faire une observation de base : indépendamment de la réponse finale, une politique qui recyclerait toujours l'espace de l'objet serait incorrecte. En l'absence d'information supplémentaire concernant le système d'où la structure à l'exécution ci-dessus a été extraite, nous ne savons pas si d'autres références sont encore attachées à O2. Ainsi, une affectation de référence, en elle-même, ne nous dit pas ce qu'il faut faire avec l'objet précédemment attaché ; tout mécanisme de recyclage d'objets aura besoin d'un contexte plus précis.

Comparaison de références

De même que nous disposons d'une opération (l'affectation :=) pour attacher une référence à un objet, il nous faut un moyen pour tester si deux références sont attachées au même objet. Nous utiliserons l'opérateur d'égalité habituel =. Si *x* et *y* sont des entités de type référence, l'expression :

```
x = y
```

est vraie si et seulement si les références correspondantes sont soit toutes les deux vides, soit attachées toutes les deux au même objet. L'opérateur opposé, "pas égal", est écrit /= (une notation empruntée à Ada). Par exemple, l'expression :

```
r = a.loved_one
```

a pour valeur vrai dans la dernière figure, où les deux côtés du signe = désignent des références attachées à l'objet O3, mais pas sur l'avant-dernière figure, où *a.loved_one* est attaché à O2 et *r* à O3.

De même qu'une affectation à une référence est une opération de référence, et non une opération sur des objets, les expressions *x = y* et *x /= y* comparent des références, pas des objets. Ainsi, si *x* et *y* sont attachés à des objets distincts, *x = y* a pour valeur faux même si ces objets sont identiques champ à champ. Des opérations qui comparent les objets plutôt que les références seront introduites plus tard.

La valeur vide

Bien qu'il soit facile d'obtenir une référence vide — puisque tous les champs références sont, par défaut, initialisés à *Void* –, il nous sera pratique d'avoir un nom pour la valeur de référence, accessible dans tous les contextes, qui est toujours vide. La caractéristique prédéfinie :

```
Void
```

jouera ce rôle.

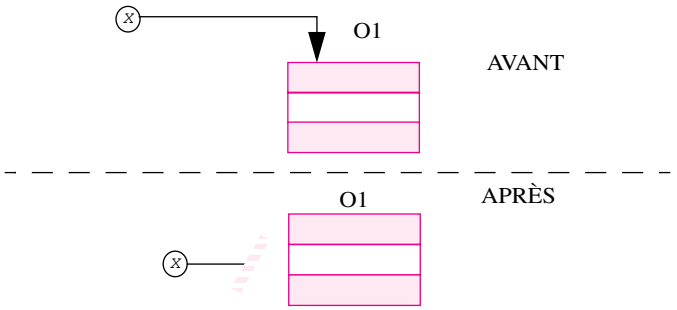
Deux utilisations fréquentes de *Void* consistent à tester si une certaine référence est vide, comme dans :

```
if x = Void then ...
```

et de rendre vide une référence, via l'affectation :

```
x := Void
```

Cette dernière affectation a pour effet de remettre la référence dans un état vide et, ainsi, de la détacher de l'objet attaché, s'il y en avait un :



Détacher une référence d'un objet

Le commentaire élaboré lors de l'étude générale de l'affectation de référence mérite d'être répété ici : l'affectation de `Void` à `x` n'a pas d'effet immédiat sur l'objet attaché (O1 dans la figure) ; elle coupe simplement le lien entre la référence et l'objet. Il serait incorrect de considérer qu'elle libère la mémoire associée à O1, puisqu'une autre référence peut être encore attachée à O1 même après que `x` ait été détaché de lui. Voir l'exposé sur la gestion mémoire dans le prochain chapitre.

Clonage d'objet et égalité

Les affectations de référence peuvent permettre à plusieurs références de devenir attachées à un objet unique. Parfois, vous aurez besoin d'une forme différente d'affectation, qui opère sur l'objet lui-même : plutôt qu'attacher une référence à un objet existant, vous voudrez créer une nouvelle copie d'un objet existant.

Cet objectif est atteint grâce à un appel à la fonction appelée `clone`. Si `y` est attaché à un objet OY, l'expression :

```
clone (y)
```

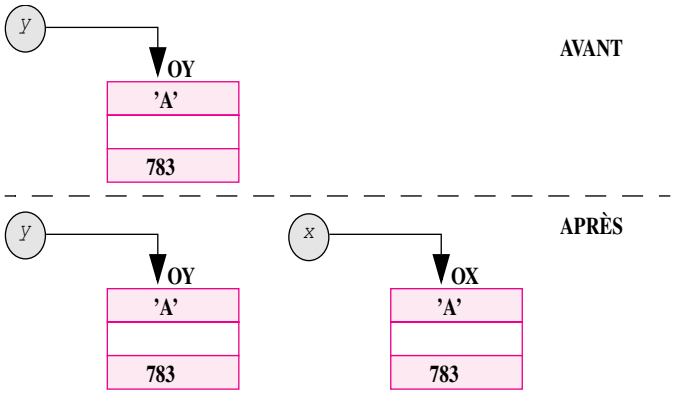
désigne un nouvel objet OX tel que OX a le même nombre de champs que OY, chaque champ de OX étant identique au champ correspondant de OY. Si `y` est vide, la valeur de `clone (y)` est aussi vide.

Pour dupliquer l'objet attaché à `y` et attacher l'objet résultant à `x` (ou rendre `x` vide si `y` est vide), vous pouvez utiliser un appel à `clone` dans une affectation :

[1]

```
x := clone (y)
```

Voici une illustration de ce mécanisme.



Nous avons, de même, besoin d'un mécanisme pour comparer deux objets. L'expression `x = y`, comme nous l'avons vu, remplit un autre rôle : comparer des références. Pour les objets, nous utiliserons la fonction `equal`. L'appel :

```
equal (x, y)
```

renvoie une valeur booléenne, vraie si et seulement si `x` et `y` sont soit tous deux vides, soit attachés à des objets dont les champs correspondants ont les mêmes valeurs. Si un système exécute l'affectation de clonage [1], l'état qui suit immédiatement cette affectation vérifiera `equal (x, y)`.

"La forme des opérations de clonage et d'égalité", page 271.

Il se peut que vous vous demandiez pourquoi la fonction `clone` possède un argument, et `equal` deux arguments traités de manière symétrique, plutôt que d'être appelées sous des formes plus proches du style habituel orienté objet, par exemple `y.twin` et `x.is_equal (y)`. La réponse se trouve dans la section de discussion, mais il n'est pas trop tôt pour essayer de la deviner.

Copier un objet

La fonction `clone` crée un nouvel objet qui est un double d'un objet existant. Parfois, l'objet cible existe déjà ; tout ce que nous voulons, c'est réécrire ses champs. La procédure `copy` fait cela. Elle est appelée par l'instruction :

```
x.copy (y)
```

où `x` et `y` ont le même type ; son effet est de copier les champs de l'objet attaché à `y` dans les champs correspondants de l'objet attaché à `x`.

Comme tout appel de caractéristique, un appel à `copy` impose que la cible `x` ne soit pas vide. De plus, `y` ne doit pas être vide non plus. Cette incapacité à traiter des valeurs vides distingue `copy` de `clone`.

Voir le chapitre 11 à propos des assertions.

Le fait que `y` ne doit pas être vide est tellement important que nous devrions avoir un moyen de l'exprimer formellement. Le problème est, en fait, plus général : comment exprimer dans une routine des **préconditions** concernant les arguments passés par ses appelants. De telles préconditions, qui sont des cas particuliers de la notion plus générale d'assertions, seront évoquées en détail dans un prochain chapitre. De même, nous apprendrons à exprimer par des **postconditions** des propriétés sémantiques fondamentales comme l'observation, faite ci-dessus, selon laquelle le résultat de `clone` vérifie `equal`.

La procédure `copy` peut être considérée comme plus fondamentale que la fonction `clone` dans la mesure où nous pouvons, au moins pour une classe n'ayant pas de procédure de création, exprimer `clone` en fonction de `copy` grâce à la fonction équivalente suivante :

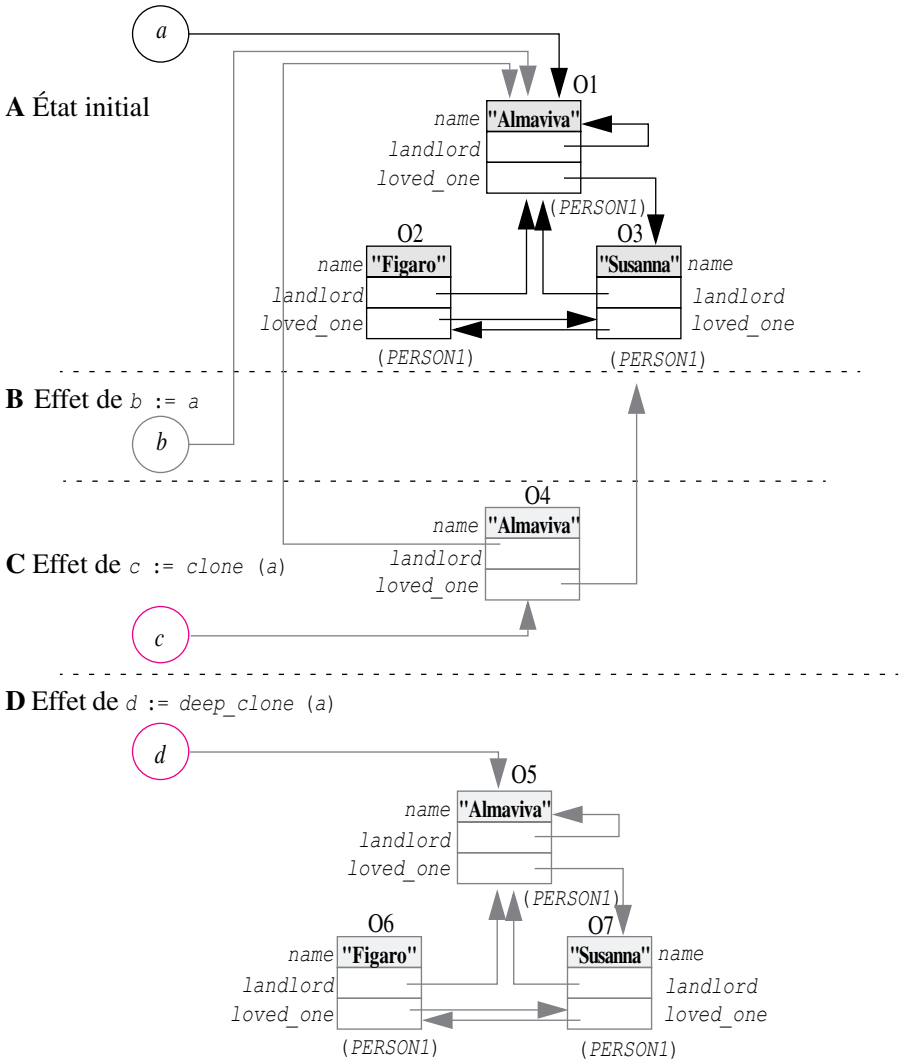
```
clone (y: SOME_TYPE) is
            -- Vide si y est vide : sinon, double de l'objet attaché à y
      do
            if y /= Void then
                  !! Result           -- Valide uniquement en l'absence de procédure
                                      -- de création
                  Result.copy (y)
            end
      end
```

Durant l'exécution d'un appel de fonction, `Result` est automatiquement initialisé selon les mêmes règles que celles définies ci-dessus pour les attributs. C'est la raison pour laquelle **if** n'a pas besoin de **else** : puisque `Result` est initialisé à `Void`, le résultat de la fonction ci-dessus est une valeur vide si `y` est vide.

Clonage profond et comparaison

La forme de copie et de comparaison réalisée par les routines `clone`, `equal` et `copy` peut être appelée **superficielle**, puisque ces opérations ne travaillent que sur le premier niveau d'un objet, n'essayant jamais de suivre les références. On a aussi besoin de variantes **profondes** qui parcourent récursivement une structure entière.

Pour comprendre la différence, supposez, par exemple, que nous débutions avec la structure d'objets apparaissant en noir (sauf les noms des attributs et des classes) sous **A** dans la figure suivante, où l'entité `a` est attachée à l'objet étiqueté O1.



Différentes formes d'affectation et de clonage

À titre de comparaison, considérez d'abord la simple affectation de référence :

```
b := a
```

Illustrée par la figure **B**, elle attache simplement la cible `b` de l'affectation au même objet O1 que celui auquel la source `a` était attaché. Aucun nouvel objet n'est créé.

Ensuite, considérez l'opération de clonage :

```
c := clone (a)
```

Cette instruction créera, comme on le voit sous **C**, un nouvel objet O4 unique, identique champ à champ à O1. Elle copie les deux champs références dans les champs correspondants de O4, ce qui donne des références qui sont attachées aux mêmes objets O1 et O3 que les celles d'origine. Mais elle ne duplique pas O3 lui-même, ou tout autre objet différent de O1. C'est pourquoi l'opération de base `clone` est considérée comme superficielle : elle s'arrête au premier niveau de la structure d'objet.

> Notez qu'une autoréférence a disparu : le champ `landlord` de O1 était attaché à O1 lui-même. Dans O4, ce champ devient une référence à l'objet O1 original.

Dans d'autres cas, il se peut que vous vouliez poursuivre et dupliquer récursivement une structure, sans introduire de partage de références comme celui introduit lors de la création de O4. La fonction `deep_clone` réalise cela. Plutôt que de s'arrêter à l'objet attaché à `y`, le processus de création `deep_clone (y)` parcourt récursivement tout champ référence contenu dans cet objet et en duplique la structure complète. (Si `y` est vide, le résultat est également vide.) La fonction est, bien sûr, capable de traiter correctement les structures cycliques de références.

Le bas de la figure, étiqueté **D**, illustre le résultat de l'exécution de :

```
d := deep_clone (a)
```

Ce cas n'introduit pas de nouveaux partages ; tous les objets accessibles directement ou indirectement à partir de O1 (l'objet attaché à `a`) seront dupliqués pour donner des nouveaux objets, O5, O6 et O7. Il n'y a pas de connexion entre les anciens objets (O1, O2 et O3) et les nouveaux. L'objet O5, imitant O1, présente une autoréférence.

De même que nous avons besoin à la fois d'opérations de clonage profond et superficiel, l'égalité doit offrir une variante profonde. La fonction `deep_equal` compare des structures d'objets pour déterminer si elles sont structurellement identiques. Dans l'exemple de la figure, `deep_equal` est vérifié pour toute paire dans `a`, `b` et `d` ; mais, alors que `equal (a, c)` est vrai, puisque les objets correspondants O1 et O4 sont identiques champ à champ, `equal (a, d)` est faux. En fait, equal n'est jamais vérifié entre d et aucun des trois autres. (`equal (a, b)` et `equal (b, c)` sont tous deux vérifiés.) Dans le cas général, nous pouvons constater les propriétés suivantes :

- Après une affectation `x := clone (y)` ou un appel `x.copy (y)`, l'expression `equal (x, y)` a pour valeur vrai. (Pour la première affectation, cette propriété est vérifiée, que `y` soit vide ou non.)
- Après `x := deep_clone (y)`, l'expression `deep_equal (x, y)` a pour valeur vrai.

Ces propriétés seront exprimées par des postconditions dans les routines correspondantes.

Stockage profond : une première approche de la persistance

L'étude de la copie et de l'égalité profondes nous amènera à étudier un autre mécanisme qui, dans les environnements où il est disponible, constitue un des avantages pratiques majeurs de la méthode OO.

Jusqu'à présent, nous n'avons pas évoqué la question des entrées et des sorties. Mais, bien sûr, un système orienté objet devra communiquer avec d'autres systèmes et avec le reste du monde. Puisque l'information qu'il manipule existe sous forme d'objets, il doit être capable d'écrire et de lire des objets sur fichiers, bases de données, lignes de communication et autres médias.

Par souci de simplicité, cette section supposera que le problème consiste à écrire ou lire sur des fichiers, et utilisera les termes “stockage” et “récupération” pour ces opérations (“entrées” et “sorties” seraient également adéquats.) Mais les mécanismes étudiés doivent être aussi applicables aux échanges d’objets avec le monde extérieur via d’autres moyens de communication, par exemple en envoyant et recevant des objets sur un réseau.

Pour les instances de classes comme *POINT* ou *BOOK1*, le stockage et la récupération d’objets ne posent pas de problème particulier. Ces classes, utilisées comme exemples au début de ce chapitre, ont des attributs de type comme *INTEGER*, *REAL* et *STRING* pour lesquels des représentations externes bien connues existent. Stocker une instance d’une telle classe dans un fichier, ou la récupérer à partir de ce fichier, est semblable à une opération de sortie ou d’entrée d’un enregistrement Pascal ou d’une structure C. Il faut, bien sûr, prendre en compte les particularités de représentations des données sur des machines différentes et dans des langages différents. (C, par exemple, a une convention spéciale pour les chaînes, auxquelles le langage impose d’être terminées par un caractère nul) ; mais ce sont des problèmes techniques bien connus pour lesquels des solutions standard existent. Aussi, peut-on attendre d’un bon environnement OO qu’il fournisse, pour de tels objets, des procédures générales, disons *read* et *write*, qui, comme *clone*, *copy* et autres, seraient disponibles pour toutes les classes.

Mais, de tels mécanismes ne nous mèneront pas loin, car ils ne gèrent pas un composant essentiel de la structure des objets : les références. Puisque les références peuvent être représentées en mémoire (par des adresses ou autres), il est possible d’en trouver également une représentation externe. Ce n’est pas la partie difficile du problème. Ce qui compte, c’est le sens de ces références. Une référence attachée à un objet est inutile sans cet objet.

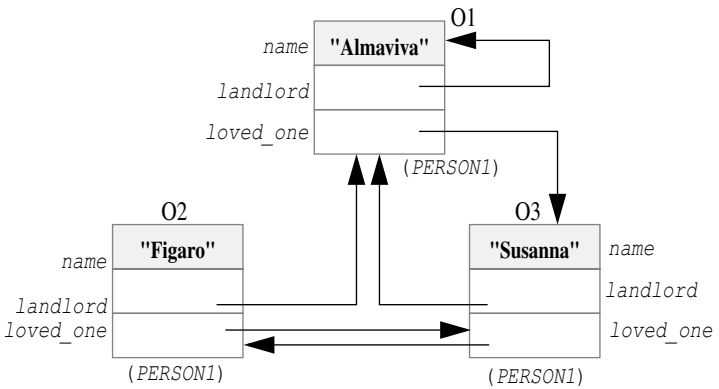
Ainsi, dès que nous commençons à traiter le cas d’objets non triviaux — des objets qui contiennent des références — nous ne pouvons pas nous satisfaire d’un mécanisme de stockage et de récupération qui ne marcherait qu’avec des objets individuels ; le mécanisme doit traiter, outre l’objet, toutes ses dépendances selon la définition suivante :

Définition : dépendances directes, dépendances

Les dépendances directes d’un objet sont les objets attachés à ses champs références, s’il y en a.

Les dépendances d’un objet sont l’objet lui-même et (récursivement) les dépendances de ses dépendances directes.

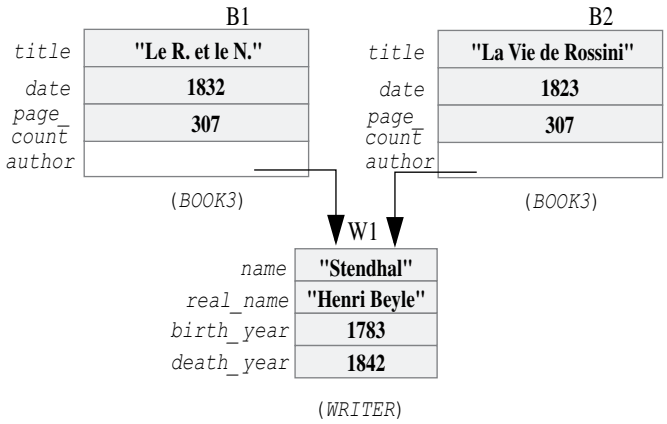
Avec la structure d’objets indiquée ci-dessous (identique aux exemples précédents), cela n’aurait pas de sens de stocker uniquement l’objet *O1* dans un fichier ou de le transmettre sur un réseau. L’opération doit aussi inclure les dépendances de O1, O2 et O3 :



Trois objets mutuellement dépendants

Dans cet exemple, chacun des trois objets est dépendant des deux autres. Dans l'exemple *BOOK3* reproduit ci-dessous, nous pourrons stocker W1 tout seul et, chaque fois que nous stockerons B1 ou B2, nous devrons également stocker W1.

Les objets "livre" et "écrivain"



La notion de dépendance était implicitement présente dans la description de `deep_equal`. Voici la règle générale :

> ***Principe de fermeture persistante***
>
> Chaque fois qu'un mécanisme de stockage stocke un objet, il doit le stocker avec les dépendances de cet objet. Chaque fois qu'un mécanisme de récupération récupère un objet précédemment stocké, il doit aussi récupérer toute dépendance de cet objet qui n'a pas encore été récupérée.

Le mécanisme de base qui nous fournit cela est appelé le service *STORABLE*, du nom de la classe de la bibliothèque Base qui contient les caractéristiques correspondantes. Les caractéristiques de base de *STORABLE* sont de la forme :

```
store (f: IO_MEDIUM)
retrieved (f: IO_MEDIUM): STORABLE
```

Un appel de la forme `x.store (f)` a pour effet de stocker l'objet attaché à `x`, ainsi que toutes ses dépendances, dans le fichier associé à `f`. L'objet attaché à `x` est dit **objet de tête** de la structure stockée. La classe génératrice de `x` doit être un descendant de *STORABLE* (c'est-à-dire qu'elle doit hériter directement ou indirectement de *STORABLE*) ; ainsi, vous devrez ajouter *STORABLE* à la liste de ses parents, si elle n'y est pas déjà. Cela ne s'applique qu'à la classe génératrice de l'objet de tête ; il n'y a pas d'exigence particulière pour les classes génératrices des objets dépendants — heureusement, puisqu'un objet de tête peut avoir un nombre arbitraire de descendants directs et indirects, instances de classes arbitraires.

La classe *IO_MEDIUM* est une autre classe de la bibliothèque Base, recouvrant non seulement les fichiers mais aussi les structures pour les transmissions sur réseau. Il est clair que `f` ne doit pas être vide et que le fichier attaché ou le médium de transmission doit être accessible en écriture.

Le résultat d'un appel `retrieved (f)` est une structure d'objets récursivement identique, au sens de `deep_clone`, à la structure complète d'objets stockée dans `f` par un appel précédent à `store`. La caractéristique `retrieved` est une fonction ; son résultat est une référence à l'objet de tête de la structure récupérée.

Si vous avez déjà une connaissance de base concernant l'héritage et les règles de type associées, vous pouvez voir que *retrieved* soulève un problème de typage. Le résultat de cette fonction est de type *STORABLE* ; mais il semble que son utilisation normale sera dans des affectations de la forme `x := retrieved (f)` où le type de *x* est un descendant propre de *STORABLE*, pas *STORABLE* lui-même, bien que les règles de type ne permettent `x := y` que si le type de *y* est un descendant du type de *x* — pas dans l'autre sens. La clé de ce problème se trouve dans une construction importante, la **tentative d'affectation**. Tout cela sera examiné en détail quand nous étudierons l'héritage et les règles de type associées.

Voir "LA TENTATIVE D'AFFECTATION", 16.5, page 572.

Le mécanisme *STORABLE* est notre premier exemple de ce qu'on appelle un service de **persistance**. Un objet est persistant s'il survit aux sessions individuelles successives des systèmes qui le manipulent. *STORABLE* ne fournit qu'une solution partielle au problème de la persistance, et souffre de plusieurs limitations :

- Dans la structure stockée et récupérée, un seul objet est connu individuellement : l'objet de tête. Il peut être souhaitable de sauvegarder également l'identité des autres objets.
- En conséquence, le mécanisme n'est pas directement utilisable pour récupérer sélectivement des objets par des requêtes fondées sur le contenu ou des mots-clés, comme cela se fait dans les systèmes de gestion de bases de données.
- Un appel à *retrieved* crée une structure d'objets complète. Cela implique que vous ne pouvez pas utiliser plusieurs appels pour récupérer diverses parties d'une structure, à moins qu'elles ne soient disjointes.

Prendre en compte ce problème revient à dépasser ce simple mécanisme de persistance pour adopter la notion de base de données orientée objet, présentée dans un prochain chapitre, qui évoquera également un certain nombre de questions liées à *STORABLE* et aux autres mécanismes de persistance, comme l'évolution de schéma (qu'arrive-t-il quand vous récupérez un objet dont la classe a changé ?) et l'identité des objets persistants.

Chapitre 31.

Mais, les limitations introduites ci-dessus ne devraient pas pour autant minimiser les bénéfices considérables obtenus, en pratique, avec le mécanisme *STORABLE* tel qu'il a été décrit ci-dessus. En fait, on peut conjecturer que l'absence d'un tel mécanisme a été l'un des obstacles majeurs à l'utilisation de structures de données sophistiquées dans les environnements traditionnels de développement. Sans *STORABLE* ou son équivalent, stocker une structure de données requiert un effort de programmation majeur : pour chaque genre de structure qui doit offrir des propriétés de persistance, vous devez écrire un mécanisme spécial d'entrée et de sortie, contenant un ensemble de procédures mutuellement récursives (une pour chaque type) et des mécanismes de parcours spécialisés (qui sont particulièrement difficiles à écrire dans le cas de structures potentiellement cycliques). Mais la mise en oeuvre initiale n'est pas la partie la plus ardue : comme d'habitude, les vrais problèmes commencent quand la structure change et qu'il vous faut mettre à jour ces procédures.

Avec *STORABLE*, un mécanisme prédéfini est disponible indépendamment de la structure de votre objet, de sa complexité et de l'évolution du logiciel.

Une application typique du mécanisme *STORABLE* est un service SAVE. Considérez un système interactif, par exemple un éditeur de texte, un éditeur graphique, un programme de dessin ou un système de conception assistée par ordinateur ; il doit fournir à ses utilisateurs une commande SAVE pour stocker, dans un fichier, l'état de la session courante. L'information stockée devrait être suffisante pour redémarrer la session à n'importe quel moment, et doit donc inclure toutes les structures de données importantes du système. Ecrire une telle procédure de manière spécifique introduit les difficultés mentionnées ; en particulier, il vous

faudra la mettre à jour chaque fois que vous changerez une classe pendant le développement. Mais, avec le mécanisme *STORABLE* et un choix approprié d'objet de tête, vous pouvez implémenter le service SAVE en utilisant une seule instruction :

```
head.store (save_file)
```

Ce mécanisme justifierait, à lui seul, le choix d'un environnement orienté objet par rapport à ses alter ego plus traditionnels.

8.7 OBJETS COMPOSITES ET TYPES EXPANSÉS

La présentation précédente a décrit l'essentiel de la structure à l'exécution. Elle fait jouer un rôle important aux références. Pour terminer cette présentation, nous devons voir comment sont gérées les valeurs qui ne sont *pas* des références aux objets, mais les objets eux-mêmes.

Les références ne sont pas suffisantes

Les valeurs que nous avons considérées jusqu'à présent, si ce n'étaient des entiers, booléens et autres, étaient des références aux objets. Deux raisons sous-tendent l'intérêt de recourir à des entités dont la valeur est un objet :

- Un objectif important annoncé dans le chapitre précédent est d'avoir un système de types complètement uniforme, dans lequel les types de base (comme *BOOLEAN* et *INTEGER*) sont gérés de la même manière que les types définis par le développeur (comme *POINT* ou *BOOK*). Mais, si vous utilisez une entité *n* pour manipuler un entier, vous voudrez presque toujours que la valeur de *n* soit un entier, par exemple 3, et non une référence à un objet contenant la valeur 3. La raison est, en partie, une question d'efficacité — pensez au temps et à l'espace perdus si chaque accès à un entier était indirect ; tout aussi important, dans ce cas, est l'objectif de modélisation systématique. Un entier n'est conceptuellement pas la même chose qu'une référence à un entier.
- Même avec des objets complexes définis par le développeur, nous préférerons peut-être, dans certains cas, considérer que l'objet O1 contient un sous-objet O2 plutôt qu'une référence à un autre objet O2. La raison, à nouveau, peut être l'efficacité, la modélisation systématique ou les deux.

Types expansés

La réponse à ce besoin de modélisation d'objets composites est simple. Soit *C* une classe déclarée, comme toutes les classes jusqu'à présent, sous la forme :

```
class C feature
        ...
end
```

C peut être utilisée comme un type. Toute entité déclarée de type *C* représente une référence ; en conséquence, *C* est appelée un **type référence**.

Maintenant, supposons que nous ayons besoin d'une entité *x* dont la valeur à l'exécution soit une instance de *C* — pas une référence à une telle instance. Nous pouvons obtenir cela en déclarant *x* comme :

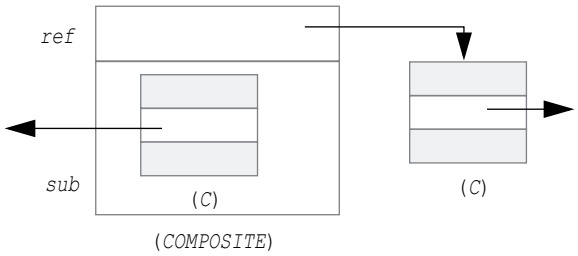
```
x : expanded C
```

Cette notation utilise un nouveau mot-clé, **expanded**. La notation **expanded** *C* désigne un type. Les instances de ce type sont exactement les mêmes que les instances de *C*. La seule différence concerne les déclarations qui utilisent ces types : une entité de type *C* désigne une référence qui peut devenir attachée à une instance de *C* ; une entité de type **expanded** *C*, comme *x* ci-dessus, désigne directement une instance de *C*.

Ce mécanisme ajoute la notion d'objet composite à la structure définie dans les sections précédentes. Un objet *O* est dit composite si au moins un de ses champs est lui-même un objet — appelé un **sous-objet** de *O*. L'exemple de classe suivant (en omettant à nouveau les routines) montre comment décrire des objets composites :

```
class COMPOSITE feature
        ref: C
        sub: expanded C
end
```

Cette classe repose sur *C* telle qu'elle est déclarée ci-dessus. *COMPOSITE* a des attributs : *ref*, qui désigne une référence, et *sub*, qui désigne un sous-objet ; *sub* rend la classe composite. Toute instance directe de *COMPOSITE* peut ressembler à ceci :



Un objet composite avec un sous-objet

Le champ *ref* est une référence attachée à une instance de *C* (ou vide). Le champ *sub* (qui ne peut être vide) contient une instance de *C*.

Une extension de notation sera pratique ici. Vous pouvez parfois écrire une classe *E* en voulant que toutes les entités déclarées de type *E* soient expansées. Pour rendre cette intention explicite, déclarez la classe comme :

```
expanded class E feature
        ... La suite comme pour toute autre classe ...
end
```

Une classe définie de cette manière est appelée une classe expansée. Ici, également, la nouvelle déclaration ne change rien aux instances de *E* : elles restent les mêmes que si la classe avait été déclarée simplement comme **class** *E* .... Mais, une entité déclarée de type *E* désignera dorénavant un objet, pas une référence. Du fait de cette nouvelle possibilité, la notion de type expansé contient deux cas :

Définition : type expansé

Un type est dit expansé dans les deux cas suivants :

- Il est de la forme **expanded** *C*.
- Il est de la forme *E*, où *E* est une classe expansée.

Ce n'est pas une erreur de déclarer une entité *x* comme étant de type **expanded** *E* si *E* est une classe expansée, mais c'est inutile, puisque le résultat, dans ce cas, est le même que si vous déclariez *x* comme étant simplement de type *E*.

Nous avons maintenant deux genres de types ; un type qui n'est pas expansé est un **type référence** (un terme déjà utilisé dans ce chapitre). Nous pouvons appliquer la même terminologie aux entités déclarées de la même manière : entités références et entités expansées. De même, une classe est une classe expansée si elle a été déclarée par **expanded class**..., sinon c'est une classe référence.

Le rôle des types expansés

Pourquoi avons-nous besoin de types expansés ? Ils jouent trois rôles majeurs :

- améliorer l'efficacité,
- fournir une meilleure modélisation,
- supporter les types de base dans un système de types uniforme orienté objet.

La première application est peut-être la plus évidente de prime abord : sans les types expansés, vous auriez à utiliser des références chaque fois que vous voulez décrire un objet composite. Cela veut dire qu'accéder à leurs sous-objets imposerait à toute opération de suivre une référence — "déréférencer", comme on le dit parfois — ce qui induit une perte de temps. Il y a aussi une perte en espace, puisque la structure à l'exécution doit allouer de la place aux références elles-mêmes.

Cet argument de performance n'est, cependant, pas la première justification. L'argument clé, en phase avec l'idée générale de ce chapitre selon laquelle la construction de logiciel orienté objet est une activité de modélisation, vient du besoin de modéliser des objets composites séparément des objets qui contiennent des références à d'autres objets. Ce n'est pas une question d'implémentation, mais une question conceptuelle.

Considérez les deux déclarations d'attributs :

D1 • *ref*: *S*

D2 • *exp*: **expanded** *S*

qui apparaissent dans une classe *C* (en supposant que *S* est une classe référence). La déclaration D1 exprime simplement que toute instance de *C* "a connaissance" d'une certaine instance de *S* (sauf si *ref* est vide). La déclaration D2 s'engage plus : elle indique que toute instance de *C* **contient** une instance de *S*. Hormis les questions d'implémentation, c'est une relation toute différente.

En particulier, la relation "contient" fournie par les types expansés ne permet aucun **partage** des éléments contenus, tandis que la relation "a connaissance" permet à deux références ou plus d'être attachées au même objet.

Vous pouvez appliquer cette propriété pour obtenir une modélisation correcte des relations entre objets. Considérez, par exemple, cette déclaration de classe :

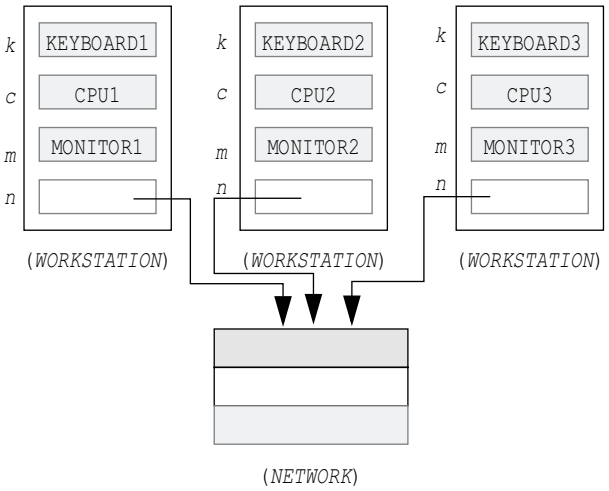
Toutes les classes indiquées sont supposées être des classes références (non-expansées).

```
class WORKSTATION feature

        k: expanded KEYBOARD
        c: expanded CPU
        m: expanded MONITOR
        n: NETWORK
        ...
end
```

Dans ce modèle, une station de travail informatique a un clavier, un CPU (unité centrale de traitement) et un moniteur, et est attachée à un réseau. Les clavier, CPU et moniteur font partie d'une station unique et ne peuvent pas être partagés entre stations. Le composant réseau est, lui, partagé : plusieurs stations peuvent être connectées au même réseau. La définition de classe reflète ces propriétés en utilisant des types expansés pour les trois premiers attributs et un type référence pour l'attribut réseau.

Relations "a connaissance" et "contient" entre objets



Ainsi, le concept de type expansé, qui semblait être, au début, une technique d'implémentation, sert, en fait, à décrire certaines relations utilisées dans la modélisation de l'information. La relation "contient" et son inverse, souvent appelée "fait partie de", sont centrales dans tout effort de construction des modèles de systèmes externes ; elles apparaissent dans les méthodes d'analyse et dans la modélisation de bases de données.

Voir "UN SYSTÈME DE TYPES UNIFORME", 7.4, page 175. L'aperçu de la classe REAL se trouve page 192.

La troisième application majeure des types expansés est, en fait, un cas spécial de la seconde. Le chapitre précédent a insisté sur l'intérêt d'un système uniforme de types, fondé sur la notion de classe, qui intègre à la fois les types définis par le développeur et les types de base. L'exemple de *REAL* a été utilisé pour montrer comment, à l'aide de caractéristiques infixes et préfixes, nous pouvons, de fait, modéliser la notion de nombre réel comme une classe ; nous pouvons faire de même avec les autres types de base *BOOLEAN*, *CHARACTER*, *INTEGER*, *DOUBLE*. Mais un problème subsiste. Si ces classes étaient traitées comme des classes références, une entité déclarée avec un type de base, comme :

```
r: REAL
```

désignerait, à l'exécution, une référence à un possible objet contenant une valeur (ici de type *REAL*). Cela est inacceptable : pour se conformer à la pratique courante, la valeur de *r* devrait être la valeur réelle elle-même. La solution découle de ce qui précède : définir la classe *REAL* comme expansée. Sa déclaration sera :

```
expanded class REAL feature
        ... Déclarations de caractéristiques comme avant
        (voir page 192) ...
end
```

Tous les autres types de base sont définis de manière similaire par des classes expansées.

Agrégation

Dans certains domaines de l'informatique — les bases de données, la modélisation de l'information, l'analyse d'exigences — les auteurs ont développé une classification des relations qui peuvent exister entre éléments d'un système modélisé. La relation d'"agrégation" est souvent mentionnée dans ce contexte ; elle sert à exprimer que tout objet d'un certain type est une combinaison (un agrégat) d'objets, chacun ayant un type spécifié. Par exemple, nous pouvons définir une "voiture" comme une agrégation d'un "moteur", d'une "carrosserie", etc.

Les types expansés fournissent un mécanisme équivalent. Nous pouvons, par exemple, déclarer une classe *CAR* avec des caractéristiques de type **expanded** *ENGINE* et **expanded** *BODY*. Une autre manière d'exprimer cette observation est de remarquer que l'agrégation peut être interprétée par la relation "client expansé", où l'on dit qu'une classe *C* est un client expansé d'une classe *S* si elle contient une déclaration d'une caractéristique de type **expanded** *S* (ou simplement *S*, si *S* est expansé). Cette approche de modélisation présente l'avantage de réduire la relation "client expansé" à un cas spécial de la relation client générale, de façon que nous puissions utiliser un cadre et une notation uniques pour combiner des dépendances correspondant à une agrégation (c'est-à-dire des dépendances sur des sous-objets comme la relation entre *WORKSTATION* et *KEYBOARD* dans l'exemple précédent) avec des dépendances qui permettent le partage (comme la relation entre *WORKSTATION* et *NETWORK*).

Avec l'approche orientée objet, on peut éviter la multiplicité des relations que l'on trouve dans la littérature concernant la modélisation de l'information, et couvrir tous les cas possibles avec simplement deux relations : client (expansé ou non) et héritage.

Propriétés des types expansés

Considérez un type expansé *E* (d'une des deux formes) et une entité expansée *x* de type *E*.

Puisque la valeur de *x* est toujours un objet, il ne peut jamais être vide. Ainsi, l'expression

```
x = Void
```

"Références vides et appels", page 241.

aura toujours la valeur faux, et un appel de la forme *x*.*some_feature* (*arg1*, ...) ne lèvera jamais l'exception "appel avec une cible vide" qui est possible dans le cas des références.

Soit l'objet O la valeur de *x*. Comme dans le cas d'une référence non vide, *x* est dit attaché à O. Ainsi, pour toute entité non vide, nous pouvons parler de l'objet attaché, qu'il s'agisse d'une entité ayant un type référence ou expansé.

Que devient la création ? L'instruction :

```
!! x
```

Voir "Effet d'une instruction de création de base", page 234.

peut être appliquée à un objet expansé *x*. Pour une référence *x*, elle permettait de réaliser trois opérations : (C1) créer un nouvel objet ; (C2) initialiser ses champs avec les valeurs par défaut ; (C3) l'attacher à *x*. Pour un *x* expansé, l'étape C1 n'est pas pertinente, tout comme l'étape C3 ; ainsi, son seul effet est de positionner tous les champs à leurs valeurs par défaut.

Plus généralement, la présence de types expansés affecte l'initialisation par défaut effectuée dans l'étape C2. Soit une classe, expansée ou non, possédant au moins un attribut expansé :

```
class F feature
        u: BOOLEAN
        v: INTEGER
        w: REAL
        x: C
        y: expanded C
        z: E
        ...
end
```

où `E` est expansé alors que `C` ne l'est pas. L'initialisation d'une instance directe de `F` impose de positionner le champ `u` à faux, le champ `v` à 0, le champ `w` à 0.0, le champ `x` à une référence vide, et `y` et `z` à des instances de `C` et `E` respectivement, dont les champs sont eux-mêmes initialisés selon les règles standard. Ce processus d'initialisation doit être appliqué récursivement, puisque `C` et `E` sont eux-mêmes formés de champs expansés.

Les cycles dans la relation client ont été étudiés dans "Autoréférence", page 228.

Comme vous l'avez peut-être compris, il est nécessaire d'introduire une restriction pour rendre utilisables les types expansés (pour s'assurer que le processus récursif ainsi défini termine toujours) : bien que, comme nous l'avons déjà vu, la relation client puisse, en général, inclure des cycles, ceux-ci ne doivent pas utiliser des attributs expansés. Par exemple, il n'est pas permis à la classe `C` d'avoir un attribut de type **expanded** `D` si `D` a un attribut de type **expanded** `C` ; cela voudrait dire que tout objet de type `C` contient un sous-objet de type `D` et inversement — une impossibilité manifeste. D'où la règle suivante, fondée sur la notion de client expansé qui a déjà été introduite informellement ci-dessus :

Règle du client expansé

Soit "client expansé" la relation entre classes définie comme suit : `C` est un client expansé de `S` si un attribut de `C` est un type expansé fondé sur `S` (c'est-à-dire **expanded** `S`, ou simplement `S`, si `S` est une classe expansée).

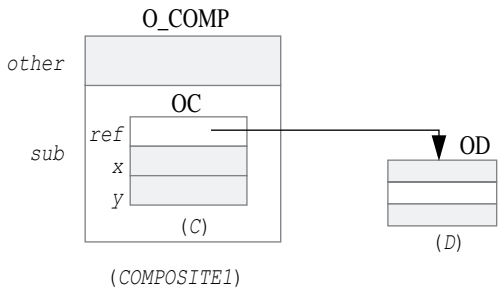
La relation client expansé ne doit alors contenir aucun cycle.

En d'autres termes, il ne doit pas exister un ensemble de classes `A`, `B`, `C`, ... `N` tel que `A` est un client expansé de `B`, `B` un client expansé de `C`, etc., quand `N` est un client expansé de `A`. En particulier, `A` ne doit pas avoir d'attribut de type `expanded A`, car cela rendrait `A` client expansé de lui-même.

Pas de référence aux sous-objets

Un dernier commentaire sur les types expansés nous apprendra comment mélanger références et sous-objets. Une classe expansée ou un type expansé fondé sur une classe référence peut avoir des attributs références. Ainsi, un sous-objet peut contenir des références attachées à des objets :

Un sous-objet avec une référence à un autre objet



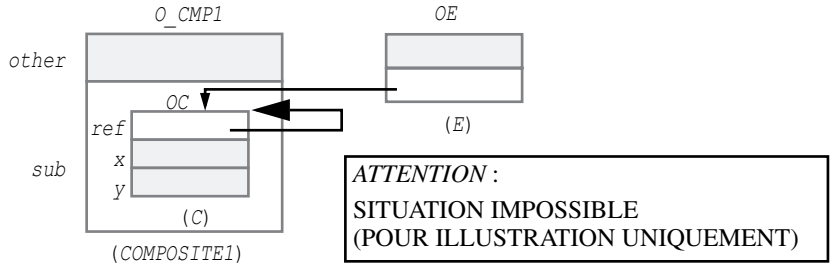
La situation présentée ci-dessus suppose les déclarations suivantes :

```
class COMPOSITE1 feature
      other: SOME_TYPE
      sub: expanded C
end
class C feature
      ref: D
      x: OTHER_TYPE; y: YET_ANOTHER_TYPE
end
class D feature
      ...
end
```

Chaque instance *COMPOSITE*, comme O_COMP dans la figure, a un sous-objet (OC dans la figure) qui contient une référence *ref* pouvant être attachée à un objet (OD dans la figure).

Mais la situation inverse, dans laquelle une référence deviendrait attachée à un sous-objet, est impossible. (Cela découlera des règles sur l'affectation et le passage d'argument, étudiées dans la prochaine section.) Ainsi, la structure à l'exécution ne peut jamais aboutir à l'état décrit par la figure, où OE contient une référence à OC, un sous-objet de O_CMP1, et OC, de même, contient une référence à lui-même.

Une référence à un sous-objet



Cette règle est critiquable, car elle limite la puissance de modélisation de l'approche. Des versions précédentes de la notation présentée dans ce livre permettaient, en fait, d'avoir des références à des sous-objets. Mais cette possibilité engendrait trop de problèmes :

Le ramasse-miettes est étudié dans le prochain chapitre.

- D'un point de vue implémentation, le mécanisme de ramasse-miettes doit être conçu pour traiter les références à des sous-objets même si, dans une exécution donnée, celles-ci sont rares, voire inexistantes. Cela induit une dégradation significative de la performance.
- Du point de vue de la modélisation, exclure les références aux sous-objets permet de simplifier les descriptions des systèmes en définissant une unité unique de référencement, l'objet.

La section de discussion signalera quelles règles précises d'attachement devraient être modifiées pour revenir au schéma dans lequel des références peuvent être attachées à des sous-objets.

Si vous sautez ce passage , allez à "UTILISER LES RÉFÉRENCES : BÉNÉFICES ET DANGERS", 8.9, page 262.

8.8 ATTACHEMENT : SÉMANTIQUE PAR RÉFÉRENCE ET PAR VALEUR

(Cette section fournit des informations plus spécialisées qui peuvent être omises en première lecture.)

L'introduction des types expansés nous oblige à regarder à nouveau deux opérations fondamentales étudiées précédemment dans ce chapitre : l'affectation, écrite `:=`, qui attache une référence à un objet, et l'opération associée de comparaison, écrite `=`. Puisque des entités peuvent maintenant désigner des objets aussi bien que des références à des objets, nous devons décider ce qu'affectation et égalité veulent dire dans le premier de ces cas.

Attachement

La sémantique de l'affectation concernera, en fait, plus que cette seule opération. Un autre cas dans lequel la valeur d'une entité peut changer est le passage d'argument dans les appels de routines. Soit une routine (procédure ou fonction) de la forme :

```
r (..., x: SOME_TYPE, ...)
```

Ici, l'entité `x` est l'un des **arguments formels** de `r`. Considérons maintenant un appel particulier à `r`, de l'une des deux formes possibles (non qualifiée et qualifiée) :

```
r (..., y, ...)
t.r (..., y, ...)
```

où l'expression `y` est l'**argument réel** ayant la même position dans la liste des arguments réels que `x` dans la liste des arguments formels.

Chaque fois que `r` démarre du fait d'un de ces appels, il initialise chacun de ses arguments formels avec la valeur de l'argument réel correspondant, comme `y` pour `x`.

Pour des raisons de simplicité et de cohérence, les règles qui gouvernent ces associations entre argument formel et argument réel sont les mêmes que les règles qui gouvernent l'affectation. En d'autres termes, l'effet sur `x` d'un tel appel est exactement le même que si `x` avait été la cible d'une affectation de la forme :

```
x := y
```

Cette règle conduit à la définition :

Définition : attachement

Un attachement de `y` à `x` est l'une des deux opérations :

- une affectation de la forme `x := y`,
- l'initialisation de `x` au moment de l'appel d'une routine, où `x` est un argument formel de la routine et `y` l'argument réel correspondant de l'appel.

Dans les deux cas, `x` est la **cible** de l'attachement et `y` sa **source**.

Ce sont exactement les mêmes règles qui s'appliquent dans les deux cas pour déterminer si un attachement est valide (en fonction des types de la cible et de la source) et, s'il l'est, ce qu'en sera l'effet à l'exécution.

Attachement par référence et par copie

Nous avons vu une première règle concernant l'effet d'un attachement lors de l'étude de l'affectation par référence. Si source et cible sont toutes deux des références, alors l'effet de l'affectation :

```
x := y
```

et du passage d'argument correspondant est de faire que *x* désigne la même référence que *y*. Cela a été illustré par de nombreux exemples. Si *y* est vide avant attachement, l'opération rendra *x* également vide ; si *y* est attaché à un objet, *x* deviendra attaché au même objet.

Que se passe-t-il, maintenant, quand les types de *x* et *y* sont expansés ? L'affectation par référence n'aurait aucun sens, mais une copie (la forme superficielle) est possible. Le sens d'un attachement d'une source expansée vers une cible expansée sera effectivement une copie. Avec les déclarations :

```
x, y: expanded SOME_CLASS
```

l'affectation `x := y` copiera chaque champ de l'objet attaché à *y* sur le champ correspondant de l'objet attaché à *x*, produisant le même effet que :

```
x.copy (y)
```

qui, bien sûr, est encore autorisé dans ce cas. (Dans le cas de types références, `x := y` et `x.copy (y)` sont tous deux légaux, mais ont des effets différents.)

Cette sémantique par copie des types expansés donne le résultat attendu dans le cas des types de base qui, comme on l'a vu ci-dessus, sont tous expansés. Par exemple, si *m* et *n* ont été déclarés de type *INTEGER*, vous vous attendez que l'affectation `m := n`, ou un passage d'argument correspondant, copie la valeur de *n* sur celle de *m*.

L'analyse que l'on vient juste d'appliquer à l'attachement se transpose immédiatement à une opération voisine : la comparaison. Considérez les expressions booléennes `x = y` et `x /= y`, qui ont des valeurs opposées. Si *x* et *y* ont des types références, les tests comparent, comme on l'a déjà vu, les références : `x = y` est vrai si et seulement si *x* et *y* sont soit tous deux vides, soit tous deux attachés au même objet. Pour des *x* et *y* expansés, cela n'aurait pas de sens ; la seule sémantique acceptable consiste à utiliser une comparaison champ à champ, de sorte que, dans ce cas, `x = y` aura la même valeur que `equal (x, y)`.

"Sémantique fixe des caractéristiques de copie, de clonage et d'égalité", page 565.

> Il est possible, comme nous le verrons lors de l'étude de l'héritage, d'adapter la sémantique de *equal* pour prendre en compte une notion spécifique d'égalité pour les instances d'une classe donnée. Cela n'a pas d'effet sur la sémantique de = qui, pour des raisons de sûreté et de simplicité, est toujours celle de la fonction originale *standard_equal*.

La règle de base pour l'attachement et la comparaison est, alors, résumée par l'observation suivante :

> Un attachement de *y* à *x* est une copie d'objet *x* si *x* et *y* ont des types expansés (ce qui inclut tout type de base). C'est un attachement par référence si *x* et *y* ont des types références.
>
> De même, un test d'égalité ou d'inégalité `x = y` ou `x /= y` est une comparaison d'objets si *x* et *y* ont des types expansés ; c'est une comparaison de références si *x* et *y* ont des types références.

Attachements hybrides

Dans les cas que nous avons vus jusqu'à présent, les types source et cible d'un attachement étaient de la même catégorie — tous deux expansés ou tous deux références. Que se passe-t-il s'ils sont de catégories différentes ?

Voir le chapitre 12, en particulier "Sources d'exceptions", page 400.

Considérons d'abord `x := y` où la cible *x* a un type expansé et la source *y* a un type référence. Puisque l'affectation par référence n'a pas de sens pour *x*, la seule sémantique acceptable pour cet attachement est la sémantique de copie : copier les champs de l'objet attaché à *y* sur les

champs correspondants de l'objet attaché à *x*. C'est effectivement l'effet de l'affectation dans ce cas ; mais cela n'a de sens que si *y* n'est pas vide au moment de l'exécution (sinon il n'y a pas d'objet attaché). Si *y* est vide, le résultat sera de déclencher une exception. L'effet des exceptions et la spécification de la manière de poursuivre après une exception seront étudiés dans un prochain chapitre.

> Pour un *x* expansé, le test *x = Void* ne crée aucun événement anormal ; il renvoie simplement le résultat faux. Mais il n'y a aucun moyen de définir une sémantique acceptable pour l'affectation *x := Void*, et donc toute tentative d'exécuter celle-ci causera une exception.

Considérez maintenant l'autre cas : *x := y* où *x* a un type référence et *y* un type expansé. Alors, à l'exécution, *y* est toujours attaché à un objet, que nous pouvons appeler OY, et l'attachement devrait aussi attacher *x* à un objet. Une possibilité serait d'attacher *x* à OY. Cette convention, cependant, introduirait la possibilité de référence à des sous-objets, comme dans la routine *reattach* ci-dessous :

```
class C feature
      ...
end
class COMPOSITE2 feature
      x: C
      y: expanded C

      reattach is
            do x := y end
end
```

Si, comme nous l'avons suggéré plus tôt, nous interdisons les références aux sous-objets, nous pouvons, dans ce cas, définir que l'attachement se fait sur un **clone** de OY. Cela sera, en conséquence, l'effet de l'attachement d'une source expansée à une cible référence : attacher la cible à un clone de la source.

La table suivante résume la sémantique de l'attachement dans les cas étudiés :

Type de source *y* → ↓ **Type de cible** *X*	**Référence**	**Expansée**
Référence	Attachement par référence	Clone ; effet de *x := clone (y)*
Expansée	Copie ; effet de *x.copy (y)* (échouera si *y* est vide)	Copie ; effet de *x.copy (y)*

Effet de l'attachement *x := y*

Pour autoriser les références aux sous-objets, il suffirait de remplacer la sémantique à base de clone définie dans la case supérieure droite par la sémantique de l'attachement par référence.

Comparaison d'égalité

La sémantique de la comparaison par égalité (les signes = et /=) devrait être compatible avec la sémantique de l'attachement : si *y /= z* est vrai et que vous exécutiez *x := y*, alors *x = y* et *x /= z* devraient tous deux être vrais immédiatement après l'affectation.

Outre =, nous avons vu qu'il existe une opération *equal* applicable aux objets. La disponibilité de l'une ou l'autre de ces opérations dépend des circonstances :

E1 • Si *x* et *y* sont des références, vous pouvez tester à la fois l'égalité des références et, si les références ne sont pas vides, l'égalité des objets. Nous avons défini l'opération *x = y* comme désignant, dans ce cas, l'égalité des références. La fonction *equal* a été introduite pour spécifier l'égalité d'objets ; pour des raisons de complétude, elle s'utilise également quand *x* ou *y* est vide (renvoyant vrai dans le seul cas où elles le sont toutes deux).

E2 • Si *x* et *y* sont expansées, la seule opération ayant un sens est la comparaison d'objets.

E3 • Si *x* est une référence et *y* est expansé, l'égalité d'objets est également la seule opération sensée — étendue pour prendre en compte un *x* vide, auquel cas elle renverra faux, puisque *y* ne peut être vide.

Cette analyse nous précise la façon dont il faut interpréter = dans tous les cas. Pour une comparaison d'objets, *equal* est toujours utilisable, étendue pour prendre en compte les cas dans lesquels l'un ou les deux opérandes est vide. = sert à appliquer la comparaison par référence quand cela a un sens, utilisant *equal* dans les autres cas :

Sens de la comparaison *x = y*

Type de source *y* → / ↓ **Type de cible** *X*	**Référence**	**Expansée**
Référence	Comparaison de références	*equal* (*x*, *y*) Comparaison d'objets si *x* n'est pas vide, faux si *x* est vide.
Expansée	*equal* (*x*, *y*) Comparaison d'objets si *y* n'est pas vide, faux si *y* est vide.	*equal* (*x*, *y*) Comparaison d'objets.

En comparant avec la table précédente, vous pouvez remarquer que = et /= sont effectivement compatibles avec := dans le sens défini ci-dessus. Rappelez-vous, en particulier, que *equal* (*x*, *y*) sera vrai après *x* := *clone* (*y*) ou *x*.*copy* (*y*).

Le problème que nous venons de résoudre se pose dans tout langage qui contient des pointeurs ou des références (comme Pascal, Ada, Modula-2, C, Lisp, etc.), mais il est particulièrement sensible dans un langage orienté objet dans lequel tous les types qui ne sont pas de base sont des types références ; de plus, pour des raisons évoquées dans la section de discussion, la syntaxe n'indique pas explicitement qu'il s'agit de références et nous devons donc être particulièrement prudents.

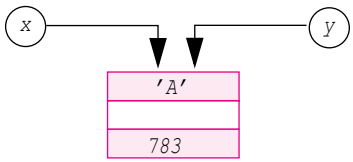
8.9 UTILISER LES RÉFÉRENCES : BÉNÉFICES ET DANGERS

Deux propriétés du modèle d'exécution introduit dans les sections précédentes méritent d'être examinées plus avant. L'une est le rôle important des références ; l'autre est la sémantique duale des opérations de base comme l'affectation, le passage d'argument et les tests d'égalité qui, comme nous l'avons vu, ont des effets différents pour des opérandes références et expansés.

Alias dynamique

Si *x* et *y* sont des types références et si *y* n'est pas vide, l'affectation `x := y`, ou l'attachement correspondant dans un appel, rend *x* et *y* attachés au même objet.

Partage résultant d'un attachement



Le résultat est de lier *x* et *y* de manière durable (jusqu'à une nouvelle affectation à l'un d'entre eux). En particulier, une opération de la forme *x*.*f*, où *f* est une caractéristique de la classe correspondante, aura le même effet que *y*.*f*, puisqu'elles affectent le même objet.

L'attachement de *x* au même objet que *y* s'appelle l'alias dynamique : alias, car l'affectation rend un objet accessible via deux références, comme une personne connue sous deux noms ; dynamique, car l'alias a lieu à l'exécution.

> L'alias statique, quand un texte logiciel spécifie que deux noms désigneront toujours la même valeur quoi qu'il arrive à l'exécution, est possible dans certains langages de programmation : la directive *EQUIVALENCE* de Fortran indique que deux variables désigneront toujours le contenu de la même adresse mémoire ; et la directive `#define x y` du processeur C spécifie que toute occurrence ultérieure de *x* dans le texte du programme veut dire exactement la même chose que *y*.

Du fait de l'alias dynamique, les opérations d'attachement ont un effet beaucoup plus durable sur les entités de type référence que sur celles de type expansé. Si *x* et *y* sont de type *INTEGER*, un cas particulier de type expansé, l'affectation `x := y` ne fait que rendre la valeur de *x* égale à celle de *y* ; mais elle ne lie pas durablement *x* et *y*. Pour des types références, l'affectation fait que *x* et *y* deviennent alias du même objet.

La sémantique de l'alias

Une conséquence un peu choquante de l'alias (statique ou dynamique) est qu'une opération peut affecter une entité qu'elle ne cite même pas.

Les modèles de calcul qui n'introduisent pas d'alias bénéficient d'une propriété agréable : la correction d'extraits comme :

[PAS DE SURPRISE]

```
        -- Supposez ici que P (y) est vérifié.
x := y
C (x)
        -- Alors P (y) est toujours vérifié ici.
```

Cet exemple suppose que `P (y)` est une propriété arbitraire de *y* et `C (x)` une opération dont la description textuelle dans le logiciel peut contenir *x* mais pas *y*. Être correct veut dire ici que la propriété "PAS DE SURPRISE" exprimée par les commentaires est effectivement vérifiée : si `P (y)` est initialement vraie, aucune action sur *x* ne peut invalider cette propriété. Une opération sur *x* n'affecte aucune propriété de *y*.

Avec des entités de types expansés, la propriété PAS DE SURPRISE est effectivement vérifiée. Voici un exemple typique, qui suppose que *x* et *y* sont de type *INTEGER* :

```
-- Supposez ici que y >= 0.
x := y
x := -1
-- Alors ici y >= 0 est toujours vrai.
```

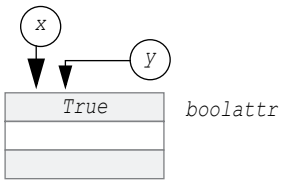
L'affectation à `x` ne peut, en aucune manière, dans ce cas, avoir d'effet sur `y`. Mais, maintenant, considérez un cas similaire mettant en jeu l'alias dynamique. Soient `x` et `y` de type `C`, où la classe `C` est de la forme :

```
class C feature
        boolattr: BOOLEAN
                -- Attribut booléen, modélisant une propriété de l'objet.
        set_true is
                -- Mettre boolattr à vrai.
            do
                boolattr := True
            end
        ... Autres caractéristiques...
end
```

Supposez que `y` soit de type `C` et que sa valeur à un moment de l'exécution ne soit pas vide. Alors, l'instance suivante du schéma ci-dessus viole la propriété PAS DE SURPRISE :

[SURPRISE, SURPRISE!]

```
        -- Supposez que y.boolattr est faux.
x := y
        -- Ici, y.boolattr est encore faux.
x.set_true
        -- Mais ici y.boolattr est vrai !
```



La dernière instruction de cet extrait ne fait, en aucune façon, intervenir `y` ; cependant, un de ses effets est de changer les propriétés de `y`, comme l'indique le commentaire final.

Amadouer l'alias dynamique

Après avoir vu les conséquences néfastes de l'affectation de référence et de l'alias dynamique, on peut légitimement se demander pourquoi nous devrions garder un tel système dans notre modèle de calcul.

La réponse est double — à la fois théorique et pratique :

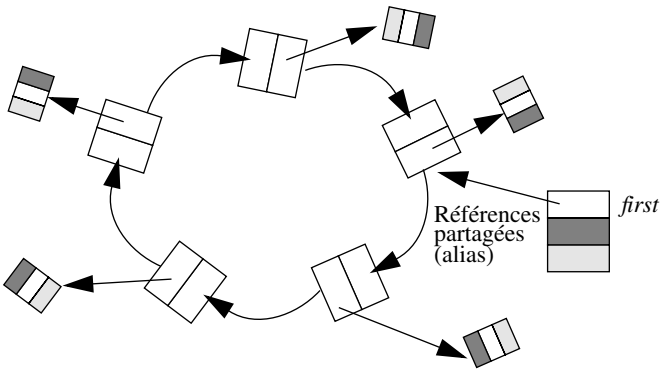
- Nous avons besoin des affectations de référence si nous voulons bénéficier de la puissance totale de la méthode orientée objet, en particulier pour écrire des structures de données complexes. Nous devons être sûrs que nos outils seront suffisamment généraux pour répondre à nos besoins de modélisation.
- Dans la pratique de la construction de logiciel orienté objet, l'encapsulation permet d'éviter les dangers provenant de la manipulation des références.

Examinons tour à tour ces deux aspects importants.

Alias dans le logiciel et ailleurs

La première observation découle de ce que, parmi les structures de données dont nous aurons besoin, bon nombre utilisent des références et profitent du partage de références. Certaines structures de données standard contiennent, par exemple, des éléments circulairement chaînés,

qui ne peuvent pas être implémentés sans référence. Pour représenter des structures de liste et d'arbre, il est souvent pratique de mettre dans chaque noeud une référence à son voisin ou parent. La figure ci-dessous illustre une représentation de liste circulaire combinant ces deux idées. Ouvrez n'importe quel livre décrivant les principes des structures de données et algorithmes, comme ceux utilisés dans les cours d'introduction à l'informatique, et vous trouverez nombre d'exemples similaires. Avec la technologie objet, nous voudrons, bien évidemment, utiliser des structures encore plus sophistiquées.



Une liste circulaire chaînée

En fait, le besoin de référence, d'attachement de référence et de partage de référence se présente déjà avec des structures de données relativement simples. Rappelez-vous les classes utilisées ci-dessus pour décrire les livres ; une des variantes était :

```
class BOOK3 feature
        ... Autres caractéristiques; ...
        author: WRITER
end
```

Page 227.

Ici, la nécessité de partage de référence est une simple conséquence de la propriété selon laquelle plusieurs livres peuvent avoir le même auteur. Plusieurs exemples de ce chapitre utilisent également le partage ; dans le cas `PERSON`, plusieurs personnes peuvent avoir le même propriétaire. La question, comme nous l'avons déjà noté, relève de la puissance de modélisation, pas simplement des besoins d'implémentation.

Ainsi, si `b1` et `b2` sont des instances de `BOOK3` ayant le même auteur, nous sommes en présence d'alias : `b1.author` et `b2.author` sont deux références attachées au même objet, et utiliser l'une quelconque d'entre elles comme cible d'un appel de caractéristique aura exactement le même effet qu'utiliser l'autre. Vu ainsi, l'alias dynamique apparaît moins comme un service logiciel qu'un fait de l'existence, le prix à payer pour pouvoir faire référence à des choses via plusieurs noms.

Il est d'ailleurs facile de rencontrer des violations de la propriété PAS DE SURPRISE ci-dessus sans même aborder le domaine du logiciel. Considérez les propriété et opération suivantes, définies pour tout livre `b` :

- `NOT_NOBEL (b)` correspond à : "L'auteur de `b` n'a jamais reçu le prix Nobel".
- `NOBELIZE (b)` correspond à : "Donner le prix Nobel à l'auteur de `b`".

Stendhal a, bien sûr, vécu avant la création du prix — et ne l'aurait probablement pas reçu de toute façon ; il n'est même pas rentré à l'Académie.

Supposez maintenant que `rb` corresponde au livre *Le Rouge et le Noir* et `cp` désigne *La Chartreuse de Parme*. Alors, ce qui suit est une déduction correcte :

[SURPRISE A OSLO]

```
        -- Supposez ici que NOT_NOBEL (rb) soit vrai.
NOBELIZE (cp)
        -- Alors ici NOT_NOBEL (rb) n'est plus vrai!
```

Une opération sur `cp` a changé une propriété d'une entité différente, `rb`, qui n'est même pas nommée dans l'instruction ! Les conséquences sur `rb` peuvent, en pratique, être tout à fait significatives (avec un prix Nobel, un livre épuisé sera réimprimé, son prix peut augmenter, etc.). Dans cet exemple non logiciel, on observe la même chose que quand l'opération `x.set_true`, dans l'exemple logiciel précédent, produisait un effet important sur `y` sans faire référence à `y`.

Ainsi, l'alias dynamique n'est pas uniquement une conséquence des "bidouilles" des programmeurs avec les références ou les pointeurs. C'est une conséquence de la capacité que nous avons de `nommer` les choses (des "objets" dans le sens le plus général de ce mot) et de donner plusieurs noms à une même chose. En rhétorique classique, cela s'appelait la *polyonymie,* comme dans l'utilisation de "Cybèle", "Déméter" et "Cérès" pour désigner la même déesse, et l'*antonomase*, qui permet de faire référence à un objet grâce à des phrases indirectes, comme dans "la belle fille d'Agammemnon" pour Hélène de Troie. La polyonymie, l'antonomase et l'alias dynamique qui en découle ne sont pas limités aux dieux et aux héros ; si, dans la cafétéria, vous entendez deux énoncés provenant de conversations séparées, l'un disant que l'époux du vice-président en charge de la technologie vient de recevoir une grosse promotion, et l'autre que la compagnie a licencié son comptable, vous ne vous rendrez pas compte de la contradiction — à moins que vous ne sachiez que le comptable est le mari du vice-président.

Encapsuler les manipulations de références

Nous avons donc suffisamment prouvé que tout cadre réaliste de modélisation et de développement logiciel doit fournir la notion de référence et, en conséquence, d'alias dynamique. Comment allons-nous composer avec les conséquences désagréables de ces mécanismes ? L'incapacité à assurer la propriété PAS DE SURPRISE montre combien les références et l'alias mettent en péril notre capacité de raisonner de manière systématique sur notre logiciel, c'est-à-dire d'inférer des propriétés à l'exécution du logiciel, de manière sûre et simple, en examinant le texte du logiciel.

Pour trouver une réponse, il nous faut comprendre, d'abord, en quoi cette question est spécifique à la méthode orientée objet. Si vous avez l'habitude des langages de programmation comme Pascal, C, PL/I, Ada et Lisp, vous aurez probablement remarqué qu'une grande partie de l'exposé précédent s'applique également à ceux-ci. Ils offrent tous la possibilité d'allouer des objets dynamiquement (quoique, en C, la fonction correspondante, `malloc`, se trouve dans une librairie plutôt que dans le langage proprement dit) et d'introduire des références à d'autres objets. Le niveau d'abstraction des mécanismes varie de manière significative suivant les langages : les pointeurs de PL/I et C et sont des adresses mémoires hâtivement maquillées ; Pascal et Ada utilisent des règles de typage pour camoufler leurs pointeurs de manière plus respectable, bien qu'ils ne demandent qu'à retourner à leur état d'origine.

Qu'y a-t-il donc de si neuf dans le développement orienté objet ? La réponse ne se trouve pas dans la puissance théorique de la méthode (dont les structures à l'exécution sont semblables à

celles de Pascal ou Ada, hormis l'importante différence du ramasse-miettes, étudié dans le prochain chapitre), mais dans la pratique de la construction logicielle. Le développement OO repose sur la réutilisation. En particulier, tout projet dans lequel plusieurs classes d'application effectuent des manipulations complexes (comme des manipulations de références) utilise mal l'approche orientée objet. De telles opérations devraient être encapsulées une fois pour toutes dans des classes bibliothèques.

Indépendamment du domaine d'application, si un système contient des structures d'objets qui nécessitent des opérations complexes sur des références, la grande majorité de ces structures ne seront pas spécifiques à l'application, mais seront simplement des instances de structures bien connues et utilisées fréquemment comme des listes de genres variés, d'arbres sous diverses représentations, de graphes, de tables de hachage et autres. Dans un bon environnement OO, une bibliothèque sera facilement accessible, offrant de nombreuses implémentations de ces structures ; l'annexe A en montrera un exemple, la bibliothèque Base. Les classes d'une telle bibliothèque peuvent contenir de nombreuses opérations sur références (pensez, par exemple, aux manipulations de références que demandent l'insertion ou l'élimination d'un élément dans une liste chaînée, ou d'un noeud dans un arbre qui utilise une représentation chaînée). La bibliothèque devrait être conçue et validée avec soin de façon à traiter ces problèmes difficiles une fois pour toutes.

Si vous avez besoin, lors de la construction d'une application, de structures d'objets complexes qui ne sont pas implémentées de manière adéquate par les bibliothèques disponibles, vous devriez les considérer comme de nouvelles classes d'application générale. Vous devriez les concevoir et les valider soigneusement, dans la perspective d'une utilisation ultérieure dans une bibliothèque. En utilisant la terminologie introduite dans un chapitre précédent, ce cas illustre l'évolution d'une attitude de consommateur de réutilisation vers une attitude de producteur de réutilisation.

"Réutiliser les consommateurs, réutiliser les producteurs", page 72.

Les manipulations de références qui subsistent dans les classes dépendant de l'application devraient se limiter à des opérations simples et sûres. (Les notes bibliographiques citent un article de Suzuki qui approfondit cette idée.)

8.10 DISCUSSION

Ce chapitre a introduit un certain nombre de règles et de notations pour manipuler des objets et les entités correspondantes. Vous avez peut-être été surpris par certaines de ces conventions. Aussi est-il utile de conclure notre exploration des objets et de leurs propriétés en réexaminant les problèmes abordés et les raisons derrière les choix qui ont été faits. J'espère que vous serez, *in fine*, d'accord avec ces choix, quoique l'objectif le plus important de cette discussion soit de s'assurer que vous maîtrisez totalement les problèmes sous-jacents de façon que, si vous adoptez une solution différente, vous la choisissiez en connaissance de cause.

Conventions graphiques

Pour nous échauffer, débutons avec une petite question de notation, un détail, assurément, mais le logiciel est parfois une affaire de détails. Ce détail particulier est l'ensemble des conventions utilisées pour illustrer les classes et les objets dans les représentations graphiques.

"Le moule et l'instance", page 171.

Le chapitre précédent a insisté sur le fait qu'il était important de ne pas confondre les notions de classe et d'objet. En conséquence, leurs représentations graphiques sont différentes. Les objets sont représentés par des rectangles. Les classes, quand elles apparaissent dans les diagrammes d'architecture de système, sont représentées par des ellipses (connectées par des flèches qui représentent les relations entre classes : une simple flèche pour la relation d'héritage, une double flèche pour la relation client).

Les représentations de classe et d'objet apparaissent dans des contextes différents : une ellipse de classe fera partie d'un diagramme représentant la structure d'un système logiciel ; un rectangle d'objet fera partie d'un diagramme représentant un instantané de l'état d'un système durant son exécution. Puisque ces deux genres de diagrammes correspondent à des objectifs complètement différents, il est habituellement inutile, dans les représentations sur papier comme dans ce livre, de faire apparaître à la fois les représentations de classe et d'objet dans le même contexte. Mais la situation est différente dans un outil CASE interactif : durant l'exécution d'un système logiciel, il se peut que vous souhaitiez (par exemple pour des raisons de débogage) regarder un objet, puis afficher sa classe génératrice pour en examiner les caractéristiques, les parents ou d'autres propriétés de cette classe.

À propos de BON, voir les notes bibliographiques et le chapitre 27.

Les conventions graphiques utilisées pour les classes et les objets sont compatibles avec le standard établi par la méthode BON de Nerson et Waldén. Dans BON (Business Object Notation), conçue pour les outils CASE interactifs et la documentation papier, les bulles des classes peuvent être étendues verticalement de façon à exposer les caractéristiques d'une classe, son invariant, ses mots d'indexation et d'autres propriétés.

Comme dans tout choix de représentation graphique, il n'y a pas de justification absolue pour les conventions utilisées dans BON et dans ce livre. Mais, si les symboles graphiques à notre disposition sont des ellipses et des rectangles, et si les éléments qui doivent être représentés sont des classes et des objets, il paraît alors préférable d'affecter les rectangles aux objets : un objet est un ensemble de champs, et nous pouvons représenter chaque champ par un petit rectangle et les coller ensemble pour former un ensemble de champs représenté par un rectangle plus grand correspondant à un objet.

Une autre convention, illustrée par les figures de ce chapitre, consiste à faire apparaître en ombré les champs expansés, alors que les champs de références sont blancs ; les sous-objets apparaissent comme des rectangles plus petits qui contiennent leurs propres champs. Toutes ces conventions découlent de la décision d'utiliser des rectangles pour les objets.

Tiré de la critique de "Object-Oriented Analysis and Design", par Martin et Odell, dans OOPS (lettre d'information du groupe OO de la British Computer Society), 16, Hiver 1992, pages 35-37.

Sur un ton plus léger, la tentation est grande de citer l'argument non scientifique suivant, tiré de la critique par Ian Graham d'un livre d'analyse OO qui utilise une convention différente :

> Je n'aime pas non plus représenter les classes par des triangles aigus. J'aime à penser que les instances ont des coins aigus, car, si vous les laissez tomber sur votre pied, cela fait mal, tandis que les classes ne peuvent blesser personne et ont donc des coins arrondis.

Références et valeurs simples

Une question syntaxique importante est de savoir si nous devons différencier références et valeurs simples. Comme on l'a vu, l'affectation et le test d'égalité ont des sens différents pour les références et les valeurs de types expansés — ces dernières contenant les valeurs des types de base : entier et autres. Pourtant, les mêmes symboles sont utilisés dans les deux cas : `:=`, `=`,

/=. N'est-ce pas dangereux ? Ne serait-il pas préférable d'utiliser des ensembles différents de symboles pour rappeler au lecteur que les sens sont différents ?

Utiliser deux ensembles de symboles était, effectivement, la solution de Simula 67. En transposant légèrement la notation pour la rendre compatible avec celle de ce livre, la solution Simula consiste à déclarer une entité de type référence `C` comme :

Simula, évoqué dans le chapitre 35, simplifie **reference** *en* **ref**.

```
x: reference C
```

où le mot-clé **reference** rappelle au lecteur que les instances de `x` seront des références. En supposant les déclarations :

```
m, n: INTEGER
x, y: reference C
```

des notations différentes sont alors utilisées pour les opérations sur les types simple et référence, comme suit :

OPÉRATION	OPÉRANDES EXPANSÉS	OPÉRANDES RÉFÉRENCES
Affectation	`m := n`	`x :- y`
Test d'égalité	`m = n`	`x == y`
Test d'inégalité	`m /= n`	`x =/= y`

Notations à la Simula pour les opérations sur les références et les valeurs expansées

Les conventions de Simula éliminent toute ambiguïté. Pourquoi ne pas les conserver ? Parce qu'en pratique, elles causent plus de mal que de bien. Les problèmes surgissent de manière anodine : des erreurs de typage. Les deux ensembles de symboles sont si proches que l'on tend à faire des confusions syntaxiques, comme d'utiliser `:=` au lieu de `:-`. De telles erreurs seront détectées par le compilateur. Mais, bien que les restrictions vérifiables par compilateur et présentes dans les langages de programmation aient pour but d'aider les programmeurs, les vérifications sont ici inutiles : soit vous connaissez la différence entre une sémantique par référence et une sémantique par valeur, auquel cas l'obligation de démontrer à nouveau, chaque fois que vous écrivez une affectation ou une égalité, que vous comprenez cette différence est plutôt pénible ; soit vous ne comprenez pas la différence, et le message du compilateur ne vous aidera alors pas beaucoup !

L'aspect remarquable de la convention de Simula est que vous n'avez, en fait, pas le choix : pour les références, aucune construction prédéfinie n'est disponible qui vous donnerait une sémantique par valeur. Il aurait pu paraître raisonnable de définir deux ensembles d'opérations sur les entités `a` et `b` de type référence :

- `a :- b` pour l'affectation de référence, et `a == b` pour la comparaison de références.
- `a := b` pour l'affectation de copie (l'équivalent de `a := clone (b)` ou `a.copy (b)` dans notre notation), et `a = b` pour la comparaison d'objets (l'équivalent de notre `equal (a, b)`).

Mais ce n'est pas le cas ; pour des opérandes de types références, à une seule exception près, Simula ne fournit que le premier ensemble d'opérations, et toute tentative d'utilisation de `:=` ou `=` produira une erreur syntaxique. Si vous avez besoin des opérations du deuxième ensemble (copie ou clone, comparaison d'objets), vous devez écrire des routines spécifiques qui correspondent à nos `clone`, `copy` et `equal`, et ce pour chaque classe cible. (L'exception est le type

TEXT, représentant des chaînes de caractères, pour lequel Simula offre les deux ensembles d'opérations.)

Après mûre réflexion, d'ailleurs, l'idée d'avoir deux ensembles d'opérations pour tous les types références ne paraît pas si brillante. Cela voudrait dire qu'une simple distraction, comme de taper := à la place de :-, deviendrait maintenant indétectable au compilateur, mais produirait un effet tout à fait différent de ce qu'attend le programmeur, par exemple un *clone* là où une affectation de référence était prévue.

Suite de cette analyse, la notation de ce livre utilise une convention différente de celle de Simula : les mêmes symboles s'appliquent aux types expansé et référence, avec des sémantiques différentes (valeur dans un cas, référence dans l'autre). Vous pouvez obtenir l'effet de la sémantique par valeur pour des objets de types références en utilisant des routines prédéfinies, disponibles pour tous les types :

- *a* := *clone* (*b*) ou *a*.*copy* (*b*) pour l'affectation d'objet,
- *equal* (*a*, *b*) pour la comparaison (champ à champ) des objets.

Ces notations sont suffisamment différentes de leur équivalent par référence (:= et = respectivement) pour éviter tout risque de confusion.

Au-delà des aspects purement syntaxiques, la question est intéressante, car elle illustre certains des compromis qu'impose la conception des langages quand un équilibre doit être trouvé entre critères en conflit. Un critère, qui a prévalu dans le cas Simula, peut être énoncé comme suit : "Des concepts différents doivent être exprimés par des symboles différents".
Mais, les forces opposées qui ont dominé dans la conception de notre notation suggèrent :

- "Évitez d'ennuyer le développeur logiciel."
- "Pesez soigneusement toute nouvelle restriction en regard des bénéfices réels qu'elle apportera en terme de sécurité ou d'autres facteurs de qualité." Ici, la restriction est l'interdiction d'utiliser les opérateurs := et autres sur des références.
- "Les opérations les plus courantes doivent être exprimées par des notations courtes et simples." L'application de ce principe demande un certain soin, car le concepteur du langage peut se tromper dans la détermination des cas qui seront les plus courants. Mais, dans le cas présent, il semble clair que, sur les entités de types expansés (comme *INTEGER*), les affectations et comparaisons par valeur sont les opérations les plus fréquentes, tandis que, pour les entités de références, les affectations et les comparaisons par référence sont plus fréquentes que les clonages, copies et comparaisons d'objets. Il est donc approprié d'utiliser := et = pour les opérations fondamentales dans les deux cas.
- "Pour que le langage reste concis et simple, n'introduisez pas de nouvelles notations si elles ne sont pas absolument nécessaires". Tel est le cas, comme dans cet exemple, lorsque des notations existantes suffisent et qu'il n'y a pas de risque de confusion.
- "Si vous savez qu'il existe un risque sérieux de confusion entre deux services, rendez les notations correspondantes aussi différentes que possible." Cela nous amène à éviter d'offrir à la fois :- et := pour les mêmes opérandes avec des sémantiques différentes.

Une raison supplémentaire intervient dans le cas présent, bien qu'elle mette en jeu des mécanismes que nous n'avons pas encore étudiés. Dans les prochains chapitres, nous apprendrons à écrire des classes génériques comme *LIST* [*G*] , où *G*, appelé paramètre générique formel, représente un type arbitraire. Une telle classe peut manipuler des entités de type *G* et les utiliser dans des affectations et des tests d'égalité. Les clients qui doivent utiliser la classe

le feront en fournissant un type qui servira de paramètre générique réel ; par exemple, ils peuvent utiliser *LIST* [*INTEGER*] ou *LIST* [*POINT*] . Comme l'indiquent ces exemples, le paramètre générique réel peut tout aussi bien être un type expansé (comme dans le premier cas) qu'un type référence (comme dans le second cas). Dans les routines d'une telle classe générique, si *a* et *b* sont de type *G*, il est souvent utile d'utiliser des affectations de la forme *a* := *b* ou des tests de la forme *a* = *b* avec l'intention d'obtenir une sémantique par valeur si le paramètre générique est expansé (comme avec *INTEGER*) et une sémantique par référence si c'est un type référence (comme avec *POINT*).

> Un exemple de routine qui a besoin d'un tel comportement dual est une procédure d'insertion d'un élément *x* dans une liste. La procédure crée une nouvelle cellule de liste ; si *x* est un entier, la cellule doit contenir une copie de cet entier, mais, si *x* est une référence à un objet, la cellule contiendra une référence à ce même objet.

Dans un tel cas, les règles définies ci-dessus nous permettent d'obtenir le comportement dual désiré, ce qui serait impossible si une syntaxe différente avait été requise pour les deux sortes de sémantique. Si, par ailleurs, vous voulez un comportement unique identique dans tous les cas, vous pouvez également le spécifier : ce comportement ne peut être qu'une sémantique par valeur (puisque la sémantique par référence n'a pas de sens pour les types expansés) ; ainsi, dans les routines, vous ne devriez pas utiliser := et =, mais *clone* (ou *copy*) et *equal*.

La forme des opérations de clonage et d'égalité

Un aspect stylistique mineur qui peut vous avoir surpris est la forme utilisée pour appeler les routines *clone* et *equal*. Les notations :

```
clone (x)
equal (x, y)
```

ne paraissent pas, de prime abord, très OO ; une lecture dogmatique du chapitre précédent suggérerait des conventions plus dans la ligne de ce qui était appelé alors le style orienté objet du calcul ; par exemple :

"LE STYLE ORIENTÉ OBJET DE CALCUL", 7.7, page 184.

```
x.twin
x.is_equal (y)
```

Dans une version très antérieure de cette notation, les conventions utilisées étaient, de fait, celles-ci. Mais elles posent le problème des références vides. Un appel de caractéristique de la forme *x.f*(...) ne peut pas être exécuté correctement si, à l'exécution, la valeur de *x* est vide. (Dans ce cas, l'appel déclenchera une exception qui, sauf si la classe contient des clauses spécifiques pour traiter cette exception, entraînera la terminaison anormale de l'exécution du système complet.) Ainsi, le deuxième ensemble de conventions ne marcherait que pour les *x* non vides. Puisque, dans de nombreux cas, *x* peut, en fait, être vide, cela voudrait dire que la plupart des utilisations de *twin* seraient, en pratique, de la forme :

```
if x = Void then
        z := Void
else
        z := x.twin
end
```

et la plupart des utilisations de `is_equal` de la forme :

`and then` est une variante de `and`. Voir "Opérateurs booléens non stricts", page 439.

```
if
        ((x = Void) and (y = Void)) or
        ((x /= Void) and then x.is_equal (y))
then
        ...
```

Il va sans dire que de telles conventions n'ont pas duré très longtemps. Nous nous sommes rapidement lassés d'écrire des expressions si compliquées — et encore plus d'avoir à affronter les conséquences (des erreurs à l'exécution) quand nous les avions oubliées. Les conventions qui ont été finalement retenues, décrites plus haut dans ce chapitre, possèdent la propriété agréable de donner les résultats escomptés quand `x` est vide : dans ce cas, `clone (x)` est une valeur vide et `equal (x, y)` est vrai si et seulement si `y` est vide aussi.

La procédure `copy`, appelée sous la forme `x.copy (y)`, ne soulève pas de problème particulier : elle demande que `x` (et aussi `y`) ne soient pas vides, mais cette exigence est acceptable, car elle est une conséquence de la sémantique de `copy`, qui copie un objet sur un autre, ce qui n'a donc pas de sens si un des objets n'existe pas. La condition sur `y`, qui sera expliquée dans un prochain chapitre, est décrite par une précondition officielle de `copy` et est donc présente, sous une forme claire, dans la documentation de cette procédure.

Il convient de noter qu'une fonction `is_equal`, introduite ci-dessus, existe. La raison en est qu'il est souvent pratique de définir des variantes spécifiques d'égalité, adaptées à une classe et qui remplacent la sémantique par défaut de la comparaison champ à champ. Pour obtenir cet effet, il suffit de redéfinir la fonction `is_equal` dans les classes désirées. La fonction `equal` est définie en termes de `is_equal` (grâce à l'expression indiquée ci-dessus pour illustrer l'utilisation de `is_equal`) et profitera donc de ces redéfinitions.

En ce qui concerne `clone`, il n'y a pas besoin de `twin`. En effet, `clone` est définie simplement comme une création plus un appel à `copy`. Ainsi, pour adapter le sens de `clone` aux besoins spécifiques d'une classe, il suffit de redéfinir la procédure `copy` pour cette classe ; `clone` suivra automatiquement.

Voir aussi "Sémantique fixe des caractéristiques de copie, de clonage et d'égalité", page 565.

Le statut des opérations universelles

Les derniers commentaires ont partiellement levé le voile sur une question qui avait peut-être déjà attiré votre attention : quel est le statut des opérations universelles `clone`, `copy`, `equal`, `is_equal`, `deep_clone`, `deep_equal` ?

"LA STRUCTURE GLOBALE D'HÉRITAGE", 16.2, page 562.

Bien qu'elles soient fondamentales en pratique, ces opérations ne sont pas des constructions du langage. Elles viennent d'une classe de bibliothèque Kernel, `ANY`, qui possède la propriété spéciale selon laquelle toute classe écrite par un développeur logiciel hérite automatiquement (directement ou indirectement) de `ANY`. C'est pourquoi il est possible de redéfinir les caractéristiques mentionnées pour implémenter une vue particulière de l'égalité ou de la copie.

Nous n'avons pas à nous préoccuper des détails ici, car ils seront étudiés avec l'héritage. Mais il est utile de savoir que, grâce au mécanisme d'héritage, nous pouvons compter sur des classes bibliothèques pour fournir des services qui seront alors rendus accessibles à toute classe — et que chaque classe pourra adapter pour répondre à ses propres objectifs spécifiques.

8.11 CONCEPTS CLÉS INTRODUITS DANS CE CHAPITRE

- Le calcul orienté objet est caractérisé par une structure à l'exécution très dynamique, où les objets sont plutôt créés à la demande qu'à l'avance.
- Les objets manipulés par le logiciel sont (d'habitude de manière indirecte) des modèles d'objets extérieurs. D'autres servent uniquement à la conception et à l'implémentation.
- Un objet est formé d'un certain nombre de valeurs appelées champs. Chaque champ correspond à un attribut du générateur de l'objet (la classe dont l'objet est une instance directe).
- Une valeur, et, en particulier, un champ d'un objet, est un objet ou une référence.
- Une référence est vide ou attachée à un objet. Le test `x = Void` indique dans quel cas on se trouve. Un appel comme `x.f (...)`, ayant pour cible `x`, ne peut être correctement exécuté que si `x` n'est pas vide.
- Si la déclaration d'une classe débute par `class C ...`, une entité déclarée de type `C` désignera une référence, qui peut devenir attachée à des instances de `C`. Si la déclaration débute par **`expanded class`** `D ...`, une entité déclarée de type `D` désignera un objet (une instance de `D`) et ne sera jamais vide.
- Les types de base (*BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL*, *DOUBLE*) sont définis par des classes expansées.
- Les déclarations expansées permettent également la définition d'objets composites : des objets ayant des sous-objets.
- Les structures d'objets peuvent contenir des chaînes cycliques de références.
- L'instruction de création `!! x` crée un objet, initialise ses champs avec des valeurs par défaut (vide pour les références et zéro pour les nombres) et attache `x` à celui-ci. Si la classe a défini des procédures de création, l'instruction réalisera, sous la forme `!! x.creatproc (...)`, toute initialisation spécifique souhaitée.
- Pour les entités de type référence, l'affectation (`:=`) et le test d'égalité (`=`) sont des opérations sur références. Pour les entités de type expansé, elles représentent des opérations de copie et de comparaison champ à champ. Elles fournissent également une sémantique appropriée dans le cas d'opérandes mixtes.
- Les opérations sur références créent de l'alias dynamique qui rend plus difficiles les raisonnements formels sur le logiciel. En pratique, la plupart des manipulations non triviales de références devraient être encapsulées dans des classes de bibliothèque.

8.12 NOTES BIBLIOGRAPHIQUES

La notion d'identité d'objet joue un rôle important dans les bases de données, en particulier celles qui sont orientées objet. Voir le chapitre 31 et ses notes bibliographiques.

Les conventions graphiques de la méthode BON (Business Object Notation), conçue par Jean-Marc Nerson et Kim Waldén, se trouve dans [Waldén 1995]. James McKim et Richard Bielak examinent les mérites des procédures multiples de création dans [Bielak 1994].

Les risques induits par des opérations sauvages sur référence ou pointeur ont inquiété les méthodologistes du logiciel depuis longtemps, suggérant inévitablement qu'elles sont l'équivalent pour les données de ce que représentent les détestables instructions `goto` pour le contrôle. Un article bizarrement peu connu de Nori Suzuki [Suzuki 1982] évalue si une

approche disciplinée, qui utiliserait des opérations de haut niveau (de la même façon que l'on évite les `goto` en utilisant uniquement les constructions de "programmation structurée" que sont la séquence, la conditionnelle et la boucle), pourrait éviter les problèmes d'alias dynamique. Bien que les résultats soient relativement décevants — selon l'aveu même de l'auteur —, l'article mérite d'être lu.

Je suis redevable à Ross Scaife, de l'université du Kentucky, pour son aide concernant les termes de rhétorique. Voir sa page *http://www.uky.edu/ArtsSciences/Classics/rhetoric.html*.

EXERCICES

E8.1 Livres et auteurs

En vous inspirant des divers exemples donnés dans ce chapitre, écrivez des classes `BOOK` et `WRITER` qui fournissent une présentation claire des livres et de leurs auteurs. Veillez à inclure les routines pertinentes (et pas seulement les attributs, comme cela a souvent été le cas dans ce chapitre).

E8.2 Personnes

Écrivez une classe `PERSON` représentant une notion simple de personne, avec les attributs de nom `name` (une `STRING`), `mother` pour la mère, `father` pour le père et `sibling` décrivant le frère ou la soeur plus âgé, si il ou elle existe. Vous ajouterez les routines qui donneront (respectivement) la liste des noms des ancêtres, les cousins directs uniquement, les cousins directs ou indirects, les oncles ou les tantes, les beaux-frères ou les belles-soeurs, les beaux-parents, etc. d'une personne donnée. **Suggestion** : Écrivez des procédures récursives (mais évitez une récursion infinie quand les relations, par exemple cousin direct ou indirect, sont cycliques).

E8.3 Conception de notation

Voir "Caractéristiques d'opérateurs", page 190 à propos des caractéristiques infixes et des opérateurs autorisés.

Supposez que vous utilisiez fréquemment des comparaisons de la forme `x.is_equal (y)` et que vous vouliez simplifier la notation pour profiter des caractéristiques infixes (applicables ici, puisque `is_equal` est une fonction à un argument). Avec une caractéristique infixe qui utilise un opérateur `§`, l'appel sera écrit `x § y`. Ce petit exercice vous demande d'inventer un symbole pour `§`, compatible avec les règles des opérateurs infixes. Il y a, bien sûr, plusieurs réponses possibles ; le choix sera, en partie (mais en partie seulement), une affaire de goût.

Suggestion : Le symbole devrait être facile à mémoriser et devrait suggérer d'une manière ou d'une autre l'égalité ; mais, et c'est peut-être encore plus important, il devrait être suffisamment différent de `=` pour éviter les erreurs. Ici, vous pouvez bénéficier de l'étude de C et C++ qui, en s'éloignant de la tradition mathématique, utilisent le symbole `=` pour l'affectation plutôt que pour la comparaison d'égalité, introduisant pour cette dernière opération un symbole similaire `==`. Deux éléments rendent le problème encore plus délicat : la règle qui permet de traiter une affectation comme une expression, dont la valeur est celle qui est affectée à la cible, et celle qui considère les valeurs entières comme étant des expressions booléennes, s'évaluant à vrai si elles sont non nulles, de telle manière que les compilateurs accepteront un texte de la forme :

```
if (x = y) then ...
```

bien que, dans la majorité des cas pratiques, il s'agisse d'une erreur (en utilisant `=` à la place de `==`) qui aura l'effet probablement incorrect d'affecter à `x` la valeur de `y`, renvoyant vrai si et seulement si cette valeur n'est pas nulle.