# Conception et programmation orientées objet

**Bertrand Meyer**

Traduit de l'anglais par Pierre Jouvelot

 ISBN : 978-2-212-12270-1

EYROLLES

# 24

# Bien utiliser l'héritage

Avoir découvert tous les détails techniques concernant l'héritage et ses mécanismes, comme nous l'avons fait dans la partie C, ne confère pas nécessairement la maîtrise de toutes les conséquences méthodologiques. De tous les aspects de la technologie objet, l'emploi de l'héritage est celui qui soulève le plus de questions ; les opinions tranchées sont légion, par exemple dans les groupes de discussion sur Internet, mais la littérature contient relativement peu de conseils précis et utiles.

Dans ce chapitre, nous pousserons plus loin notre réflexion sur la raison d'être de l'héritage, non pour le seul plaisir intellectuel, mais pour nous assurer que nous l'utiliserons au mieux lors de nos projets de développement logiciel. En particulier, nous essayerons de comprendre dans quelle mesure l'héritage diffère de l'autre relation entre modules que l'on trouve dans les structures de systèmes orientés objet, sa consoeur et rivale, la relation client : quand il convient d'utiliser l'une, l'autre, ou les deux. Après avoir défini les critères de base gouvernant l'utilisation de l'héritage — en identifiant, chemin faisant, les cas typiques où il est contre-indiqué — nous pourrons élaborer une classification de ses diverses utilisations légitimes, certaines largement acceptées (l'héritage de sous-typage), d'autres, comme l'héritage d'implémentation ou de service, plus controversées. Nous en profiterons pour essayer de profiter de l'expérience accumulée dans le domaine de la taxonomie, ou *systématique*, par les disciplines scientifiques plus anciennes.

## 24.1 COMMENT NE PAS UTILISER L'HÉRITAGE

Pour élaborer un principe méthodologique, il est souvent utile — comme on l'a souvent vu dans ce livre — d'étudier d'abord les choses à ne *pas* faire. Comprendre en quoi une idée est mauvaise aide à en suggérer des bonnes. Un poirier ne fleurira pas dans un climat perpétuellement trop chaud ; il lui faut le coup de fouet qu'apportent les frimas de l'hiver pour pleinement fleurir au printemps.

*Extraits de "Software Engineering" par Ian Sommerville, Fourth edition, Addison-Wesley, 1993.*

La secousse nous viendra, obligeamment, d'un livre qui a connu un très large succès et qui a été utilisé dans le monde entier pour enseigner le génie logiciel à plus d'élèves que tout autre ouvrage. Ayant déjà été réédité trois fois, il introduit certains aspects de l'orientation objet, y compris l'héritage multiple. En voici le début :

*L'héritage multiple permet à plusieurs objets de jouer le rôle d'objets de base, et est présent dans des langages orientés objet comme* [la notation du présent livre] [M 1988].

La référence bibliographique est celle de la première édition de ce livre. Si l'on met de côté l'utilisation malheureuse du terme "objet" à la place de classe, c'est un début prometteur. Le texte continue :

*Les caractéristiques de plusieurs classes d'objets différents*

(des classes, bien !)

*peuvent être combinées pour donner un nouvel objet.*

(raté). Arrive alors l'exemple de l'héritage multiple :

*Par exemple, supposons que nous ayons une classe d'objets* `CAR` *qui encapsule l'information concernant des voitures et une classe d'objets* `PERSON` *qui encapsule l'information concernant des personnes. Nous pourrions utiliser celles-ci pour définir*

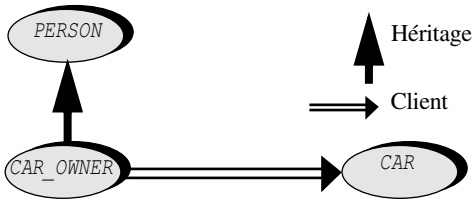
(est-ce que nos angoisses les plus folles vont s'avérer justifiées ?)

*une nouvelle classe d'objet propriétaire de voiture* `CAR-OWNER`*, qui combine les attributs de* `CAR` *et* `PERSON`*.*

(Et oui !) On nous propose de considérer que tout objet `CAR-OWNER` peut, à la fois, être vu non seulement comme une personne mais aussi comme une voiture. Pour tous ceux qui auront étudié l'héritage, ne serait-ce qu'à un niveau élémentaire, cela sera une surprise.

Comme vous l'aurez sans aucun doute compris, la relation à utiliser dans le second cas était la relation client, et non l'héritage : un propriétaire de voiture *est* une personne, mais *a* une voiture. En image :

***Un modèle correct***




De manière formelle :

```
class CAR_OWNER inherit
        PERSON
feature
        my_car: CAR
        ...
end -- class CAR_OWNER
```

Dans le texte cité, les deux liens utilisent la relation d'héritage. Le plus intéressant apparaît un peu plus loin dans le texte, quand l'auteur conseille au lecteur de considérer l'héritage avec prudence :

*L'adaptation par héritage conduit souvent à hériter de fonctionnalités superflues, ce qui rend les composants inefficaces et volumineux.*

Volumineux, pour le moins ; pensez au pauvre propriétaire de voiture écrasé par son toit, son moteur et son carburateur, sans même parler des quatre roues, plus celle de secours. Cette vision a pu être influencée par une des phrases imagées de l'argot australien à propos d'un propriétaire de voiture qui ressemble *vraiment* à sa voiture (voir ci-contre).

L'héritage n'est pas un concept trivial, et il nous faut pardonner à l'auteur de cet extrait, qui s'était, peut-être, un peu éloigné de son domaine de compétence. Mais cet exemple a un avantage important en pratique, sans compter celui de nous faire croire plus intelligent : il nous rappelle la règle de base de l'héritage.

En d'autres termes, nous devons être capables de convaincre, d'une manière ou d'une autre, quelqu'un — ne serait-ce que nous-mêmes — que "tout `B` est un `A`" (d'où le nom :"est-un").


***"Il a la tête d'une Austin Mini avec les portes ouvertes".***

Dessin de Geoff Hocking ; extrait de *The Dictionary of Aussie Slang*, The Five Mile Press, Melbourne, Australia, reproduit avec autorisation.

***Règle d'héritage "est-un"***

Ne faites hériter une classe `B` d'une classe `A` que s'il est possible de considérer toute instance de `B` comme une instance de `A`.

En dépit de ce que vous pouvez penser de prime abord, c'est une règle souple, et non stricte. Voici pourquoi :

- remarquez l'expression "convaincre, d'une manière ou d'une autre". C'est volontairement vague : nous n'exigeons pas une *preuve* que tout `B` est un `A`. De nombreux cas laisseront place à l'interprétation. Est-il vrai que "tout livret d'épargne est un compte dépôt" ? Il n'y a pas de réponse absolue ; suivant le règlement de la banque et votre analyse des propriétés des divers types de comptes en banque, vous pouvez décider de faire hériter la classe `SAVINGS_ACCOUNT` de `BANK_ACCOUNT`, ou de la mettre ailleurs dans la structure d'héritage, en vous appuyant sur d'autres critères évoqués dans ce chapitre. Des personnes raisonnables peuvent ne pas être d'accord avec le résultat. Mais, pour que cela soit le cas, la justification "est-un" doit être plausible. Notre contre-exemple nous aide à nouveau : l'argument selon lequel un `CAR_OWNER` "est-un" `CAR` n'est *pas* plausible ;
- notre interprétation de ce que veut dire "est-un" sera particulièrement libérale. Par exemple, cela n'interdira pas l'*héritage d'implémentation* — une forme d'héritage que beaucoup considèrent comme discutable — tant que l'argument "est-un" peut, raisonnablement, être tenu.

Ces observations définissent à la fois l'utilité et les limitations de la règle est-un. Elle est utile comme règle *négative*, au sens de Popper, nous permettant de détecter et de rejeter des utilisations impropres de l'héritage. Mais, comme règle positive, elle n'est pas suffisante ; les suggestions d'utilisation qui passent le test ne seront pas toutes appropriées.

Aussi gratifiant que puisse être le contre-exemple de `CAR_OWNER`, tout sentiment de joie que nous aurons pu en tirer sera de courte durée. Il s'agissait là du début et, simultanément, de la fin des vraies bonnes nouvelles — le fait que certaines propositions d'utilisation d'héritage sont trivialement erronées et faciles à détecter. Le reste de ce chapitre devra affronter des nouvelles désagréables ou, tout au moins, mitigées : dans presque tous les autres cas, la décision est une vraie question de conception, c'est-à-dire difficile, bien que nous puissions trouver, heureusement, certaines règles générales de conduite.

## 24.2 PRÉFÉRERIEZ-VOUS ACHETER OU HÉRITER ?

La règle de base pour choisir entre les deux relations possibles entre modules, client et héritage, est remarquablement simple : être client, c'est *avoir* ; hériter, c'est *être*. Pourquoi alors le choix n'est-il pas toujours si facile ?

### Avoir et être

La raison en est que, si avoir n'est pas toujours être, *être est également*, dans de nombreuses situations, *avoir.*

Mais non, ce n'est ni un piètre aphorisme de philosophie existentialiste ni un slogan pour vous faire acheter une maison si vous en louez une actuellement ; ce sont de simples observations sur la difficulté de modélisation des systèmes. Nous avons déjà rencontré une illustration de la première propriété — avoir n'est pas toujours être — dans l'exercice précédent : un propriétaire de voiture a une voiture, mais il nous est impossible d'affirmer, même avec le raisonnement le plus tarabiscoté qui soit, qu'il est une voiture.

Et la réciproque ? Prenons une simple affirmation concernant des types d'objet de la vie courante, comme :

*Tout ingénieur logiciel est un ingénieur.*
[A]

dont nous admettons la véracité puisqu'il s'agit d'un exemple de la relation "est-un" (et ce quelle que soit votre opinion personnelle sur la pertinence de cette affirmation). Il semble, en effet, difficile de trouver un cas qui puisse plus clairement exprimer l'"être" plutôt que l'"avoir". Mais considérons la nouvelle formulation suivante de la propriété :

*Dans tout ingénieur logiciel se trouve un ingénieur.*
[B]

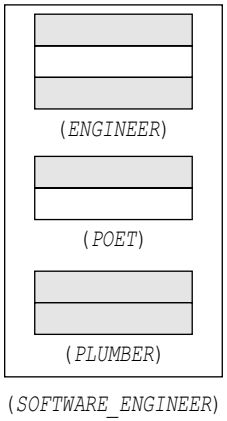
qui peut, alors, être réécrite sous la forme :

*Tout ingénieur logiciel a une composante "ingénieur".*
[C]

C'est, certes, tordu, et la manière dont est tournée l'expression est, même, un peu bizarre ; mais pas fondamentalement différente de notre version de base [A] ! Et voilà : en changeant légèrement notre perspective, nous pouvons reformuler la propriété "est" en une "a".

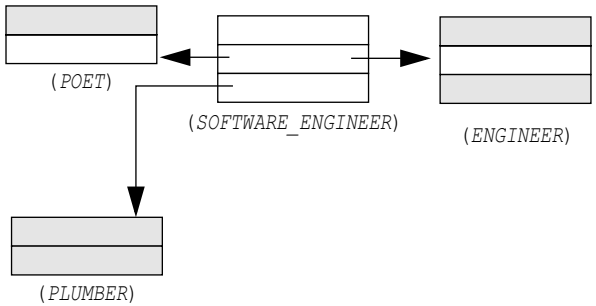
*"OBJETS COMPOSITES ET TYPES EXPANSÉS", 8.7, page 252.*

En adoptant le point de vue d'un programmeur, nous pouvons élaborer un diagramme d'objet dans le style de ceux qui nous ont servi lors de l'étude du modèle dynamique, dans un précédent chapitre, pour illustrer une instance typique d'une classe et de ses composants :




***Un objet "ingénieur logiciel" vu comme un agrégat***

Celui-ci montre une instance de `SOFTWARE_ENGINEER` formée de plusieurs sous-objets divers représentant les divers aspects de la personnalité et des tâches d'un ingénieur logiciel. À la place de sous-objets (la vision expansée), nous pourrions préférer raisonner en termes de références :




***Une autre vue possible***

Considérez ces deux représentations comme des moyens de visualiser la situation selon une perspective orientée implémentation, et rien de plus. Toutes deux suggèrent, cependant, qu'une interprétation client, ou "a" — tout ingénieur logiciel a un ingénieur comme sous-partie — est fidèle à l'expression d'origine. La même observation peut être faite pour toute relation "est-un" ("*is-a*") du même genre.

Cela explique pourquoi le problème de choix entre client et héritage n'est pas trivial : chaque fois que la vision "est" est légitime, on peut très bien adopter la vision "a" à la place.

La réciproque n'est pas vraie : quand "a" est légitime, "est" n'est pas toujours applicable, comme le montre clairement l'exemple `CAR_OWNER`. Cette observation règle leur sort aux erreurs faciles, évidentes pour toute personne ayant compris les principes de base, et peut-être même explicables aux auteurs de livres d'étude. Mais, chaque fois qu'on peut appliquer "est", il existe une alternative. Ainsi, deux personnes raisonnables et compétentes peuvent rester en désaccord, l'une souhaitant utiliser l'héritage, l'autre préférant la relation client.

Heureusement, il existe deux critères pouvant nous aider lors de telles discussions. Ils ne donnent pas toujours une solution claire et unique, ce qui n'est pas surprenant (puisqu'ils concernent une

vaste question de conception). Mais, dans de nombreux cas pratiques, ils indiquent, sans hésitation, laquelle des deux relations est la bonne.

Qui plus est, l'un de ces deux critères favorise l'héritage, tandis que l'autre favorise la relation client.

## La règle du changement

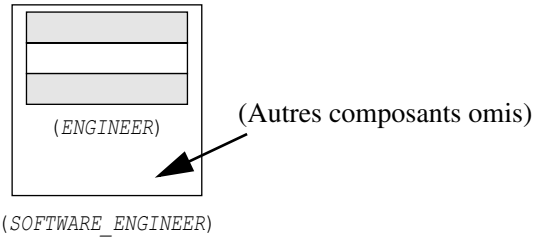
La première observation est que la relation client permet habituellement les changements, alors que la relation d'héritage ne les autorise pas. Ici, il faut être prudent quand on utilise les verbes "être" et "avoir" du langage de tous les jours ; ils nous ont, jusqu'à présent, aidé à caractériser la nature générale de nos deux relations logicielles, mais les règles logicielles sont, comme toujours, plus précises que leurs alter ego non logiciels.

Une des propriétés définissant l'héritage est qu'il s'agit d'une relation entre **classes**, et non entre objets. Nous avons interprété la propriété "la classe `B` hérite de `A`" par "tout objet `B` est un objet `A`", mais nous devons nous rappeler qu'aucun objet n'est en mesure de changer cette propriété : seul un changement de classe peut parvenir à un tel résultat. La propriété caractérise le logiciel, et non une exécution particulière.

Avec la relation client, les contraintes sont plus lâches. Si un objet de type `B` a un composant de type `A` (sous forme d'un sous-objet ou d'une référence à un objet), il est tout à fait possible de changer ce composant ; les seules restrictions sont celles du système de types, qui assure une exécution dont on peut démontrer qu'elle est fiable (et régie, ce qui est amusant, par la structure d'héritage).

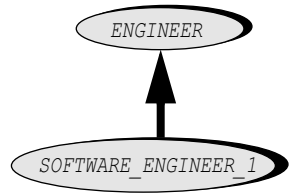
Ainsi, même si une relation donnée entre objets peut provenir d'une relation d'héritage ou d'une relation client entre les classes correspondantes, l'effet sur ce qui peut être changé ou non sera différent. Par exemple, notre structure fictive d'objet :

***Objet et sous-objet***




pourrait découler d'une relation d'héritage entre les classes correspondantes :

```
class SOFTWARE_ENGINEER_1 inherit
      ENGINEER
feature
            ...
end -- class SOFTWARE_ENGINEER_1
```

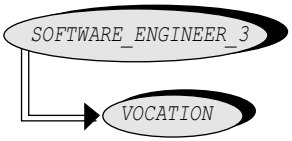



mais pourrait tout aussi bien avoir été obtenue via la relation client :

```
class SOFTWARE_ENGINEER_2 feature
      l_ingénieur_en_moi: ENGINEER
      ...
end -- class SOFTWARE_ENGINEER_2
```

qui pourrait, en fait, être :

```
class SOFTWARE_ENGINEER_3 feature
      ma_partie_vraiment_importante: VOCATION
      ...
end -- class SOFTWARE_ENGINEER_3
```




sous réserve que nous satisfassions aux règles de typage en faisant de la classe *ENGINEER* un descendant de la classe *VOCATION*.

> En toute rigueur, les deux dernières variantes représentent une situation légèrement différente de la première, si nous supposons qu'aucune des classes données ne soit expansée : à la place de sous-objets, les objets "ingénieur logiciel" contiendront, dans les deux derniers cas, des *références* vers des objets "ingénieur", comme dans la seconde figure de la page 785. L'introduction des références ne modifie, cependant, pas fondamentalement cette discussion.

Avec la première définition de classe, il n'est pas possible de modifier dynamiquement la relation entre objets, puisqu'une relation d'héritage existe entre les classes génératrices : ingénieur tu es, ingénieur tu resteras.

Mais une telle modification reste possible avec les deux autres définitions : une procédure de la classe "ingénieur logiciel" peut affecter une nouvelle valeur au champ correspondant de l'objet (le champ *l_ingénieur_en_moi* ou *ma_partie_vraiment_importante*). Dans le cas de la classe *SOFTWARE_ENGINEER_2*, la nouvelle valeur doit être compatible avec le type *ENGINEER* ; mais avec la classe *SOFTWARE_ENGINEER_3*, elle peut être d'un type quelconque, tant que celui-ci reste compatible avec *VOCATION*. Notre logiciel peut ainsi modéliser l'idée d'un ingénieur logiciel qui, après avoir passé des années à se dire ingénieur, se détourne finalement de cet aspect de sa personnalité au profit de quelque chose qu'il considère comme plus représentatif de son oeuvre, comme poète ou plombier.

Ceci conduit à notre premier critère :

***Règle du changement***

N'utilisez pas l'héritage pour décrire une relation qui semble de type "est-un" si les composants de l'objet correspondant peuvent être changés à l'exécution.

N'utiliser l'héritage que si la relation correspondante entre objets est permanente. Dans les autres cas, utilisez la relation client.

> Le cas vraiment intéressant est celui illustré par *SOFTWARE_ENGINEER_3*. Avec *SOFTWARE_ENGINEER_2*, vous pouvez simplement remplacer la composante ingénieur par une autre ayant exactement le même type. Mais, dans le schéma *SOFTWARE_ENGINEER_3*, *VOCATION* pourrait être une classe de haut niveau, très probablement retardée ; l'attribut peut ainsi (via le polymorphisme) représenter des objets ayant plusieurs types possibles, tous conformes à *VOCATION*.
> Cela veut également dire que, bien que cette solution utilise la relation client comme relation primaire, sa forme finale utilise souvent, en pratique, l'héritage en complément. Cela sera particulièrement évident quand nous en viendrons à la notion de *handle*.

## La règle du polymorphisme

Passons maintenant à un critère qui conduira à l'héritage et exclura la relation client. Ce critère est simple : les utilisations polymorphes. Lors de notre étude de l'héritage, nous avons vu que, avec une déclaration de la forme :

```
x: C
```

*x* désigne, à l'exécution (en supposant que la classe *C* ne soit pas expansée), une référence potentiellement polymorphe ; c'est-à-dire que *x* peut devenir attaché non seulement à de simples instances directes de *C*, mais à tout descendant propre de *C*. Cette propriété contribue, bien sûr, de manière essentielle, à la puissance et à la flexibilité de la méthode orientée objet, en particulier via son corollaire, la possibilité de définir des structures polymorphes de données comme *LIST* [ *C*] , qui peut contenir des instances de n'importe quel descendant de *C*.

Dans notre exemple, avec la solution *SOFTWARE_ENGINEER_1* — la forme de la classe héritant de *ENGINEER* —, cela veut dire qu'un client peut déclarer une entité :

```
eng: ENGINEER
```

qui peut devenir attachée à un objet de type *SOFTWARE_ENGINEER_1* lors de l'exécution. Nous pouvons également avoir une liste ou une base de données d'ingénieurs, qui contient des ingénieurs mécaniciens, des ingénieurs chimistes et aussi quelques ingénieurs logiciels.

> Un rappel méthodologique : utiliser des mots qui ne sont pas tirés de la littérature du logiciel peut être une aide précieuse pour comprendre des concepts, mais nous ne devrions pas nous laisser emporter par de tels exemples anthropomorphiques ; les objets concernés sont des objets logiciels. En conséquence, bien qu'il soit possible de considérer les mots "un ingénieur logiciel" pour ce qu'ils représentent, ils désignent, de fait, une instance de *SOFTWARE_ENGINEER_1*, c'est-à-dire un objet logiciel modélisant, d'une façon ou d'une autre, une personne réelle.

Ces effets polymorphes reposent sur l'héritage : dans *SOFTWARE_ENGINEER_2* ou *SOFTWARE_ENGINEER_3*, une entité ou une structure de données de type *ENGINEER* ne peut pas désigner directement des objets "ingénieur logiciel".

La généralisation de ces observations — qui ne sont, bien sûr, pas spécifiques de cet exemple — conduit au complément de la règle du changement :

***Règle du polymorphisme***

L'héritage est approprié pour décrire une relation qui semble de type "est-un" s'il peut s'avérer nécessaire d'attacher des entités ou des composants de structures de données d'un type plus général à des objets d'un type plus spécialisé.

# Résumé

Bien qu'elle n'introduise pas de concept nouveau, la règle suivante sera pratique pour résumer cette présentation des critères pour ou contre l'héritage.

***Choisir entre relation client et héritage***

Pour déterminer la manière d'exprimer la dépendance d'une classe *B* envers une classe *A*, appliquez les critères suivants :

CI1 • si chaque instance de *B* possède initialement un composant de type *A*, et si ce composant a besoin d'être remplacé à l'exécution par un objet de type différent, faites de *B* un client de *A* ;

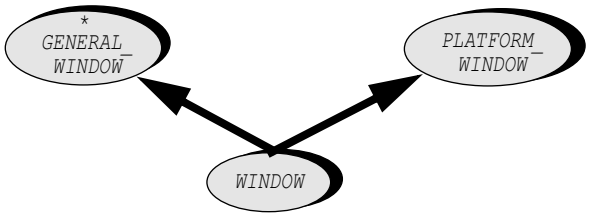
CI2 • s'il est nécessaire que des entités de type *A* puissent désigner des objets de type *B*, ou si on a besoin de structures polymorphes contenant des objets de type *A* dont certains peuvent être de type *B*, faites de *B* un héritier de *A*.

## 24.3 UNE APPLICATION : LA TECHNIQUE DU HANDLE

Voici un exemple qui utilise la règle précédente. Il conduit à un schéma de conception ayant de nombreuses applications : les *handles* (ou poignées).

On a rencontré, lors de la première conception de la bibliothèque graphique *Vision*, indépendante de la plate-forme sous-jacente, un problème d'ordre général : comment prendre en compte les dépendances induites par les différentes plates-formes. La première solution était fondée sur l'héritage multiple, de la façon suivante : une classe typique, comme celle décrivant les fenêtres, avait un parent pour décrire les propriétés de l'abstraction qui étaient indépendantes de la plate-forme, et une autre pour fournir les éléments spécifiques de celle-ci.

```
class WINDOW inherit
      GENERAL_WINDOW
      PLATFORM_WINDOW
feature
      ...
end -- class WINDOW
```




***Adaptation de plate-forme par héritage***

La classe `GENERAL_WINDOW` et celles du même genre, comme `GENERAL_BUTTON`, sont retardées : elles expriment tout ce qui peut être dit à propos des objets graphiques correspondants, y compris les opérations qui sont applicables sans faire référence à une plate-forme graphique particulière. Les classes comme `PLATFORM_WINDOW` fournissent le lien vers la plate-forme graphique, comme Windows, OS/2-Presentation-Manager ou Unix-Motif ; elles permettent d'accéder aux mécanismes spécifiques de la plate-forme (encapsulés dans une bibliothèque comme WEL ou MEL).

*En ce qui concerne les bibliothèques WEL et MEL spécifiques de une plate-forme, voir "Restructuration orientée objet", page 427.*

Une classe comme `WINDOW` combinera alors ses deux parents via des caractéristiques qui rendent effectives (implémentent) les caractéristiques retardées de `GENERAL_WINDOW` en utilisant les mécanismes d'implémentation fournis par `PLATFORM_WINDOW`.

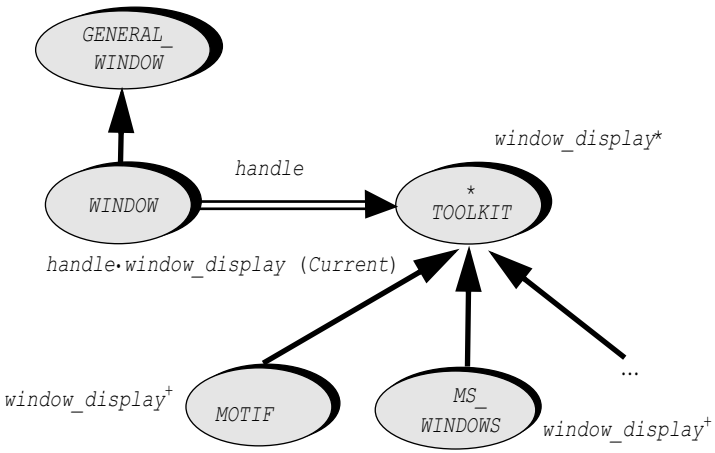
`PLATFORM_WINDOW` (et les autres classes du même genre) a besoin de plusieurs variantes, chacune propre à une plate-forme donnée. Ces classes de noms identiques seront stockées dans des répertoires différents ; lors d'une compilation, le fichier Ace (le fichier de commande) sélectionnera celui qui est approprié.

*Sur la notion de Ace, voir "Assembler un système", page 201.*

Cette solution fonctionne, mais elle présente l'inconvénient de fortement lier la notion de `WINDOW` à la plate-forme choisie. En transposant un commentaire précédent concernant l'héritage : fenêtre Motif tu es, fenêtre Motif tu seras. Ce n'est pas trop grave, car il est difficile d'envisager qu'une fenêtre Unix, prise d'une soudaine crise existentielle, décide de devenir une fenêtre OS/2. La situation devient moins absurde si nous généralisons notre définition de "plate-forme" pour prendre en compte des formats comme Postscript ou HTML ; un objet graphique pourrait alors changer de représentation en vue d'une impression ou d'une intégration dans un document Web.

L'observation selon laquelle nous pourrions avoir besoin d'un lien moins fort entre les objets GUI comme des fenêtres et la boîte à outils graphiques sous-jacente conduit à envisager d'essayer la relation client. Un lien d'héritage subsistera entre *WINDOW* et *GENERAL_WINDOW* ; mais la dépendance envers la plate-forme sera représentée par un lien client vers une classe *TOOLKIT* représentant la "boîte à outils" sous-jacente (la plate-forme graphique). La figure suivante illustre la structure qui en résulte, mettant en jeu relation client et héritage.

***Adaptation à la plate-forme via un handle***




Cette solution à l'avantage de mettre en exergue la notion de boîte à outils vue comme une abstraction à part entière et représentée par une classe retardée *TOOLKIT*. Chaque boîte à outils spécifique est alors représentée par un descendant effectif de *TOOLKIT* comme *MOTIF* ou *MS_WINDOWS*.

Voici le fonctionnement. Chaque classe décrivant des objets graphiques, comme *WINDOW*, possède un attribut fournissant un accès à la plate-forme sous-jacente :

```
handle: TOOLKIT
```

Cela introduit un champ dans chaque instance de la classe. Il est possible de changer le handle :

```
set_handle (new: TOOLKIT) is
                    -- Faire de new le nouveau handle de cet objet.
        do
                handle := new
        end
```

Une opération typique héritée de *GENERAL_WINDOW* sous forme retardée sera rendue effective par un appel au mécanisme de la plate-forme :

```
display is
                    -- Afficher la fenêtre sur l'écran.
        do
                handle.window_display (Current)
        end
```

Via le handle, l'objet graphique demande à la plate-forme d'effectuer l'opération requise. Une caractéristique comme *window_display* est retardée dans la classe *TOOLKIT* et est rendue effective différemment suivant les divers descendants comme *MOTIF*.

Remarquez qu'il serait impropre de tirer de cet exemple la conclusion "Voilà encore un cas où l'héritage a été utilisé à mauvais escient, et où la version finale s'abstient de l'utiliser." La version initiale n'est pas en tort ; en fait, elle fonctionne très bien, mais elle est moins flexible que la seconde. Et cette seconde version repose fondamentalement sur l'héritage et les techniques associées du polymorphisme et de la liaison dynamique, qui se combinent avec la relation client. Sans la hiérarchie d'héritage fondée sur `TOOLKIT`, l'entité polymorphe `handle` et la liaison dynamique des caractéristiques comme `window_display`, cela ne marcherait pas. Loin de rejeter l'héritage, cette technique illustre plutôt une forme plus sophistiquée de celui-ci.

La technique du handle est souvent utilisée lors du développement de bibliothèques offrant une compatibilité entre plusieurs plates-formes. Outre la bibliothèque graphique *Vision*, nous l'avons appliquée à la bibliothèque de base de données *Store*, où la notion de plate-forme correspond aux diverses interfaces SQL de Oracle, Ingres, Sybase et ODBC.

## 24.4 TAXOMANIE

À l'aide de chacune des catégories d'héritage qui seront introduites par la suite dans ce chapitre, l'héritier pourra redéclarer (redéfinir ou rendre effectives) certaines caractéristiques héritées, introduire des caractéristiques propres ou enrichir l'invariant. (Il pourra, bien évidemment, faire plusieurs choses en même temps.) En conséquence :

***Règle de taxomanie***

Tout héritier doit introduire une caractéristique, redéclarer une caractéristique héritée ou ajouter une clause d'invariant.

*C'est, en fait, une conséquence de la règle d'héritage évoquée plus loin dans ce chapitre, page 793.*

Cette règle s'adresse aux nouveaux adeptes de la méthode OO, dont l'enthousiasme leur fait voir partout des divisions taxonomiques (d'où le nom de la règle, contraction de "manie de la taxonomie"). Il en résulte des hiérarchies d'héritage beaucoup trop compliquées. La taxonomie et l'héritage ont pour vocation de nous *aider* à maîtriser la complexité, et non de l'introduire. Ajouter des niveaux de classification inutiles est digne d'un masochiste.

Comme c'est souvent le cas, pour être dans le vrai — et ramener les néophytes à la raison —, il suffit de garder constamment à l'esprit la vision ADT. Une classe est une implémentation, partielle ou totale, d'un type abstrait de données. Des classes différentes, en particulier un parent et son héritier, devraient décrire des ADT différents. Ainsi, puisqu'un ADT est entièrement caractérisé par ses caractéristiques et leurs propriétés (représentées, dans la classe, par les assertions), une nouvelle classe devrait changer une caractéristique héritée, en introduire une nouvelle ou changer une assertion. Puisque vous ne pouvez changer une précondition ou une postcondition qu'en redéfinissant la caractéristique englobante, le dernier cas revient à ajouter une clause d'invariant (comme dans l'*héritage de restriction*, une des catégories de notre taxonomie).

Vous pouvez parfois justifier un cas de taxomanie — une classe qui, hormis sa propre existence, n'apporte rien de neuf — en arguant du fait que la classe héritière décrit une variante importante de la notion décrite par le parent, et que vous l'introduisez maintenant pour préparer une future introduction ou redéclaration d'une caractéristique, même s'il n'en a encore aucune. Cela peut être justifié si la structure d'héritage correspond à une classification généralement acceptée du domaine du problème. Mais méfiez-vous cependant de telles situations et résistez à l'envie d'introduire des classes sans nouvelles caractéristiques, sauf si vous avez des arguments irréfutables.

Voici un exemple. Supposez qu'un certain système ou qu'une bibliothèque contienne une classe *PERSON* et que vous envisagiez d'ajouter les héritiers *MALE* et *FEMALE*. Est-ce justifié ? Il vous faudra y regarder à deux fois. Un système de gestion du personnel qui contient des caractéristiques propres à chaque genre, par exemple pour traiter les congés de maternité, peut tirer parti des classes *MALE* et *FEMALE*. Mais, dans bien d'autres cas, ces variantes, si elles existaient, n'auraient pas de caractéristiques spécifiques ; par exemple, un logiciel statistique qui ne ferait qu'enregistrer le genre des individus aurait intérêt à se limiter à une classe *PERSON* unique et un attribut booléen :

```
female: BOOLEAN
```

ou, peut-être :

```
Female: INTEGER is unique
Male: INTEGER is unique
```

au lieu d'introduire de nouveaux héritiers. Pourtant, s'il est plausible d'envisager, dans le futur, l'ajout de caractéristiques spécifiques, la classification correspondante est tellement ancrée dans le domaine du problème qu'il se peut que vous souhaitiez néanmoins introduire ces héritiers.

Une règle de conduite à garder à l'esprit est le principe de choix unique. Nous avons appris à nous méfier des listes explicites de variantes, implémentées par des constantes **unique**, pour éviter que notre logiciel soit pollué par des instructions conditionnelles de la forme :

```
if female then
        ...
else
        ...
```

ou par des instructions **inspect**. Il n'y a pourtant aucune raison de s'inquiéter ici :

- une des principales critiques adressées à ce style était que tout nouvel ajout d'une variante entraînerait une réaction en chaîne de changements dans le logiciel, mais, dans certains cas — comme ici — nous pouvons, en toute confiance, être sûrs qu'il n'y aura pas d'autres variantes ;
- si l'ensemble des variantes est fixe, le style du **if** ... explicite est moins efficace que celui auquel conduit la liaison dynamique par des appels comme *this_person.some_operation*, où *MALE* et *FEMALE* sont des redéclarations différentes de *some_operation*. Mais, s'il nous faut effectivement distinguer le genre d'une personne, nous violons la prémisse de cette discussion — à savoir qu'il n'y a pas de caractéristiques spécifiques des variantes. Si de telles caractéristiques existent, l'héritage est justifié.

Le dernier commentaire met en avant la vraie difficulté. Des cas simples de taxomanie — dans lesquels le parent ajoute sans raison des noeuds intermédiaires dans la structure d'héritage — sont relativement faciles à diagnostiquer (en identifiant des classes sans caractéristiques spécifiques) et à corriger. Mais, que faire si les variantes ont *effectivement* des caractéristiques spécifiques, bien que la classification résultante entre en conflit avec d'autres critères ? Un système de gestion du personnel dans lequel nous pouvons justifier une classe *FEMALE_EMPLOYEE* par quelques caractéristiques spécifiques pourrait, tout aussi bien, avoir besoin d'autres distinctions, comme employés permanents et temporaires, ou contrôleurs et non-contrôleurs. Il ne s'agit plus d'un cas de taxomanie mais d'un problème général et délicat, la *classification multicritère*, pour lequel nous proposerons plus tard, dans ce chapitre, des solutions possibles.

## 24.5 UTILISER L'HÉRITAGE : UNE TAXONOMIE DE LA TAXONOMIE

La puissance de l'héritage provient de sa versatilité. Cela le rend parfois inquiétant, au point de conduire de nombreux auteurs à lui imposer certaines restrictions. Tout en comprenant ces angoisses et les partageant parfois — les plus courageux n'ont-ils pas, eux aussi, leurs moments de doute ? —, nous devrions les surmonter et apprendre à apprécier l'héritage sous toutes ses variantes légitimes, variantes que nous allons explorer ici.

Après avoir rappelé les utilisations erronées que l'on rencontre le plus fréquemment, nous évoquerons chacune de ses utilisations valides :

- l'héritage de sous-type ;
- l'héritage de vue ;
- l'héritage de restriction ;
- l'héritage d'extension ;
- l'héritage de variation fonctionnelle ;
- l'héritage de variation de type ;
- l'héritage de concrétisation ;
- l'héritage de structure ;
- l'héritage d'implémentation ;
- l'héritage de service (et ses deux variantes spéciales : l'héritage de constante et l'héritage de machine).

Certaines de ces catégories (sous-type, vue, implémentation, service) soulèvent des questions spécifiques et seront évoquées plus en détail dans des sections séparées.

### Portée des règles

La vision relativement large de l'héritage adoptée dans ce livre n'implique en aucune manière que "tout est possible". Nous acceptons et, en fait, encourageons certaines formes d'héritage que dénigrent certains auteurs ; mais, il est bien sûr possible d'utiliser l'héritage de travers, et pas simplement comme `CAR_OWNER`. Le corollaire inévitable de notre ouverture d'esprit est une contrainte particulièrement stricte :

> ***Règle d'héritage***
>
> Toute utilisation de l'héritage devrait appartenir à l'une des catégories acceptées.

Cette règle est effectivement rigide : elle indique que les types d'héritage sont d'ores et déjà connus et que, si vous rencontrez un cas qui n'est pas couvert par l'un de ces types, vous ne devriez *pas* utiliser l'héritage.

Quelles sont les "catégories acceptées" ? Cette question contient une qualification implicite, "les catégories décrites dans le reste de cette section". J'espère, effectivement, que tous les cas pertinents sont traités. Mais la formulation est un peu plus précise, car il est possible qu'il faille réviser la taxonomie. J'ai trouvé très peu d'informations dans la littérature sur ce sujet ; la référence la plus utile est une thèse de doctorat non publiée [Girod 1991]. Il est donc tout à fait

possible que cette tentative de classification ait omis certaines catégories. Mais cette règle indique que, si vous envisagez une utilisation possible de l'héritage qui ne correspond pas aux catégories précitées, vous devriez y réfléchir sérieusement. L'utilisation de l'héritage sera probablement déconseillée dans ce cas ; si, après réflexion, vous êtes toujours convaincu que l'héritage est justifié, sans parvenir à raccrocher votre exemple à l'une des catégories de ce chapitre, il se peut que vous soyez en train de faire une nouvelle découverte.

*Page 791.*

> Nous avons déjà vu une conséquence de la règle d'héritage : la règle de taxonomie, qui indique que toute classe héritière devrait redéclarer une caractéristique, en introduire une nouvelle ou changer une assertion. Elle découle directement de l'observation selon laquelle toute forme légitime d'héritage détaillée ci-dessous impose à l'héritier d'effectuer au moins l'une de ces opérations.

La règle d'héritage n'interdit pas les liens d'héritage qui appartiennent à *plusieurs* catégories d'héritage. Une telle pratique n'est, cependant, pas recommandée :

***Règle de simplicité d'héritage***

Il est préférable qu'une utilisation de l'héritage n'appartienne qu'à une catégorie.

*Voir "Conseils", page 649.*

Cette règle n'est pas absolue, mais elle est en fait un "conseil positif". Cette règle est, ici aussi, motivée par le désir de simplicité et de clarté : si, chaque fois que vous introduisez un lien d'héritage entre deux classes, vous appliquez des principes méthodologiques explicites et, en particulier, que vous décidez laquelle des variantes autorisées sera utilisée, vous risquez moins de faire une erreur de conception ou de produire une structure de système bancale, difficile à utiliser et à maintenir.

Cependant, il ne semble pas exister d'argument fort permettant de rendre cette règle absolue et il s'avère pratique, de temps en temps, d'utiliser un lien unique d'héritage pour deux objectifs définis par la classification. Cela reste cependant un cas rare.

> Malheureusement, je ne connais pas de critère simple indiquant, sans ambiguïté, quand il est possible de contracter plusieurs catégories d'héritage en un seul lien. D'où la nature non catégorique de la règle de simplicité d'héritage. Le jugement du lecteur, fondé sur une compréhension claire de la méthodologie de l'héritage, sera requis dans tous les cas limites.

## Mauvaises utilisations

Les deux règles précédentes ne font que confirmer une évidence : il est possible d'utiliser l'héritage à mauvais escient. Voici une liste d'erreurs typiques ayant, pour la plupart, déjà été mentionnées. La capacité humaine à se tromper étant ce qu'elle est, nous ne pouvons en aucune manière espérer couvrir tous les cas, mais quelques erreurs fréquentes sont faciles à identifier.

La première est la présence d'une **relation "a" sans relation "est"**. `CAR_OWNER` nous a servi d'exemple — extrême, mais pas isolé. J'ai souvent entendu ou vu des cas similaires, souvent présentés comme des exemples d'héritage multiple, comme `APPLE_PIE`, héritant de `APPLE` et de `PIE`, ou (celui-ci m'a été indiqué par Adele Goldberg) `ROSE_TREE`, héritant de `ROSE` et de `TREE`.

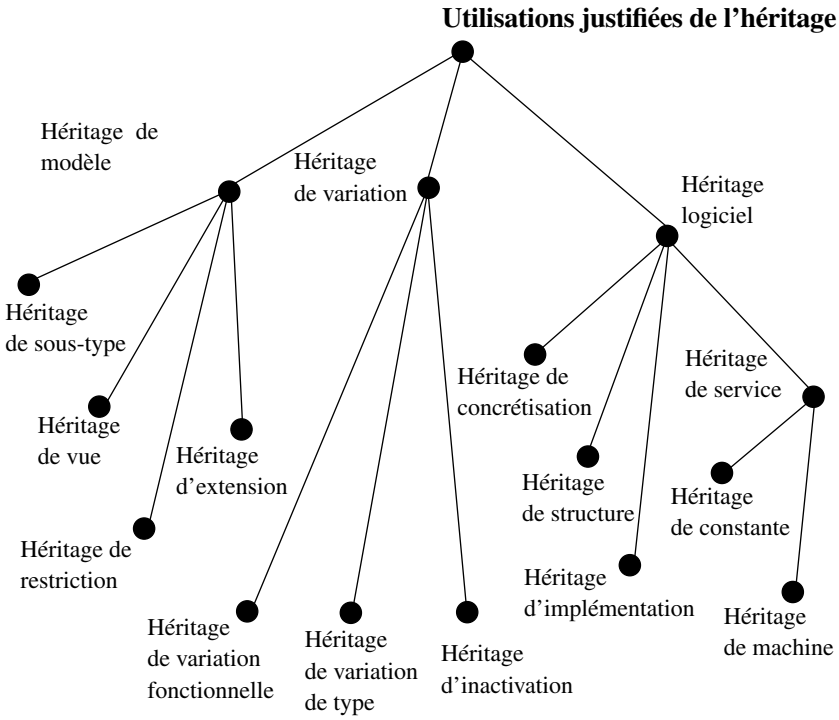
Une autre erreur correspond au cas typique de **taxomanie** dans lequel une simple propriété booléenne, comme le genre d'une personne (ou une propriété pouvant prendre quelques valeurs fixes, comme la couleur d'un feu tricolore), est utilisée comme critère d'héritage, bien qu'aucune variante de caractéristique n'en dépende de manière significative.

Une troisième erreur typique est l'**héritage de confort**, dans lequel le développeur se rend compte qu'une classe contient quelques caractéristiques utiles et hérite de celle-ci à seule fin de réutiliser ces caractéristiques. Ici, ce qui ne va pas, ce n'est ni le fait d'"*utiliser l'héritage pour l'implémentation*", ni celui d'"*hériter d'une classe pour ses caractéristiques*", qui sont toutes deux des formes acceptables d'héritage que nous étudierons plus loin dans ce chapitre, mais l'utilisation d'une classe comme parent *en l'absence d'une relation est-un entre abstractions correspondantes* — ou, dans certains cas, sans quelque abstraction adéquate que ce soit.

## Taxonomie générale

Passons maintenant aux utilisations justifiées de l'héritage. La liste contiendra deux catégories différentes, groupées, en pratique, en trois grandes familles :

***Classification des catégories d'héritage***




La classification est fondée sur l'observation que tout système logiciel reflète un certain modèle externe, lié lui-même à une réalité extérieure dans le domaine d'application du logiciel. Nous pouvons alors distinguer :

- l'héritage de modèle, qui reflète des relations "est-un" entre abstractions du modèle ;
- l'héritage logiciel, exprimant des relations au sein du logiciel, sans contrepartie évidente dans le modèle ;
- l'héritage de variation — cas spécial appartenant au logiciel ou au modèle —, qui sert à décrire une classe par ses différences avec une autre classe.

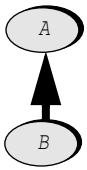
Ces trois catégories générales facilitent la compréhension, mais les propriétés les plus importantes sont groupées dans les catégories finales (les feuilles de l'arbre de la figure précédente).

*Exercice E24.2, page 838.*

> Puisque la classification est, elle-même, une taxonomie, vous pouvez vous demander, par curiosité, comment les catégories qui viennent d'être identifiées s'y appliquent. C'est le sujet d'un exercice.

Les définitions qui suivent utilisent les noms `A` pour la classe parent et `B` pour l'héritier.

***Convention de nom pour les définitions des catégories d'héritage***




Chaque définition indiquera si `A` ou `B` peut être retardée ou effective. Une table, en fin de présentation, résume les catégories applicables à chaque combinaison retardée-effective.

## Héritage de sous-type

Nous débuterons avec la forme la plus évidente d'héritage de modèle. Vous êtes en train de modéliser un système externe dans lequel une catégorie d'objets (externes) peut être décomposée en sous-catégories disjointes — comme pour les figures fermées, décomposées en polygones, ellipses, etc. — et vous utilisez l'héritage pour organiser les classes correspondantes dans le logiciel. De manière un peu plus formelle :

***Définition : héritage de sous-type***

L'héritage de sous-type s'applique si `A` et `B` représentent certains ensembles `A'` et `B'` d'objets externes tels que `B'` est un sous-ensemble de `A'` et l'ensemble modélisé par tout autre héritier de sous-type de `A` est disjoint de `B'`. `A` doit être retardée.

`A'` pourrait être l'ensemble des figures fermées, `B'` l'ensemble des polygones, `A` et `B` les classes correspondantes. Dans la plupart des cas pratiques, le "système externe" ne sera pas logiciel, par exemple un aspect quelconque d'une entreprise (où les objets externes pourraient être des comptes en banque, dépôt ou épargne) ou une partie du monde physique (où il pourrait s'agir de planètes et d'étoiles).

*"HÉRITAGE DE SOUS-TYPE ET RÉTENTION DE DESCENDANT", 24.7, page 806.*

L'héritage de sous-type est la forme d'héritage la plus proche des taxonomies hiérarchiques de la botanique, de la zoologie ou des autres sciences de la nature (`VERTEBRATE` ← `MAMMAL` et autres). Un exemple logiciel typique (différent des figures fermées et des polygones) est `DEVICE` ← `FILE`. Nous insistons pour que le parent, `A`, soit retardé, de sorte qu'il décrive un ensemble pas entièrement spécifié d'objets. `B`, l'héritier, peut être effectif ou retardé. Les deux catégories suivantes traitent le cas où `A` peut être effectif.

Une prochaine section explorera plus en détail cette catégorie d'héritage, qui n'est pas toujours aussi évidente qu'elle en a l'air à première vue.

## Héritage de restriction

***Définition : héritage de restriction***

L'héritage de restriction s'applique si les instances de `B` sont, parmi les instances de `A`, celles qui vérifient une certaine contrainte, exprimée, si possible, dans l'invariant de `B` et absente de l'invariant de `A`. Toute caractéristique introduite par `B` devrait être une conséquence logique de la contrainte ajoutée. `A` et `B` devraient être toutes deux retardées ou toutes deux effectives.

Des exemples typiques sont `RECTANGLE` ← `SQUARE`, où la contrainte supplémentaire est `side1 = side2` (introduite dans l'invariant de `SQUARE`), et `ELLIPSE` ← `CIRCLE`, où la contrainte supplémentaire indique que les deux foyers d'une ellipse  sont, dans un cercle ⊙, confondus en un même point ; dans le cas général, une ellipse est l'ensemble des points tels que la somme de leurs distances aux deux foyers  est constante. De nombreux exemples mathématiques tombent effectivement dans cette catégorie.

La dernière partie de la définition a pour objectif d'éviter de confondre cette forme d'héritage avec celles dans lesquelles l'héritier peut ajouter de toutes nouvelles caractéristiques, ce qui est, en particulier, le cas avec l'héritage d'extension. Pour éviter les complications, il est ici préférable de limiter les nouvelles caractéristiques, s'il y en a, à celles qui découlent directement de la contrainte ajoutée. Par exemple, la classe `CIRCLE` introduira une nouvelle caractéristique `radius` qui vérifie cette propriété : dans un cercle, tous les points sont à la même distance du centre et cette distance mérite le statut de caractéristique de la classe, tandis que la notion correspondante dans la classe `ELLIPSE` (la moyenne des distances aux deux foyers) n'était probablement pas considérée comme suffisamment importante pour mériter d'être une caractéristique.

Puisque le seul changement conceptuel de `A` à `B` est l'ajout de contraintes, les classes devraient être toutes retardées ou toutes effectives.

L'héritage de restriction est, conceptuellement, proche de l'héritage de sous-type ; l'étude à venir sur le sous-typage s'appliquera, pour l'essentiel, à ces deux catégories.

## Héritage d'extension

***Définition : héritage d'extension***

L'héritage d'extension s'applique quand `B` introduit des caractéristiques qui ne sont pas présentes dans `A` et qui ne sont pas applicables aux instances directes de `A`. La classe `A` doit être effective.

La présence simultanée des variantes de restriction et d'extension est un des paradoxes de l'héritage. Comme on l'a noté lors de l'étude de l'héritage, l'extension s'applique aux caractéristiques alors que la restriction (et, plus généralement, la spécialisation) s'applique aux instances, mais cela n'élimine pas totalement le paradoxe.

Le problème vient des attributs que l'on trouve, habituellement, parmi les caractéristiques ajoutées. Ainsi, si nous adoptons l'interprétation naïve d'un type (donné par la classe) comme l'ensemble de ses instances, nous avons l'impression que la relation d'inclusion va dans le mauvais sens ! Soit, par exemple :

```
class A feature a1: INTEGER end

class B inherit
      A
feature
       b1: REAL
end
```

*Les lecteurs non mathématiciens peuvent sauter ce paragraphe.*

Si nous considérons chaque instance de `A` comme représentant un singleton, c'est-à-dire un ensemble ne contenant qu'un entier (que nous pouvons écrire `<n>`, où `n` est l'entier choisi) et chaque instance de `B` comme une paire contenant un entier et un réel (comme la paire `<1, -2.5>`), l'ensemble des paires `MB` n'est pas un sous-ensemble de l'ensemble des singletons `MA`. En fait, si nous tenons absolument à une relation d'inclusion, elle sera dans le sens opposé : il existe une relation bi-univoque entre `MA` et l'ensemble de toutes les paires ayant un second élément donné, par exemple `0.0`.

La découverte que la relation d'inclusion semble aller dans le mauvais sens pourrait nous conduire à considérer avec circonspection l'héritage d'extension. Par exemple, une version précédente d'une bibliothèque OO respectée (indépendante d'ISE) faisait hériter `RECTANGLE` de `SQUARE`, et non pas l'inverse comme nous l'avons appris. Le raisonnement était simple : `SQUARE` a un attribut `side` ; `RECTANGLE` hérite de `SQUARE` et ajoute une nouvelle caractéristique, `other_side`, et voilà notre lien d'héritage ! Plusieurs personnes ont critiqué cette conception et elle a rapidement été inversée.

Mais nous ne pouvons pas laisser tomber la catégorie générale de l'héritage d'extension. Son équivalent en mathématiques, où vous spécialisez une certaine notion en lui ajoutant de nouvelles opérations, est fréquemment utilisé et est considéré comme vraiment nécessaire. Un exemple typique est celui d'un *anneau*, qui spécialise la notion de *groupe*. Un groupe est muni d'une certaine opération, disons `+`, possédant certaines propriétés. Un anneau est un groupe, qui possède donc `+` et ses propriétés, mais auquel on ajoute une nouvelle opération, par exemple `*`, ayant elle-même des propriétés. Ce n'est pas fondamentalement différent de l'introduction d'un nouvel attribut dans une classe logicielle héritière.

Le schéma correspondant est également fréquent dans le logiciel OO. Dans la plupart des applications, `SQUARE` devrait, bien sûr, hériter de `RECTANGLE`, et non l'inverse ; mais il n'est pas difficile d'envisager des exemples légitimes. Une classe `MOVING_POINT` (pour des applications cinématiques) pourrait hériter d'une classe purement graphique `POINT` et ajouter une caractéristique `speed` décrivant l'intensité et la direction de la vitesse ; ou, dans une application de traitement de texte, une classe `CHAPTER` pourrait hériter de `DOCUMENT` en ajoutant les caractéristiques spécifiques d'un document qui est un chapitre d'un livre, comme sa position courante dans le livre et une procédure pour le placer ailleurs.

## Un modèle mathématique adéquat

(Les lecteurs peu férus de mathématiques devraient sauter cette section.)

Pour avoir l'esprit en paix, résolvons le paradoxe apparent remarqué plus haut (la découverte que `MB` n'est pas un sous-ensemble de `MA`), puisque nous voulons absolument qu'une relation d'inclusion existe, d'une manière ou d'une autre, entre les instances d'un héritier et celles du parent. Cette relation existe effectivement dans le cas de l'héritage d'extension ; ce paradoxe

montre simplement qu'il n'est pas correct d'utiliser le produit cartésien des types des attributs pour modéliser une classe. Étant donné une classe :

```
class C feature
        c1: T1
        c2: T2
        c3: T3
end
```

nous ne devrions *pas* prendre pour modèle mathématique `C'` de l'ensemble des instances de `C` le produit cartésien `T'1 × T'2 × T'3`, où les apostrophes indiquent que nous utilisons récursivement les ensembles du modèle ; cela conduirait à un paradoxe (entre autres inconvénients).

Nous devrions plutôt considérer toute instance comme une fonction partielle de l'ensemble des noms possibles d'attributs `ATTRIBUTE` vers l'ensemble de toutes les valeurs possibles `VALUE`, avec les propriétés suivantes :

*Les fonctions en question sont non seulement partielles mais finies.*

A1 • la fonction est définie pour `c1`, `c2` et `c3` ;

A2 • l'ensemble `VALUE` (l'ensemble cible de la fonction) est un sur-ensemble de `T'1 ∪ T'2 ∪ T'3` ;

A3 • la valeur de la fonction pour `c1` est dans `T'1`, et ainsi de suite.

Puis, si nous nous rappelons qu'une fonction est un cas particulier de relation et qu'une relation est un ensemble de paires (par exemple, une instance de la classe `A` peut être modélisée par la fonction `{<a1, 25>}` et l'instance de `B` citée précédemment par `{<a1, 1>, <b1, -2.5>}`), nous obtenons la propriété attendue que `B'` est un sous-ensemble de `A`.

> Remarquez qu'il est essentiel de formuler la propriété A1 sous la forme "La fonction est définie pour…" et non "Le domaine de la fonction est…", qui limiterait le domaine à l'ensemble `{c1, c2 c3}`, empêchant les descendants d'ajouter leurs propres attributs. Avec cette approche, tout objet logiciel est modélisé par une infinité d'objets mathématiques (finis).

Cette étude n'a fait que donner un aperçu du modèle mathématique. Pour plus de détails concernant les fondements mathématiques généraux et l'utilisation des fonctions partielles pour modéliser les tuples, voir [M 1990].

## Héritage de variation

Saluons le retour de nos lecteurs non-mathématiciens et considérons la deuxième de nos trois grandes familles de catégories d'héritage : l'héritage de variation.

***Définition : héritages de variation fonctionnelle et de type***

L'héritage de variation s'applique si `B` redéfinit certaines caractéristiques de `A` ; `A` et `B` sont toutes deux retardées ou toutes deux effectives, et `B` n'introduit pas de nouvelles caractéristiques, si ce n'est pour les seuls besoins des caractéristiques redéfinies. Deux cas se présentent :

- l'héritage de variation fonctionnelle : certaines redéfinitions jouent sur le corps des caractéristiques et non sur leur seule signature ;
- l'héritage de variation de type : toutes les redéfinitions sont des redéfinitions de signature.

L'héritage de variation est applicable quand une classe `A` décrivant une certaine abstraction est déjà utile en elle-même, mais que vous éprouvez le besoin de représenter une abstraction

similaire qui, bien que n'étant pas identique, possède à peu près les mêmes caractéristiques, avec quelques différences de signature ou d'implémentation.

La définition impose que les deux classes soient toutes deux effectives (le cas le plus fréquent) ou toutes deux retardées : l'héritage de variation ne concerne pas le cas d'une classe rendue effective, situation dans laquelle nous faisons passer une notion d'un état abstrait à un état concret. L'inactivation, étudiée par la suite, dans laquelle certaines caractéristiques effectives deviennent retardées, est fortement liée à ce type d'héritage.

La définition stipule que l'héritier ne devrait introduire aucune nouvelle caractéristique, si ce n'est pour les seuls besoins des caractéristiques redéfinies. Cette clause distingue l'héritage de variation de celui d'extension.

Dans l'héritage de variation de **type**, vous ne changez que la signature (types et nombre des arguments et du résultat) de certaines caractéristiques. Cette forme d'héritage est suspecte ; elle est souvent signe de taxomanie. Dans les cas légitimes, l'objectif peut être cependant de préparer un futur héritage d'extension ou de variation d'implémentation. Un exemple d'héritage de variation de type pourrait être celui des héritiers *MALE_EMPLOYEE* et *FEMALE_EMPLOYEE*.

L'héritage de variation de type n'est pas nécessaire quand la signature d'origine utilise des déclarations ancrées (**like** ...). Par exemple, dans la classe *SEGMENT* d'un paquetage de dessin interactif, il se peut que vous ayez introduit une fonction :

```
perpendicular: SEGMENT is
            -- Segment de même longueur et de même milieu, tourné de 90 degrés
        ...
```

et que vous souhaitiez maintenant définir un héritier *DOTTED_SEGMENT* pour fournir une représentation graphique d'une ligne pointillée, en plus des lignes continues. Dans cette classe, *perpendicular* devrait renvoyer un résultat de type *DOTTED_SEGMENT* et vous devrez donc redéfinir le type. Tout cela ne serait pas nécessaire si la fonction d'origine avait renvoyé un résultat de type **like** *Current* et, si vous avez accès au code source de la version d'origine et le droit de la modifier, il serait préférable de mettre à jour cette définition de type, ce qui, normalement, ne devrait avoir aucun effet sur les clients existants. Mais, si pour une raison ou une autre, vous ne pouvez modifier l'original ou si une déclaration ancrée n'y est pas justifiée (à cause, peut-être, des descendants), redéfinir le type peut vous sortir de ce mauvais pas.

Dans l'héritage de variation **fonctionnelle**, nous changeons le corps de certaines caractéristiques ; si, comme c'est habituellement le cas, les caractéristiques sont déjà effectives, cela revient à changer leur implémentation. La spécification des caractéristiques, déterminée par les assertions, peut également changer. Il est aussi possible, bien que moins fréquent, d'envisager un héritage de variation fonctionnelle entre deux classes retardées ; dans ce cas, les assertions changeront. Cela peut conduire à des changements dans certaines fonctions, retardées ou effectives, utilisées dans les assertions, ou même à l'ajout de nouvelles caractéristiques, tant que cela reste du ressort des "besoins directs des caractéristiques redéfinies", comme le précise la définition.

*Voir "Le principe ouvert-fermé", page 58.*

L'héritage de variation fonctionnelle est l'application directe du principe ouvert-fermé : nous souhaitons adapter une classe existante sans affecter l'original (dont nous n'avons, peut-être, même pas le code source) ou ses clients. Il peut être la source d'abus s'il représente une certaine forme de bidouille : modifier une classe existante afin de l'adapter à un objectif légèrement différent. Quoi qu'il en soit, il s'agira là d'une forme *organisée* de bidouille, évitant les dangers auxquels conduit la modification directe d'un code existant et que l'on a évoqué lors de l'étude

du principe ouvert-fermé. Mais, si vous avez accès au code source de la classe d'origine, vous devriez envisager la possibilité de réorganiser la hiérarchie d'héritage en introduisant une classe plus abstraite dont `A` (la variante existante) et `B` (la nouvelle) seraient toutes deux héritières ou descendants propres de même rang.

## Inactivation

***Définition : héritage d'inactivation***

L'héritage d'inactivation s'applique si `B` redéfinit certaines caractéristiques effectives de `A` sous forme de caractéristiques retardées.

L'inactivation n'est pas fréquente, et doit le rester. L'idée de base va à l'encontre de la direction normale de l'héritage, puisque nous nous attendons habituellement à ce que `B` soit plus concrète que `A` (comme dans la prochaine catégorie, la concrétisation, où `A` est retardée et `B` effective ou, tout au moins, moins retardée). Pour cette raison, les débutants devraient laisser de côté l'inactivation. Mais, elle peut être justifiée dans les deux cas suivants :

- en présence d'héritage multiple, il se peut que vous souhaitiez fusionner des caractéristiques héritées de deux parents différents. Si l'une est retardée et l'autre effective, cela se produira automatiquement : dès lors qu'elles ont le même nom (éventuellement après renommage), la version effective servira d'implémentation. Mais, si elles sont toutes deux effectives, vous devrez en inactiver une ; l'implémentation de l'autre prendra alors le dessus ;

*Voir "Règles sur les noms", page 544.*

- une classe réutilisable peut être **trop concrète**, bien que l'abstraction correspondante réponde à vos besoins. L'inactivation éliminera les implémentations indésirables. Avant d'utiliser cette solution, considérez l'alternative : il est préférable de réorganiser la hiérarchie d'héritage afin que la classe plus concrète hérite d'une nouvelle classe retardée plutôt que de faire l'inverse. Mais ce n'est pas toujours possible si, par exemple, vous n'avez pas le droit de modifier `A` ou sa hiérarchie d'héritage. Inactiver peut, dans des cas semblables, fournir une forme pratique de généralisation.

Pour un lien d'inactivation, `B` sera retardée ; `A` sera, normalement, effective, mais pourrait être partiellement retardée.

## Héritage de concrétisation

Enfin, le troisième et dernier groupe, l'héritage logiciel.

***Définition : héritage de concrétisation***

L'héritage de concrétisation s'applique si `A` représente une catégorie générale de structures de données et `B` un choix partiel ou complet d'implémentations pour des structures de données de ce style. `A` est retardée ; `B` peut être effective ou retardée, ce qui permet d'envisager une concrétisation ultérieure via ses propres héritiers.

Un exemple, utilisé à plusieurs reprises dans les chapitres précédents, est celui d'une classe retardée *TABLE* décrivant les tables de manière très générale. La concrétisation conduit aux héritiers *SEQUENTIAL_TABLE* et *HASH_TABLE*, qui restent retardés. La concrétisation finale de *SEQUENTIAL_TABLE* conduit aux classes effectives *ARRAYED_TABLE*, *LINKED_TABLE*, *FILE_TABLE*.

> Le terme anglais traduit ici par "concrétisation", "reification", issu du latin et signifiant "en faire une chose", vient des analyses sociologiques de Georg Lukács. En informatique, il est utilisé par la méthode de spécification et de développement VDM.

## Héritage de structure

***Définition : héritage de structure***

L'héritage de structure s'applique si *A*, une classe retardée, représente une propriété structurelle générale et *B*, qui peut être retardée ou effective, représente un certain type d'objets possédant cette propriété.

D'habitude, *A* représente une propriété mathématique partagée par un certain ensemble d'objets ; par exemple, *A* peut être la classe *COMPARABLE*, munie d'opérations comme **infix** "<" et **infix** ">=" et représentant des objets soumis à une relation d'ordre total. Une classe ayant besoin d'une relation d'ordre personnelle, comme *STRING*, héritera de *COMPARABLE*.

*Voir "Valeurs numériques et comparables", page 506.*

Il est fréquent qu'une classe hérite de plusieurs parents de ce style. Par exemple, la classe *INTEGER* de la bibliothèque Kernel hérite de *COMPARABLE* et de *NUMERIC* (possédant des caractéristiques comme **infix** "+" et **infix** "*"), représentant ses propriétés arithmétiques. (La classe *NUMERIC* représente, plus précisément, la notion mathématique d'anneau.)

Quelle est la différence entre structure et concrétisation ? Dans l'héritage de concrétisation, *B* représente la même notion que *A*, avec un engagement d'implémentation plus marqué ; avec l'héritage de structure, *B* représente une abstraction en soi, dont *A* ne recouvre qu'un aspect, comme la présence d'une relation d'ordre ou d'opérations arithmétiques.

Waldén et Nerson ont remarqué que les débutants sont persuadés qu'ils utilisent une forme similaire d'héritage quand ils ont, en fait, confondu une relation "contient" avec une relation "est" — comme lorsque *AIRPLANE* hérite de *VENTILATION_SYSTEM*, une variante du schéma "voiture-propriétaire", et tout autant erronée. Ils signalent qu'il est facile d'éviter cette erreur en utilisant un critère de type "absolu", ne laissant planer aucune ambiguïté :

*Extrait (avec changement d'exemple) de* [Waldén 1995]*, pages 193-194.*

> *En présence d'un schéma d'héritage, bien que les propriétés héritées soient secondaires, ce sont toujours des propriétés propres aux objets de la classe* ***dans leur ensemble****. Si nous faisons hériter* *AIRPLANE* *de* *COMPARABLE* *pour prendre en compte une relation d'ordre sur les avions, les caractéristiques héritées s'appliquent à chaque avion pris comme un tout ; ce n'est, en revanche, pas le cas des caractéristiques de* *VENTILATION_SYSTEM*. *La caractéristique* `stop` *de* *VENTILATION_SYSTEM* *n'a pas pour but d'arrêter l'avion.*

La conclusion de cet exemple est évidente : *AIRPLANE* doit être un client et non un héritier de *VENTILATION_SYSTEM*.

*"HÉRITAGE D'IMPLÉMENTATION", 24.8, page 814.*

## Héritage d'implémentation

L'héritage d'implémentation est évoqué en détail dans la suite de ce chapitre. Un cas fréquent est celui du "mariage d'intérêt", fondé sur l'héritage multiple, dans lequel un des parents fournit la

spécification (héritage de concrétisation) tandis que l'autre fournit l'implémentation (héritage d'implémentation) :

***Définition : héritage d'implémentation***

L'héritage d'implémentation s'applique si `B` obtient de `A` un ensemble de caractéristiques (qui ne sont ni des attributs constants ni des fonctions à exécution unique) nécessaires à l'implémentation de l'abstraction associée à `B`. `A` et `B` doivent être effectives.

Le cas de l'héritage d'attributs constants ou de fonctions à exécution unique est traité par la variante suivante.

## Héritage de service

L'héritage de service correspond au cas où le parent est une collection de caractéristiques ayant pour seule vocation d'être utilisées par les descendants :

***Définition : héritage de service***

L'héritage de service s'applique si `A` n'existe que pour fournir un ensemble de caractéristiques liées logiquement entre elles pour le seul bénéfice d'héritiers comme `B`. Il existe deux variantes fréquentes :

- *l'héritage de constante*, dans lequel les caractéristiques de `A` sont toutes des constantes ou des fonctions à exécution unique décrivant des objets partagés ;
- *l'héritage de machine*, dans lequel les caractéristiques de `A` sont des routines pouvant être considérées comme des opérations d'une machine abstraite.

Un exemple d'héritage de service est celui de la classe `EXCEPTIONS`, classe utilitaire fournissant un ensemble de services permettant d'accéder aux détails du mécanisme de traitement des exceptions.

*Voir "TRAITEMENT AVANCÉ DES EXCEPTIONS", 12.6, page 418.*

Parfois, comme dans les exemples donnés en début de chapitre, un lien de type service n'utilise qu'une des deux variantes, constante ou machine ; mais, dans d'autres, comme `EXCEPTIONS`, la classe parent fournit à la fois constantes (comme le code d'exception `Incorrect_inspect_value`) et routines (comme `trigger` pour lever une exception définie par le développeur). Puisque cette étude a pour but d'introduire des catégories disjointes d'héritage, nous devrions traiter l'héritage de service comme une catégorie unique — ayant deux variantes (non disjointes).

Avec l'héritage de constante, `A` et `B` sont toutes deux effectives. Avec l'héritage de machine, on peut être plus souple, mais `B` devrait être au moins aussi effective que `A`.

L'héritage de service est évoqué plus en détail dans la suite de ce chapitre.

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

## Utiliser l'héritage avec des classes retardées et effectives

Chacune des catégories présentées ici impose certaines exigences quant à savoir qui de l'héritier ou du parent doit être retardé ou effectif. Le tableau suivant résume ces règles. "Variation" recouvre variation de type et variation fonctionnelle. Les éléments indiqués par un • sont présents en plusieurs endroits.

***Héritier et parent retardés et effectifs***

| *Héritier* ↓ *Parent* → | *Retardé* | *Effectif* |
|---|---|---|
| **Retardé** | Constant•<br>Restriction•<br>Structure•<br>Sous-type•<br>Inactivation•<br>Variation•<br>Vue | Extension•<br>Inactivation• |
| **Effectif** | Constant•<br>Concrétisation<br>Structure•<br>Sous-type• | Constant•<br>Extension•<br>Implémentation<br>Restriction•<br>Variation• |

## 24.6 UN MÉCANISME, OU PLUSIEURS ?

*"La perspective duale", page 478*; *"Les deux styles", page 590.*

(Note : cette partie suppose que le lecteur a déjà eu connaissance des présentations précédentes concernant "Le sens de l'héritage", en particulier la section intitulée "La perspective duale", et la rétention de descendant, en particulier la section intitulée "Les deux styles", y compris sa table de résumé.)

La grande diversité des utilisations de l'héritage mise en évidence par l'étude précédente pourrait nous inciter à introduire des mécanismes de langage plus nombreux pour couvrir les notions sous-jacentes. En particulier, un certain nombre d'auteurs ont suggéré de séparer l'héritage de *module*, qui est essentiellement un outil permettant à un nouveau module de réutiliser des caractéristiques existantes, et l'héritage de *type*, qui est, en fait, un mécanisme de classification de types.

Une telle division serait plus une source d'ennuis qu'autre chose, et ce pour plusieurs raisons.

Tout d'abord, ne distinguer que deux catégories n'est pas représentatif de la variété des utilisations de l'héritage soulignée par la classification précédente. Puisque personne ne propose d'introduire dix mécanismes différents d'héritage, le résultat serait trop restrictif.

En pratique, cela conduirait à d'inutiles tergiversations méthodologiques : supposez que vous souhaitiez hériter d'une classe d'itérateurs comme `LINEAR_ITERATOR` ; devriez-vous utiliser l'héritage de module ou de type ? On peut justifier les deux réponses. Vous perdriez votre temps à arbitrer entre deux mécanismes de langage ; une telle réflexion n'apporte rien à la question de fond — la qualité de votre logiciel et la vitesse à laquelle vous le produisez.

*Exercice E24.8, page 839.*

> Un exercice vous demande d'analyser nos catégories en essayant de voir, pour chacune d'entre elles, si elle se rapproche plus du style "module" ou "type".

Il est également intéressant d'envisager les conséquences qu'aurait une telle division sur la complexité du langage. L'héritage est accompagné d'un certain nombre de mécanismes auxiliaires. La plupart d'entre eux seront nécessaires des deux côtés :

- la *redéfinition* est utile pour le sous-typage (pensez à `RECTANGLE`, qui redéfinit `perimeter` de `POLYGON`) et pour l'extension de modules (le principe ouvert-fermé exige que, lors de l'héritage

d'un module, nous puissions changer ce qui n'est pas adapté à notre nouveau contexte — flexibilité sans laquelle nous perdrions un des principaux attraits de la méthode orientée objet) ;

- le *renommage* est utile dans l'héritage de module. Dire qu'il n'est pas adapté à l'héritage de type (voir [Breu 1995]) semble trop restrictif. Dans le système externe modélisé, les variantes d'une notion donnée peuvent introduire une terminologie spécifique, qu'il est souvent souhaitable que le logiciel respecte. Une classe `STATE_INSTITUTIONS` d'un système d'informations géographiques ou électorales pourrait avoir une classe descendante `LOUISIANA_INSTITUTIONS` représentant les particularités des structures politiques de la Louisiane ; il n'est pas ridicule de s'attendre à ce que la caractéristique `counties`, donnant la liste des comtés d'un État, soit renommée `parishes` dans le descendant, puisque les habitants de la Louisiane appellent paroisse ce que le reste des États-Unis appelle comté ;
- l'*héritage répété* peut se produire dans les deux formes. Puisque nous pouvons nous attendre à ce que l'héritage des seuls modules interdise la substitution polymorphe, les problèmes d'ambiguïté que pose la liaison dynamique, et donc la nécessité d'une clause `select`, ne se poseront que pour l'héritage de type ; mais toutes les autres questions, en particulier l'opportunité de partager ou de dupliquer des caractéristiques héritées de manière répétitive, se posent encore ;
- comme toujours quand nous introduisons de nouveaux mécanismes dans un langage, ils entrent en interaction avec le reste et entre eux. Allons-nous interdire à une classe d'hériter d'une même classe à la fois comme module et comme type ? Si tel est le cas, nous ne faisons que brimer les développeurs ayant de bonnes raisons d'utiliser la même classe de deux manières différentes ; sinon, nous ouvrons une nouvelle boîte de Pandore pleine de nouvelles questions de langage — conflits de noms, conflits de redéfinition, etc.

Tout cela pour bénéficier d'une vision de puriste de l'héritage — restrictive et controversée. Remarquez qu'il n'y a rien de mal à soutenir des vues controversées ; mais il faut être prudent avant d'en imposer les conséquences aux utilisateurs du langage — c'est-à-dire à tout le monde. Quand il y a doute, abstenez-vous. À nouveau, le contraste avec l'excommunication du `goto` par Dijkstra est frappant : Dijkstra a pris grand soin d'expliquer en détail les inconvénients de l'instruction `goto`, en s'appuyant sur une théorie de la construction et de l'exécution logicielles, et d'évoquer les mécanismes de remplacement disponibles. Dans le cas présent, aucun argument majeur — tout au moins, aucun que j'ai pu discerner — ne montre pourquoi l'utilisation d'un mécanisme unique pour définir à la fois héritage de module et de type est préjudiciable.

*À propos du conseil de Dijkstra, voir "La nécessité de règles de conduite en méthodologie", page 646.*

Si on laisse de côté les condamnations péremptoires fondées sur des idées préconçues de ce que doit être l'héritage, il n'y a qu'une seule objection à l'utilisation d'un mécanisme unique ; la complication supplémentaire qu'induit cette approche sur la **vérification statique de type**. Cette question a été abordée en détail dans le chapitre 17 ; elle accroît la charge de travail des *compilateurs*, ce qui est toujours justifiable (quand cette surcharge reste raisonnable, comme ici) si l'effet en est de faciliter la tâche du *développeur*.

En définitive, cette étude montre que la possibilité de n'utiliser qu'un unique mécanisme pour décrire héritage de module et héritage de type n'est pas — contrairement à ce qu'insinuent les partisans de mécanismes séparés — la conséquence d'une confusion des genres. C'est le résultat de la *toute première décision* de la construction logicielle orientée objet : l'unification des concepts de module et de type dans une notion unique, la classe. Si nous acceptons que les classes soient à la fois modules et types, nous devrions alors accepter l'héritage comme, à la fois, accumulation de modules et sous-typage.

## 24.7 HÉRITAGE DE SOUS-TYPE ET RÉTENTION DE DESCENDANT

La première catégorie de notre liste est probablement la seule forme sur laquelle tout le monde est d'accord, tout au moins parmi ceux qui acceptent l'héritage : ce que nous pouvons appeler l'héritage pur de sous-type.

L'essentiel de ce qui suit s'appliquera également à l'héritage de restriction, dont la principale différence avec l'héritage de sous-type est qu'il n'impose pas que le parent soit retardé.

### Définir un sous-type

Comme cela a été indiqué lors de l'introduction de l'héritage, une partie de la puissance de ce concept vient de la fusion d'un mécanisme de type, en l'occurrence la définition d'un nouveau type comme cas spécial de types existants, et d'un mécanisme de module, la définition d'un module comme extension de modules existants. Les nombreuses controverses sur l'héritage découlent des conflits apparents qui existent entre ces deux visions. L'héritage de sous-type ne soulève aucune question de ce genre — bien que, comme nous le verrons, cela n'implique pas que tout coule de source.

L'héritage de sous-type est fortement inspiré des principes taxonomiques des sciences mathématiques et naturelles. Tout vertébré est un animal ; tout mammifère est un vertébré ; tout éléphant est un mammifère. Chaque groupe (en mathématiques) est un monoïde ; tout anneau est un groupe ; tout champ est un anneau. Des exemples du même style, que nous avons vus lors des chapitres précédents, abondent dans le logiciel orienté objet :

- *FIGURE* ← *CLOSED_FIGURE* ← *POLYGON* ← *QUADRANGLE* ← *RECTANGLE* ← *SQUARE*
- *DEVICE* ← *FILE* ← *TEXT_FILE*
- *SHIP* ← *LEISURE_SHIP* ← *SAILBOAT*
- *ACCOUNT* ← *SAVINGS_ACCOUNT* ← *FIXED_RATE_ACCOUNT*

et ainsi de suite. Dans chacun de ces liens de sous-type, nous avons clairement indiqué l'ensemble d'objets que décrit le type du parent ; et nous avons distingué un sous-ensemble de ces objets, caractérisés par certaines propriétés qui ne s'appliquent pas forcément à toutes les instances du parent. Par exemple, un fichier de texte est un fichier, mais il a la propriété supplémentaire d'être formé d'une séquence de caractères — une propriété que ne possèdent pas d'autres fichiers, comme les fichiers binaires exécutables.

Une règle générale de l'héritage de sous-type est que les divers héritiers d'une classe représentent des ensembles disjoints d'instances. Aucune figure fermée n'est, par exemple, à la fois un polygone et une ellipse.

> Plusieurs exemples, comme *RECTANGLE* ← *SQUARE*, feront très vraisemblablement intervenir un parent effectif, et sont ainsi des cas d'héritage de restriction.

### Vues multiples

L'héritage de sous-type est simple quand il existe un critère précis permettant de classer les variantes d'une notion donnée. Mais, parfois, plusieurs qualités attirent notre attention. Même

dans un exemple apparemment simple comme la classification des polygones, le doute subsiste : devrions-nous utiliser le nombre de côtés, ce qui conduirait à des héritiers comme *TRIANGLE*, *QUADRANGLE*, etc., ou diviser nos objets en polygones réguliers (*EQUILATERAL_POLYGON*, *SQUARE* et ainsi de suite) et irréguliers ?

Plusieurs stratégies permettent d'aborder de tels conflits. Elles seront évoquées lors de l'étude de l'héritage de vue, plus loin dans ce chapitre.

## Imposer la vision de sous-type

Un type n'est pas, bien sûr, un simple ensemble d'objets : il est également caractérisé par les opérations applicables (les caractéristiques) et leurs propriétés sémantiques (les assertions : préconditions, postconditions, invariants). Nous nous attendons à ce que le sort des caractéristiques et des assertions soit, dans l'héritier, compatible avec le concept de sous-type — c'est-à-dire nous permettant de considérer toute instance de l'héritier comme une instance du parent.

Les règles concernant les assertions supportent, effectivement, cette vue de sous-type :

- l'invariant du parent est automatiquement partie intégrante de l'invariant de l'héritier ; toutes les contraintes ainsi spécifiées pour les instances du parent s'appliquent également à l'héritier ;
- une précondition de routine s'applique, éventuellement après affaiblissement, à toute redéclaration de la routine : tout appel qui remplit l'exigence requise pour les instances du parent remplit également l'exigence (égale ou plus faible) spécifiée pour les instances de l'héritier ;
- une postcondition de routine s'applique, éventuellement après renforcement, à toute redéclaration de la routine : ainsi, toute propriété du résultat de la routine concernant les instances du parent sera également vraie, du fait des propriétés (égales ou plus fortes) spécifiées pour les instances de l'héritier.

En ce qui concerne les caractéristiques, la situation est un peu plus délicate. La vue de sous-type implique que toute opération applicable à une instance du parent puisse être appliquée à une instance de l'héritier. C'est toujours vrai de manière interne : même dans l'héritage de *ARRAY* par *ARRAYED_STACK*, qui semble éloigné de l'héritage de sous-type, les caractéristiques de *ARRAY* étaient encore applicables à l'héritier et étaient, en fait, essentielles à l'implémentation de ses caractéristiques de *STACK*. Mais, dans ce cas, nous avions caché toutes ces caractéristiques *ARRAY* aux clients de l'héritier, et pour une bonne raison (nous ne souhaitions pas qu'un client d'une classe de pile puisse effectuer des opérations arbitraires sur la représentation, comme modifier directement un élément de tableau, car cela serait une violation de l'interface de classe).

Dans le cas d'héritage pur de sous-type, nous pourrions nous attendre à une règle bien plus forte : que *toute* caractéristique pouvant être appliquée par un client aux instances de la classe parent soit aussi applicable, par ce même client, aux instances de l'héritier. En d'autres termes, pas de rétention de descendant : si `B` hérite `f` de `A`, le statut d'exportation de `f` dans `B` est au moins aussi généreux que dans `A`. (C'est-à-dire : si `f` était exportée en général, elle l'est toujours ; et si elle était sélectivement exportée à certaines classes, elle est toujours exportée vers celles-ci, bien qu'elle puisse être exportée vers d'autres.)

## Utiliser la rétention de descendant

Dans un monde parfait, nous pourrions effectivement imposer la règle d'absence de rétention de descendant ; mais pas dans le monde réel du développement logiciel. L'héritage doit être utilisable même avec des classes écrites par des personnes n'ayant pas une vue d'ensemble parfaite ; certaines caractéristiques d'une classe peuvent ne pas avoir de sens dans un descendant écrit ultérieurement par quelqu'un d'autre et dans un contexte complètement différent. Nous pouvons appeler ces cas des **exceptions de taxonomie**. (Dans un contexte différent, le mot "exception" suffirait, mais nous ne voulons pas créer d'ambiguïté avec la notion logicielle de traitement d'exception étudiée dans les chapitres précédents.)

Devrions-nous renoncer à hériter d'une classe utile et attirante parce que quelques-unes de ses caractéristiques ne sont pas applicables à nos propres clients ? Cela ne serait pas raisonnable. Nous cacherons simplement les caractéristiques à la vue de nos clients, et continuerons notre travail.

Les alternatives envisagées lors de l'étude d'un des principes fondateurs de la technologie objet — le **principe ouvert-fermé** — ne sont pas attirantes :

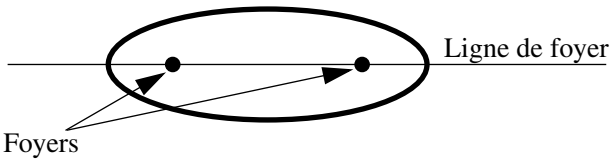
- nous pourrions modifier la classe d'origine. Cela peut nous conduire à invalider des multitudes de systèmes existants qui comptent dessus — non, merci. Dans la plupart des cas pratiques, nous ne pourrons pas, de toute façon, modifier la classe ; il se peut que nous n'ayons même pas accès à son code source ;
- nous pourrions écrire une nouvelle version de la classe (ou, si nous avons de la chance et accès au code source, en faire une copie) et la modifier. Cette approche est à l'opposé de tout ce que promeut la technologie objet ; elle annihile toute tentative de réutilisabilité et de processus logiciel organisé.

## Éviter la rétention de descendant

Avant d'étudier les conditions dans lesquelles nous pourrons envisager d'utiliser la rétention de descendant, il est essentiel de remarquer que, la plupart du temps, nous n'en avons pas besoin. La rétention d'information ne devrait être utilisée que quand tout le reste a échoué. Quand vous pouvez adapter la structure d'héritage suffisamment tôt dans le processus de conception, les *préconditions* offrent une meilleure technique pour traiter les exceptions de taxonomie apparentes.

Considérez la classe `ELLIPSE`. Une ellipse a deux foyers par lesquels vous pouvez habituellement tracer une ligne :

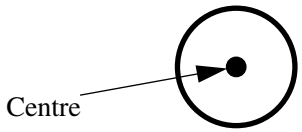
***Une ellipse et sa ligne de foyer***




La classe `ELLIPSE` pourrait donc offrir une caractéristique `focus_line`.

Il est tout à fait normal de définir la classe `CIRCLE` comme héritière de `ELLIPSE` : tout cercle est également une ellipse. Mais, dans un cercle, les deux foyers correspondent au même point — le

centre du cercle — et il n'y a donc pas de ligne de foyer. (Il est peut-être plus exact de dire qu'il y a une infinité de lignes de foyer, passant toutes par le centre, mais, en pratique, l'effet est le même.)




***Un cercle et son centre***

Est-ce un bon exemple de rétention de descendant ? En d'autres termes, est-ce que la classe *CIRCLE* devrait rendre secrète la caractéristique *focus_line*, comme dans :

```
class CIRCLE inherit
      ELLIPSE
            export {NONE} focus_line end
      ...
```

Probablement pas. Dans ce cas, le concepteur de la classe parent a toutes les informations à sa disposition pour se rendre compte que *focus_line* n'est pas applicable à toutes les ellipses. En supposant que la caractéristique soit une routine, il devrait y avoir une précondition :

```
focus_line is
            -- La ligne passant par les deux foyers
      require
            not equal (focus_1, focus_2)
      do
            ...
      end
```

(La précondition pourrait également être abstraite, utilisant une fonction *distinct_focuses* ; ceci présenterait l'avantage que *CIRCLE* pourrait redéfinir cette fonction une fois pour toutes pour renvoyer faux.)

Ici, la nécessité de traiter les ellipses sans ligne de foyer découle d'une analyse fine du problème. Écrire une classe d'ellipse munie d'une fonction *focus_line* n'ayant pas de précondition serait une erreur de conception ; prendre en compte une telle erreur par rétention de descendant reviendrait à essayer de camoufler cette erreur. Comme on l'a signalé à la fin de la présentation du principe ouvert-fermé, les erreurs de conception doivent être corrigées et non rafistolées par les descendants.

*Voir page 62.*

## Applications de la rétention de descendant

L'exemple de *focus_line* est caractéristique des exceptions de taxonomie se produisant dans des domaines d'application comme les mathématiques, qui peuvent se vanter de posséder une théorie solide associée à des classifications patiemment ajustées au cours des temps. Dans un tel contexte, la bonne réponse consiste à utiliser une précondition, concrète ou abstraite, à l'endroit où se trouve la caractéristique d'origine.

Mais cette technique n'est pas toujours applicable, en particulier dans des domaines régis par des processus humains, dont les caprices rendent la prévision de toutes les exceptions possibles souvent difficile.

Considérez, par exemple, une hiérarchie de classes d'hypothèques dont la racine soit la classe *MORTGAGE*, utilisée dans un système logiciel de gestion d'hypothèques. Les descendants ont été

organisés selon divers critères, comme la présence d'un taux fixe ou variable, l'utilisation à des fins professionnelles ou privées ou toute autre particularité considérée comme pertinente ; nous pouvons supposer, pour simplifier, que cette taxonomie est à base de sous-typage pur. La classe *MORTGAGE* contient une procédure *redeem*, qui gère les mécanismes de remboursement d'un prêt avant échéance.

Supposez maintenant que l'assemblée, dans un élan de générosité (ou sous l'influence des groupes de pression de l'industrie du bâtiment), introduise une nouvelle forme d'hypothèque garantie par le gouvernement qui, sous prétexte d'autres conditions avantageuses, interdit tout remboursement anticipé. Nous avons déterminé une place adéquate pour la classe correspondante *NEW_MORTGAGE* ; mais que faire de la procédure *redeem* ?

Nous pourrions utiliser la technique présentée pour *focus_line* : une précondition. Mais, que se passe-t-il s'il n'y a jamais eu, de mémoire de banquier, d'hypothèque non remboursable avant échéance ? La procédure *redeem* n'aura probablement pas de précondition. (La situation serait la même si la précondition existait sous forme concrète, car elle ne pourrait être redéfinie.)

Ainsi, si nous décidons d'utiliser une précondition, nous devons modifier la classe *MORTGAGE*. Comme toujours, cela suppose que nous ayons accès au code source et le droit de le modifier — ce qui est rarement le cas. Supposons, cependant, que cela ne soit pas un problème. Nous ajoutons à *MORTGAGE* une fonction à valeur booléenne *redeemable* et à *redeem* une clause :

```
require
        redeemable
```

Mais nous avons alors modifié l'interface de la classe. Tous les clients de la classe et ses nombreux descendants ont été, en un instant, rendus potentiellement incorrects ; pour être conformes à la spécification, tous les appels *m.redeem (...)* devraient, dorénavant, être écrits :

```
if m.redeemable then
        m.redeem (...)
else
        ... (Qu'allons-nous donc pouvoir mettre ici ?) ...
end
```

Ce changement n'est, au début, pas urgent, car le danger n'est que potentiel : le logiciel existant n'utilisera que les descendants existants de *MORTGAGE*, ce qui ne crée aucun risque. Mais ne pas les adapter revient à laisser dans notre logiciel une bombe à retardement — des appels non protégés à une routine munie d'une précondition. Dès qu'un développeur client songera à utiliser un attachement polymorphe avec une source de type *NEW_MORTGAGE*, mais oubliera le test, nous serons en présence d'une bogue. Et le compilateur ne produira aucun diagnostic.

L'absence de précondition dans la version originale de *redeem* n'était pas une erreur de la part des concepteurs d'origine : dans leur vision du monde, qui était jusqu'à présent correcte, il n'y avait nul besoin de précondition. Toute hypothèque était remboursable. Nous ne pouvons pas imposer à chaque caractéristique d'avoir une précondition, car chaque fonction *f* utile serait alors accompagnée d'une fonction à valeur booléenne *f_feasible* lui servant de garde du corps ; nous ne pourrions alors plus jamais écrire un simple *x.f* de toute notre vie ; chaque appel serait inclus dans un *if ...* ou autre construction équivalente, comme l'a montré ci-dessus *m.redeem*. On a vu mieux.

L'exemple *redeem* est caractéristique des exceptions de taxonomie qui, contrairement à *focus_line* et aux autres cas de classification a priori parfaite, ne peuvent être pris en compte de manière anticipée, aussi soignée que soit la conception *initiale* des préconditions. L'observation

faite précédemment s'applique ici : il serait absurde de renoncer à l'héritage — la réutilisation d'une riche structure de classe, développée avec amour et soigneusement validée — parce qu'une ou deux caractéristiques parmi des douzaines d'autres ne s'appliqueraient pas à notre objectif du moment. Nous devrions simplement utiliser la rétention de descendant :

```
class NEW_MORTGAGE inherit
      MORTGAGE
            export {NONE} redeem end
      ...
```

Aucune erreur ou anomalie ne sera introduite dans le logiciel existant — la structure de classe existante ou ses clients. Si quelqu'un modifie une classe client pour y ajouter un attachement polymorphe ayant une source de type *NEW_MORTGAGE* et si la cible de cet attachement est également utilisée avec *redeem*, comme dans :

```
m: MORTGAGE; nm: NEW_MORTGAGE
...
m := nm
...
m.redeem (...)
```

l'appel devient un appel CAT et l'erreur potentielle sera détectée statiquement par le mécanisme étendu décrit lors de notre étude du typage.

*"ATTENTION AUX APPELS CAT POLYMORPHES !", 17.7, page 618.*

## Les taxonomies et leurs limitations

Les exceptions de taxonomie ne sont pas spécifiques aux exemples logiciels. Même — ou, peut-être, en particulier — dans les domaines les mieux établis des sciences naturelles, il est parfois impossible de trouver une phrase de la forme "*les membres du phylum ABC* [ou genre, espèce, etc.] *sont caractérisés par la propriété XYZ*" qui ne soit précédée de "*la plupart des*", qualifiée par "*habituellement*" ou suivie de "*sauf dans certains cas*". C'est le cas de tous les niveaux de la hiérarchie, même des catégories les plus fondamentales, qu'un profane pourrait, naïvement, croire fondées sur des critères inattaquables !

Si vous croyez, par exemple, que la distinction entre les règnes animal et végétal est simple, jetez donc un coup d'œil à la définition qu'en donne un ouvrage de référence (les italiques ont été ajoutées) :

> DISTINGUER LES PLANTES DES ANIMAUX
>
> Il y a plusieurs facteurs *généraux* permettant de distinguer les plantes des animaux, *bien qu*'il existe de *nombreuses exceptions*.
>
> **Locomotion** *La plupart* des animaux se déplacent librement, *alors qu*'il est *rare* de trouver des plantes capables de se déplacer dans leur environnement. *La plupart* des plantes sont enracinées dans le sol ou attachées à des rochers, à du bois ou à d'*autres matériaux*.
>
> **Nourriture** Les plantes vertes qui contiennent de la chlorophylle élaborent elles-mêmes leur nourriture, mais *la plupart* des animaux obtiennent leurs aliments en ingurgitant des plantes ou d'autres animaux. […]
>
> **Croissance** Les plantes croissent, *habituellement*, le long des extrémités de leurs branches et racines et de la couche extérieure de leur tige tout au long de leur existence. Les animaux développent, *généralement*, toutes les parties de leur corps et arrêtent de grandir après la maturité.
>
> **Régulation chimique** Bien que plantes et animaux possèdent, *en général*, des hormones et autres composants chimiques qui régulent *certaines* actions de leur organisme, la composition chimique de ces hormones diffère dans les deux règnes.

*D'après* : New York Public Library Science Desk Reference, ed. *Patricia Barnes-Svarney, 1995*.

On peut appliquer les mêmes commentaires à un autre domaine d'étude, celui-ci culturel et non plus naturel, qui a également contribué au développement d'une taxonomie systématique : la classification historique des langages humains.

En zoologie, un exemple courant, tellement connu des spécialistes de l'intelligence artificielle qu'il en est devenu une vraie tarte à la crème, fournit néanmoins une bonne illustration des exceptions de taxonomie. (Souvenez-vous, cependant, que ce n'est qu'une *analogie* et non un exemple logiciel, qui ne peut, donc, prouver quoi que ce soit ; elle ne peut que nous aider à mieux appréhender des idées dont la pertinence a été justifiée par ailleurs.) Les oiseaux volent ; en termes logiciels, la classe `BIRD` aurait une procédure `fly`. Et pourtant, si nous souhaitions une classe d'autruche `OSTRICH`, il nous faudrait admettre que les autruches, qui sont, à n'en pas douter, typiquement des oiseaux, ne volent pas.

Nous pourrions envisager de classer les oiseaux en catégories volante et non volante. Mais cela rentrerait en conflit avec d'autres critères possibles, y compris, en particulier, celui que l'on retient le plus souvent, indiqué ci-dessous.

L'exemple `OSTRICH` prend un tour intéressant. Bien que la plupart d'entre elles ne semblent malheureusement pas s'en rendre compte, les autruches devraient vraiment voler. Les générations précédentes ont perdu cette aptitude ancestrale par un accident de l'évolution, mais, anatomiquement, les autruches ont préservé l'essentiel de la machinerie aéronautique des oiseaux. Cette caractéristique, qui rend un peu plus difficile le travail du taxonomiste professionnel (bien qu'elle puisse rendre plus facile celui de son collègue taxidermiste), ne l'empêchera pas, finalement, de classer les autruches parmi les oiseaux.

En termes logiciels, `OSTRICH` héritera simplement de `BIRD` et masquera la caractéristique héritée `fly`.

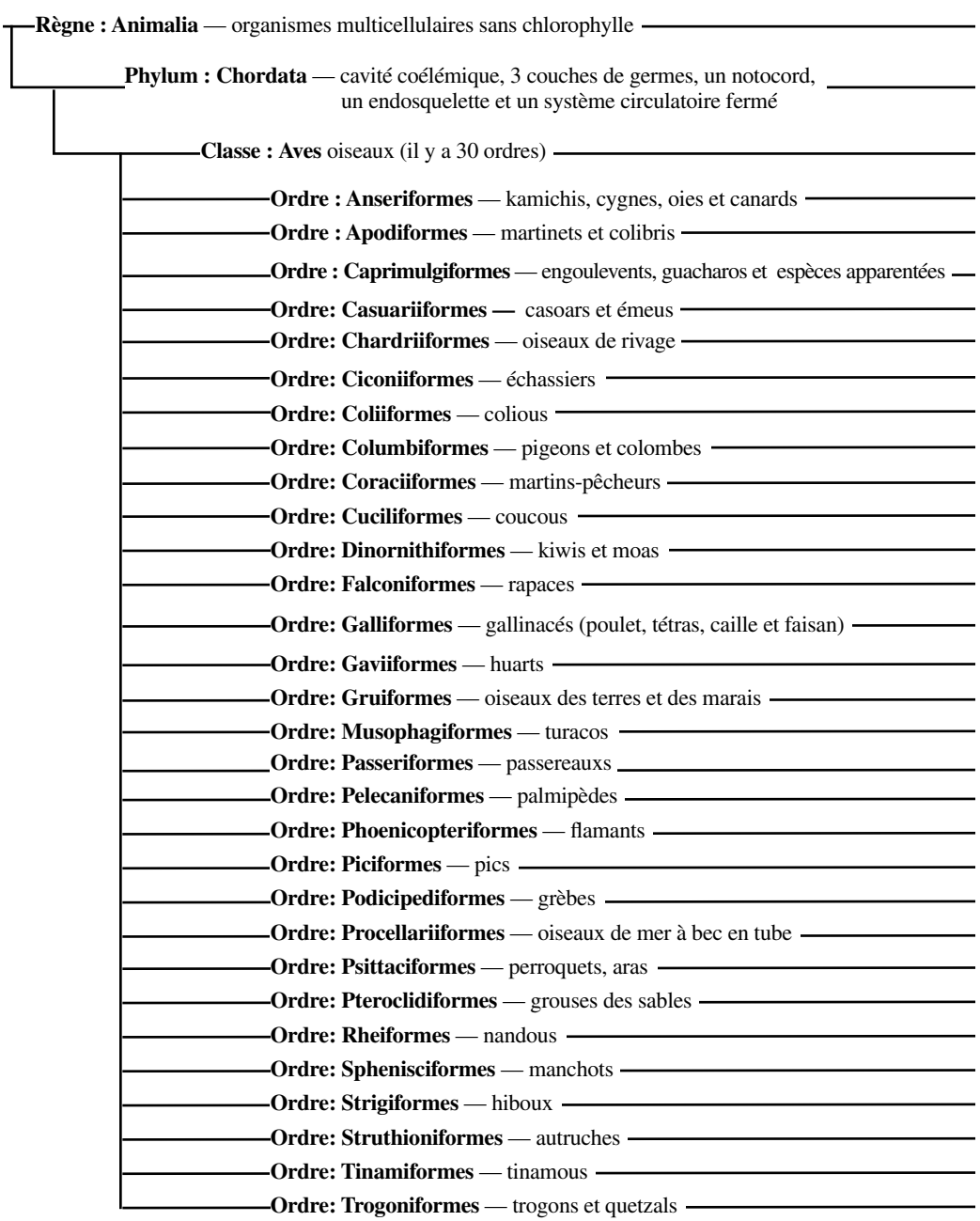
## Utiliser la rétention de descendant

Les observations précédentes, tirées de la pratique logicielle et d'analogies non logicielles, indiquent que, même en présence d'une conception soignée, des exceptions de taxonomie subsisteront. Cacher `redeem` de `NEW_MORTGAGE` ou `fly` de `OSTRICH` n'est pas nécessairement le signe d'une conception bâclée ou d'un manque d'anticipation ; c'est reconnaître que les hiérarchies d'héritage alternatives n'ayant pas besoin de rétention de descendant seraient plus complexes et moins utiles.

De telles exceptions de taxonomie existent depuis des siècles, malgré les efforts de géants intellectuels (y compris Aristote, Linné, Buffon, Jussieu et Darwin). Elles peuvent même être le signe d'une limitation intrinsèque de la capacité humaine à appréhender le monde. Pourraient-elles être liées aux résultats de non-déterminisme qui ont ébranlé la pensée scientifique au XX$^{e}$ siècle, incertitude en physique et indécidabilité en mathématiques ?

Tout ceci suppose que la rétention de descendant reste, comme nous l'avons noté, peu fréquente. Si vous concevez une taxonomie remplie d'exceptions de taxonomie — eh bien, ce ne sont plus des exceptions et vous êtes donc loin d'une vraie taxonomie.

En ce qui concerne le logiciel, dans les rares cas où les critères de classification sont en conflit ou quand la présence d'un important travail antérieur empêche tout production de hiérarchie parfaite de sous-type, la rétention de descendant est plus qu'un simple outil pratique : elle est votre bouée de sauvetage.


**Règne : Animalia** — organismes multicellulaires sans chlorophylle
**Phylum : Chordata** — cavité coélémique, 3 couches de germes, un notocord, un endosquelette et un système circulatoire fermé
**Classe : Aves** oiseaux (il y a 30 ordres)
**Ordre : Anseriformes** — kamichis, cygnes, oies et canards
**Ordre : Apodiformes** — martinets et colibris
**Ordre : Caprimulgiformes** — engoulevents, guacharos et espèces apparentées
**Ordre: Casuariiformes** — casoars et émeus
**Ordre: Chardriiformes** — oiseaux de rivage
**Ordre: Ciconiiformes** — échassiers
**Ordre: Coliiformes** — colious
**Ordre: Columbiformes** — pigeons et colombes
**Ordre: Coraciiformes** — martins-pêcheurs
**Ordre: Cuciliformes** — coucous
**Ordre: Dinornithiformes** — kiwis et moas
**Ordre: Falconiformes** — rapaces
**Ordre: Galliformes** — gallinacés (poulet, tétras, caille et faisan)
**Ordre: Gaviiformes** — huarts
**Ordre: Gruiformes** — oiseaux des terres et des marais
**Ordre: Musophagiformes** — turacos
**Ordre: Passeriformes** — passereauxs
**Ordre: Pelecaniformes** — palmipèdes
**Ordre: Phoenicopteriformes** — flamants
**Ordre: Piciformes** — pics
**Ordre: Podicipediformes** — grèbes
**Ordre: Procellariiformes** — oiseaux de mer à bec en tube
**Ordre: Psittaciformes** — perroquets, aras
**Ordre: Pteroclidiformes** — grouses des sables
**Ordre: Rheiformes** — nandous
**Ordre: Sphenisciformes** — manchots
**Ordre: Strigiformes** — hiboux
**Ordre: Struthioniformes** — autruches
**Ordre: Tinamiformes** — tinamous
**Ordre: Trogoniformes** — trogons et quetzals

## 24.8 HÉRITAGE D'IMPLÉMENTATION

L'utilisation d'un lien d'héritage entre une classe décrivant une implémentation d'une structure abstraite de données et la classe fournissant cette implémentation est une forme d'héritage qui a été souvent critiquée, mais qui, en fait, est à la fois pratique et conceptuellement fondée.

### Le "mariage d'intérêt"

*"Le mariage d'intérêt", page 514.*

Lors de l'étude de l'héritage multiple, nous avons vu un exemple de ces "mariages d'intérêt", qui combinent une classe retardée et un mécanisme qui l'implémente. L'exemple était *ARRAYED_STACK*, de forme générale :

```
class ARRAYED_STACK [ G ] inherit
        STACK [ G ]
                redefine change_top end
        ARRAY [ G ]
                rename
                        count as capacity, put as array_put
                export
                        { NONE } all
                end
feature
        ... Implémentation des routines retardées de STACK, comme put, count, full,
            et redéfinition de change_top en termes d'opérations ARRAY ...
end
```

*STACK2 se trouve page 342.*

Il est intéressant de comparer *ARRAYED_STACK*, ébauchée ici, avec la classe *STACK2* vue précédemment — une implémentation de pile définie sans utilisation d'héritage. Remarquez, en particulier, combien le fait d'éviter que la classe soit client de *ARRAY* simplifie la notation (la version précédente devait utiliser *representation*·*put* là où nous pouvons maintenant simplement écrire *put*).

Dans la partie d'héritage ci-dessus concernant *ARRAY*, toutes les caractéristiques ont été rendues secrètes. C'est typique de l'héritage pour mariage d'intérêt : toutes les caractéristiques du parent fournissant la spécification, ici *STACK*, sont exportées ; toutes les caractéristiques du parent fournissant l'implémentation, ici *ARRAY*, sont cachées. Cela oblige les clients de la classe *ARRAY_STACK* à utiliser les instances correspondantes via les seules caractéristiques de pile ; nous ne voulons pas leur permettre d'effectuer des opérations arbitraires de tableau sur la représentation, comme changer la valeur d'un élément autre que celui au sommet.

### C'est si bon, mais est-ce immoral ?

L'héritage d'implémentation ne manque pas de détracteurs. Le fait que nous cachions tant de caractéristiques héritées semble, à certains, une violation du principe "est-un" de l'héritage.

Il n'en est rien. Il y a différentes formes de "est-un". De par son comportement, une pile sous forme de tableau est une pile ; mais, de manière interne, c'est un tableau. En fait, la représentation d'une instance de *ARRAYED_STACK* est exactement la même que celle d'une instance de *ARRAY*, enrichie d'un attribut (*count*). Être fait de la même manière est une forme passablement forte de "est-un". Et il ne s'agit pas de la seule représentation : toutes les caractéristiques de *ARRAY*,

comme *put* (renommée *array_put*), **infix** "@" et *count* (renommée *capacity*) sont accessibles depuis *ARRAYED_STACK*, bien qu'elles ne soient pas exportées à ses clients ; la classe en a besoin pour implémenter les caractéristiques de *STACK*.

Il n'y a donc rien de conceptuellement répréhensible à utiliser un tel héritage à seules fins d'implémentation. La comparaison avec le contre-exemple étudié au début de ce chapitre est frappante : avec *CAR_OWNER*, nous étions en présence d'un manque total de compréhension du concept ; avec *ARRAYED_STACK*, il s'agit d'une forme bien identifiée de relation "est-un".

Il existe cependant un inconvénient : permettre au mécanisme d'héritage de restreindre la disponibilité d'exportation d'une caractéristique héritée — c'est-à-dire autoriser la clause **export** — rend plus difficile la vérification statique de type, comme nous l'avons vu en détail. Mais cette difficulté est en grande partie de la responsabilité de l'auteur du compilateur et non de celle du développeur logiciel.

## Procéder sans héritage

Allons plus loin et regardons ce qu'il faudrait faire pour traiter notre exemple sans héritage d'implémentation. Nous l'avons déjà vu : c'est la classe *STACK2* d'un chapitre précédent. Elle possède un attribut *representation* de type *ARRAY* [ *G* ] et des procédures de pile implémentées dans le style suivant (les assertions sont omises) :

*Page 342.*

```
put (x: G) is
            -- Ajouter x au sommet.
      require
            ...
      do
            count := count + 1
            representation.put (x, count)
      ensure
            ...
      end
```

Chaque manipulation de la représentation nécessite un appel à une caractéristique de *ARRAY* avec pour cible *representation*. Cela a un coût en performance : mineur en ce qui concerne l'espace (l'attribut *representation*), plus sérieux pour le temps (passer par *representation*, c'est-à-dire ajouter une indirection à chaque opération).

Outre la question de performance, notons aussi le côté fastidieux, car il faut ajouter avant toute opération de tableau le préfixe "*representation*.". Cela s'applique à toutes les classes qui implémentent diverses structures de données — piles, mais aussi listes, files et autres — via des tableaux.

Le concepteur orienté objet déteste les tâches pénibles et répétitives. "Encapsuler la répétition" est son slogan. Si un tel schéma se reproduit dans un ensemble de classes, la réaction saine et naturelle est d'essayer de comprendre l'abstraction commune et de l'encapsuler dans une classe. L'abstraction est ici quelque chose comme une "structure de données pouvant accéder à un tableau et à ses opérations". La classe pourrait être :

```
indexing
        description : "Objets pouvant accéder à un tableau et à ses opérations"
class
        ARRAYED [ G]
feature -- Access
        item (i: INTEGER): G is
                        -- L'élément de la représentation d'indice i
                require
                        ...
                do
                        Result := representation.item (i)
                ensure
                        ...
                end
feature -- Element change
        put (x: G; i: INTEGER) is
                        -- Remplacer par x l'élément de la représentation d'indice i.
                require
                        ...
                do
                        representation.put (x, i)
                ensure
                        ...
                end
feature {NONE} -- Implementation
        representation: ARRAY [ G]
end -- class ARRAYED
```

Les caractéristiques *item* et *put* ont été exportées. Puisque *ARRAYED* ne décrit que les propriétés internes d'une structure de données, elle n'a pas vraiment besoin d'exporter des caractéristiques. Ainsi, toute personne ne souhaitant pas autoriser un descendant à cacher certaines caractéristiques exportées par ses parents peut décider de rendre secrètes toutes les caractéristiques de *ARRAYED*. Elles resteront secrètes, par défaut, dans les descendants.

Avec cette définition de classe, toute controverse quant au fait que *ARRAYED_STACK* ou *ARRAYED_LIST* hérite de *ARRAYED* disparaît : ces classes décrivent, effectivement, des structures "tabulées". Elles peuvent alors utiliser *item* à la place de *representation*.*item* et ainsi de suite ; nous avons éliminé le caractère pénible de la répétition.

Mais, alors, si nous pouvons hériter sans problème de *ARRAYED*, pourquoi ne pas hériter directement de *ARRAY* ? La couche supplémentaire d'encapsulation rajoutée à *ARRAY* ne nous apporte rien — une forme d'encapsulation qui commence à ressembler plutôt à de l'obstruction. En passant par *ARRAYED*, nous nous persuadons simplement que nous ne sommes pas en train d'utiliser l'héritage d'implémentation, mais, en réalité, nous rendons le logiciel plus complexe et moins efficace.

Il n'y a, effectivement, aucune raison d'opérer ainsi dans cet exemple de la classe *ARRAYED*. Un héritage d'implémentation direct de classes comme *ARRAY* est simple et légitime.

## 24.9 HÉRITAGE DE SERVICE

Avec l'héritage de service, nous sommes encore moins hypocrites qu'avec l'héritage d'implémentation sur les raisons mêmes de ce mariage : par égoïsme pur et intéressé. Nous voyons une classe munie de caractéristiques avantageuses et nous voulons les utiliser. Mais il n'y a pas de honte à avoir : la classe n'a pas d'autre raison d'être.

### Utiliser les codes caractères

Les bibliothèques Base contiennent une classe *ASCII* :

```
indexing
        description:
                "Le jeu de caractères ASCII.%
                %Cette classe peut être utilisée comme ancêtre par toute classe%
                %ayant besoin de ses services."
class ASCII feature -- Access
        Character_set_size: INTEGER is 128; Last_ascii: INTEGER is 127
        First_printable: INTEGER is 32; Last_printable: INTEGER is 126
        Letter_layout: INTEGER is 70
        Case_diff: INTEGER is 32
                -- Lower_a - Upper_a
        ...
        Ctrl_a: INTEGER is 1; Soh: INTEGER is 1
        Ctrl_b: INTEGER is 2; Stx: INTEGER is 2
        ...
        Blank: INTEGER is 32; Sp: INTEGER is 32
        Exclamation: INTEGER is 33; Doublequote: INTEGER is 34
        ...
        ...
        Upper_a: INTEGER is 65; Upper_b: INTEGER is 66
        ...
        Lower_a: INTEGER is 97; Lower_b: INTEGER is 98
        ... etc. ...
end -- class ASCII
```

Cette classe est un répertoire d'attributs constants (142 caractéristiques en tout) décrivant les propriétés du jeu de caractères ASCII. Comme le stipule l'entrée *description*, elle a pour vocation d'être héritée par les classes ayant besoin d'accéder à de telles propriétés.

Considérez, par exemple, un analyseur lexical — la partie d'un système d'analyse de langage responsable de l'identification des éléments de base, ou *jetons*, d'un texte ; ces jetons peuvent être (en supposant que l'entrée soit un texte d'un langage de programmation quelconque) des constantes entières, des identificateurs, des symboles et ainsi de suite. Une des classes du système, disons *TOKENIZER*, devra accéder aux codes des caractères pour classer les caractères en chiffres, lettres, etc. Une telle classe héritera ces codes de *ASCII* :

```
class TOKENIZER inherit ASCII feature
        ... Les routines peuvent utiliser les caractéristiques comme Blank,
        Case_diff etc. ...
end
```

Des classes comme *ASCII* ont parfois prêté à controverse ; avant d'aborder l'aspect méthodologique de cette étude et voir s'il s'agit là d'une application justifiée de l'héritage, évoquons un autre exemple d'héritage de service.

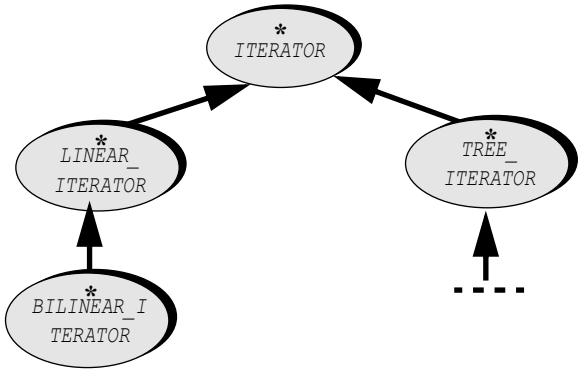
## Itérateurs

Ce second exemple illustre le cas où les caractéristiques héritées ne sont pas simplement des attributs constants (comme dans *ASCII*) mais des routines plus générales.

Supposez que nous souhaitions fournir un mécanisme général pour itérer sur des structures de données d'un certain genre, par exemple des structures linéaires comme les listes. "Itérer" revient à appliquer une procédure, disons *action*, sur les éléments d'une telle structure, pris en ordre séquentiel. On nous demande de fournir un certain nombre de mécanismes d'itération, y compris : appliquer *action* à tous les éléments ; l'appliquer aux seuls éléments qui vérifient un certain critère défini par la fonction à valeur booléenne *test* ; l'appliquer à tous les éléments qui précèdent le premier élément qui vérifie *test* ou le premier qui ne vérifie pas cette condition ; et ainsi de suite. Un système qui utilise ce mécanisme doit pouvoir l'appliquer à n'importe quel *action* et *test* de son choix.

*Exercice E24.7, page 839.*

Il semblerait, à première vue, que ces caractéristiques d'itération doivent appartenir aux classes mêmes des structures de données, comme *LIST* ou *SEQUENCE* ; mais, ce n'est pas la bonne solution, comme un prochain exercice vous invite à le vérifier vous-même. Il est préférable d'introduire une hiérarchie séparée pour les itérateurs :




La classe *LINEAR_ITERATOR*, celle qui nous intéresse ici, ressemble à ceci :

```
indexing
        description:
                "Objets pouvant itérer sur des structures linéaires"
        names: iterators, iteration, linear_iterators, linear_iteration
deferred class LINEAR_ITERATOR [ G] inherit
        ITERATOR [ G]
                redefine target end
feature -- Access
        invariant_value: BOOLEAN is
                        -- La propriété à maintenir lors d'une itération (par défaut : vrai)
                do
```

```
                Result := True
            end
        target: LINEAR [ G]
                -- La structure sur laquelle s'appliqueront les
                -- caractéristiques d'itération
        test: BOOLEAN is
                -- La condition booléenne utilisée pour sélectionner
                -- les éléments applicables
            deferred
            end
    feature -- Basic operations
        action is
                -- L'action à appliquer aux éléments sélectionnés
            deferred
            end
        do_if is
                -- Appliquer action en séquence à chaque article de
                -- target qui vérifie test.
            do
                from start invariant invariant_value until exhausted loop
                    if test then action end
                    forth
                end
            ensure then
                exhausted
            end
        ... Et ainsi de suite : do_all, do_while, do_until etc. ...

    end -- class LINEAR_ITERATOR
```

Supposez maintenant qu'une classe ait besoin d'effectuer une certaine opération sur les éléments sélectionnés d'un type spécifique donné ; par exemple, une classe de commande dans un système de traitement de texte peut avoir besoin de justifier tous les paragraphes d'un document, sauf ceux qui sont formatés au préalable (comme les textes de programme et autres paragraphes d'affichage). Ainsi :

```
    class JUSTIFIER inherit
        LINEAR_ITERATOR [ PARAGRAPH]
            rename
                action as justify,
                test as justifiable,
                do_all as justify_all
            end
    feature
        justify is
            do ... end
        justifiable is
                -- Le paragraphe doit-il être justifié ?
            do
                Result := not preformated
            end
        ...
    end -- class JUSTIFIER
```

Le renommage n'était pas indispensable, mais augmente la clarté. Remarquez qu'il n'est pas nécessaire de déclarer ou de redéclarer la procédure `justify_all` (l'ancien `do_all`) : puisqu'elle est héritée, elle fait le travail attendu, en fonction des versions de `action` et `test` rendues effectives.

La procédure `justify`, au lieu d'être décrite dans la classe, pourrait être héritée d'un autre parent. Dans ce cas, l'héritage multiple effectuerait une opération de "jointure" qui rendrait effective `action`, retardée et héritée d'un parent sous le nom `justify` (le renommage est, ici, essentiel), via la routine effective `justify` héritée de l'autre parent. Une forme de mariage d'intérêt, en fait.

*"N'appelez pas, nous vous appellerons", page 488.*

`LINEAR_ITERATOR` est un exemple remarquable de **classe de comportement**, capturant des comportements communs tout en laissant ouverts certains composants de façon que les descendants puissent y insérer leurs variantes spécifiques.

## Formes d'héritage de service

Les deux exemples, `ASCII` et `LINEAR_ITERATOR`, sont caractéristiques des deux principales variantes de l'héritage de service :

- l'héritage de *constante*, dans lequel le parent fournit essentiellement des attributs constants et des objets partagés ;
- l'héritage d'*opération*, dans lequel il fournit des routines.

Comme on l'a noté plus haut, il est possible de combiner ces deux variantes en un unique lien d'héritage. C'est pourquoi l'héritage de service est une catégorie simple, et non deux catégories.

## Comprendre l'héritage de service

Pour certaines personnes, l'héritage de service semble un abus du mécanisme — une forme de "bidouille". Mais ce n'est pas nécessairement le cas.

La principale question à envisager dans ces exemples ne concerne pas l'héritage mais les classes qui y ont été définies, `ASCII` et `LINEAR_ITERATOR`. Comme toujours quand nous étudions une conception de classe, nous devons nous demander : "Est-ce que cela décrit vraiment une abstraction de données sensée ?" — un ensemble d'objets caractérisés par leurs propriétés abstraites.

Dans ces exemples, la réponse est moins évidente qu'avec une classe `RECTANGLE`, `BANK_ACCOUNT` ou `LINKED_LIST`, mais elle existe néanmoins :

- la classe `ASCII` représente l'abstraction : "tout objet ayant accès aux propriétés du jeu de caractères ASCII" ;
- la classe `LINEAR_ITERATOR` représente l'abstraction : "tout objet qui peut effectuer des itérations séquentielles sur une structure linéaire". De tels objets ont tendance à être du type "machine" décrit dans le chapitre précédent.

*"Objets et machines", page 727.*

Une fois qu'on accepte ces abstractions, les liens d'héritage ne posent plus de problème : une instance de `TOKENIZER` a effectivement besoin d'"accéder aux propriétés du jeu de caractères ASCII", et une instance de `JUSTIFIER` a effectivement besoin "d'effectuer des itérations séquentielles sur une structure linéaire". En fait, nous pourrions classer de tels exemples d'héritage dans la catégorie sous-type. Ce qui différencie l'héritage de service, c'est la nature des parents.

L'observation selon laquelle ce sont bien les classes elles-mêmes qui sont au coeur de la question, et non l'utilisation de l'héritage, est renforcée par le fait qu'une application pourrait être cliente de ces classes et non en être héritière. Cela alourdirait les choses, en particulier pour `ASCII` :

```
charset: ASCII
...
!! charset
```

chaque utilisation d'un code caractère devrait être écrite comme `charset.Lower_a` ou autre. L'objet attaché à `ASCII` ne joue aucun rôle utile. Les mêmes commentaires s'appliquent à `LINEAR_ITERATOR` tant qu'une classe donnée n'a besoin que d'un genre d'itération. Si elle a besoin de plusieurs, il devient intéressant de créer des objets itérateurs ayant chacun sa propre version de `action` et `test` ; vous pouvez alors avoir autant de schémas d'itération que nécessaire.

*À propos des objets itérateurs, voir l'exercice E15.4, page 549.*

Si nous envisageons d'utiliser des objets itérateurs, il nous faut des classes d'itérateurs, et rien n'interdit que ces classes fassent partie du club de l'héritage.

# 24.10 CRITÈRES MULTIPLES ET HÉRITAGE DE VUE

Lorsqu'on utilise l'héritage, le problème peut-être le plus épineux qui puisse se manifester est l'existence de plusieurs critères alternatifs possibles pour classer les abstractions d'un certain domaine d'application.

## Classification par critères multiples

Les classifications traditionnelles des sciences naturelles utilisent un critère unique (qui peut, éventuellement, mettre en jeu plusieurs qualités) à chaque niveau : vertébré ou invertébré, feuilles renouvelées annuellement ou non, et ainsi de suite. Cela conduit à ce que nous appellerions des hiérarchies d'héritage unique, dont l'avantage essentiel est la grande simplicité. Mais cela pose également des problèmes, car la nature ne se plie pas facilement à un critère unique. Toute personne ayant essayé, lors d'une ballade dans la nature, de reconnaître les plantes à l'aide d'un livre de botanique fondé sur les critères officiels de Linné en sera convaincue. L'espèce A possède, selon le livre, des feuilles caduques, alors que l'espèce B n'en a pas ; combien de temps aurez-vous la patience d'attendre, si nous sommes en juillet, pour voir si les feuilles tombent ? On vous indique que des fleurs d'un mauve brillant apparaîtront en juin, mais comment le savoir en plein mois de janvier ? Les racines de A sont profondes d'au moins 7 mètres, et celles de B de 9 mètres — devez-vous creuser ?

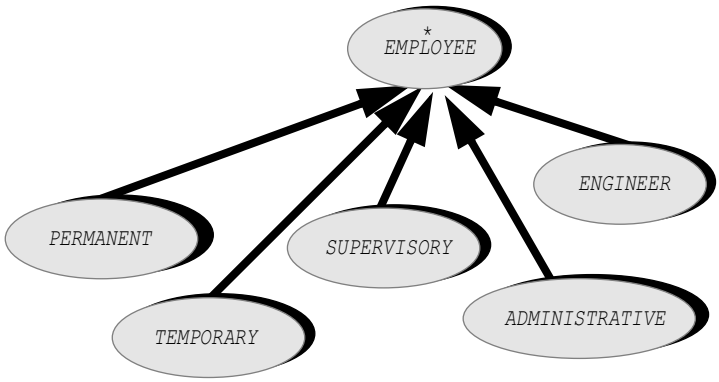
En matière de logiciel, quand un critère unique semble trop contraignant, nous pouvons utiliser les techniques d'héritage multiple, en particulier d'héritage répété, que nous avons appris à maîtriser dans les chapitres précédents. Supposez, par exemple, dans un système de gestion de personnel, que nous ayons une classe `EMPLOYEE`. Supposez, de plus, que nous ayons deux critères séparés pour classer les employés :

- par type de contrat, comme permanent ou temporaire ;
- par type de travail, comme ingénieur, administratif ou gestionnaire

et que ces deux critères conduisent, manifestement, à des classes valides de descendants ; en d'autres termes, vous ne vous engagez pas dans la taxomanie, puisque les classes que vous avez identifiées, comme `TEMPORARY_EMPLOYEE` pour le premier critère et `MANAGER` pour le second, sont effectivement caractérisées par des caractéristiques spécifiques qui ne sont pas applicables aux autres catégories. Qu'allez-vous faire ?

Une première tentative pourrait conduire à mettre toutes les variantes au même niveau :

***Une classification compliquée***



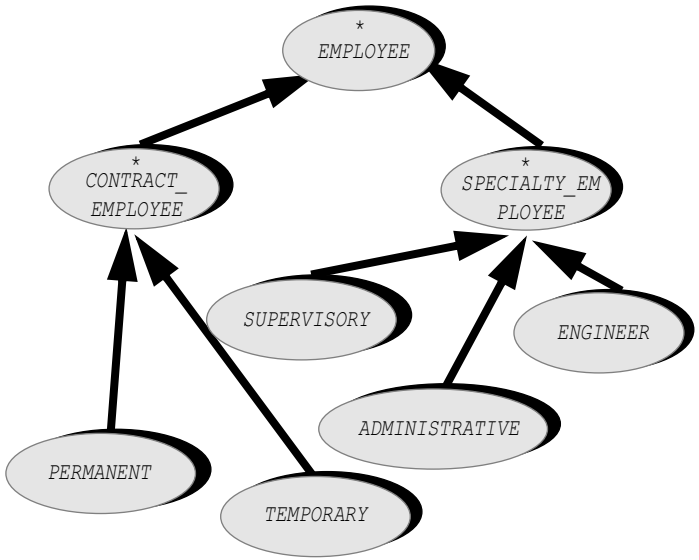

> Pour limiter la taille de cet exemple et simplifier la figure, les noms des classes ont été abrégés. Pour généraliser cet exemple à un vrai système, il nous faudrait utiliser les règles de nommage habituelles, qui conduisent à des noms plus longs et plus précis, comme `PERMANENT_EMPLOYEE`, `ENGINEERING_EMPLOYEE` et ainsi de suite.

Cette hiérarchie d'héritage n'est pas satisfaisante, car des concepts très différents sont représentés au même niveau.

## Héritage de vue

Si vous choisissez d'utiliser l'héritage pour la classification utilisée dans cet exemple, vous devriez introduire un niveau intermédiaire pour décrire les critères de classification mis en jeu :

***Classification par vues***



Remarquez que le nom *CONTRACT_EMPLOYEE* ne correspond pas à “employé ayant un contrat” (à opposer à ceux qui pourraient ne pas en avoir !) mais à “employé, caractérisé par son contrat”. Le nom de la classe cousine correspond, de même, à “employé, caractérisé par sa spécialité”.

Le fait que ces noms semblent compliqués est la manifestation d’un certain trouble, typique de ce genre d’héritage. Dans l’héritage de sous-type, nous avons rencontré la règle selon laquelle les ensembles d’instances représentées par les divers héritiers d’une classe doivent être distincts. Cette règle ne s’applique pas ici : un employé permanent peut, par exemple, être également ingénieur. Cela veut dire qu’une telle classification est conçue pour l’héritage répété : certains descendants propres des classes indiquées sur la figure auront à la fois *CONTRACT_EMPLOYEE* et *SPECIALTY_EMPLOYEE* pour ancêtres — pas directement, mais, par exemple, en héritant à la fois de *PERMANENT* et de *ENGINEER*. De telles classes seront des descendants répétés de *EMPLOYEE*.

Cette forme d’héritage peut être appelée héritage de vue : divers héritiers d’une certaine classe ne représentent pas des sous-ensembles disjoints d’instances (comme dans le cas de sous-type) mais diverses manières de classer les instances du parent. Remarquez que cela n’a de sens que si parent et héritiers sont des classes retardées, c’est-à-dire des classes décrivant davantage des catégories générales que des objets spécifiés complètement. Notre première tentative de classification par vue de *EMPLOYEE* (celle dont tous les descendants sont au même niveau) violait cette règle ; la seconde s’y plie.

## Est-ce que l’héritage de vue est approprié ?

L’héritage de vue est relativement éloigné des utilisations les plus courantes de l’héritage et est sujet aux critiques. Le lecteur jugera par lui-même de l’opportunité de l’utiliser pour ses propres besoins, mais, en tout état de cause, nous devrions examiner ses avantages et inconvénients.

Il est évident que — tout comme l’héritage répété, dont il a besoin — l’héritage de vue n’est **pas un mécanisme pour débutants**. La règle de prudence introduite pour l’héritage répété s’applique ici : si vous n’avez que quelques mois d’expérience pratique sur des projets significatifs de développement OO, mieux vaut éviter l’héritage de vue.

À la place de l’héritage de vue, vous pouvez choisir comme caractère primaire un des critères de classification, et l’utiliser comme seul guide lors de la conception de la hiérarchie d’héritage ; pour prendre en compte l’autre critère, vous utiliserez des caractéristiques spécifiques. Il est intéressant de remarquer que bon nombre de zoologistes modernes et de botanistes utilisent cette approche : leur critère de base de classification est celui de l’histoire reconstituée de l’évolution des genres et espèces mis en jeu. Dommage que nous n’ayons pas un tel standard unique et indiscutable pour nous guider lors de la conception des taxonomies logicielles.

Pour nous limiter, dans notre exemple, à un unique critère primaire, nous pourrions décider que le type de travail est le facteur le plus important et représenter le statut d’emploi par une caractéristique. Dans un premier jet, la caractéristique (de la classe *EMPLOYEE*) pourrait être :

```
is_permanent: BOOLEAN
```

mais c’est dangereusement contraignant : pour étendre le choix des possibilités, nous pourrions avoir :

```
Permanent: INTEGER is unique
Temporary: INTEGER is unique
Contractor: INTEGER is unique
...
```

mais nous avons appris à nous méfier, non sans raison, des énumérations explicites. Il vaut mieux introduire une classe *WORK_CONTRACT*, très probablement retardée, ayant autant de descendants que nécessaire pour représenter les genres spécifiques de contrats de travail. Nous pouvons alors éviter les détestables discriminations explicites de la forme :

```
if is_permanent then ... else ... end
```

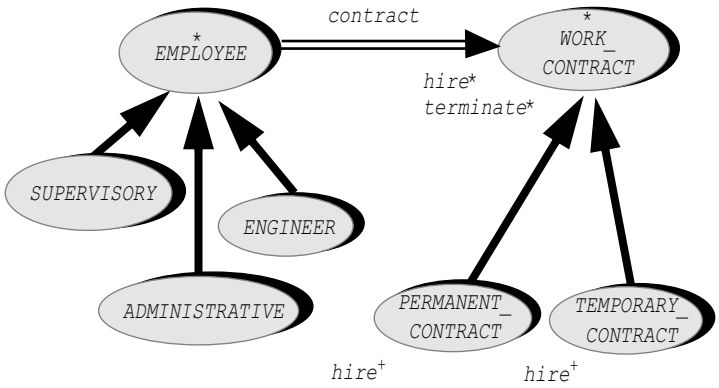
ou :

```
inspect
        contract_type
when Permanent then
        ...
when ...
        ...
end
```

et tous les problèmes d'extensibilité qu'ils présagent (découlant de la violation de la quasi-totalité des principes de modularité : continuité, choix unique, ouverture-fermeture) ; nous doterons plutôt la classe *WORK_CONTRACT* de caractéristiques retardées représentant les opérations dépendant du type de contrat, qui seront alors rendues effectives de manière différente par les descendants. La plupart de ces caractéristiques auront besoin d'un argument de type *EMPLOYEE* représentant l'employé auquel on applique l'opération ; on pourrait trouver, parmi les exemples, *hire* et *terminate*.

La structure résultante sera à peu près celle-ci :

***Classification multicritère par hiérarchies distinctes et relations clients***




*Voir "UNE APPLICATION : LA TECHNIQUE DU HANDLE", 24.3, page 789.*

Comme vous l'avez peut-être remarqué, ce schéma est presque identique à celui du schéma de conception fondée sur les **handles**, décrit plus haut dans ce chapitre.

Une telle technique peut être utilisée à la place de l'héritage de vue. Elle complique la structure en introduisant une hiérarchie séparée, un nouvel attribut (ici *contract*) et les relations clients correspondantes, mais les abstractions d'une telle hiérarchie sont inattaquables (contrat de travail, contrat permanent de travail) ; avec la solution fondée sur l'héritage de vue, les abstractions sont également claires mais un peu plus difficiles à expliquer ("employé vu selon la perspective de son contrat de travail", "employé vu selon la perspective de sa spécialité").

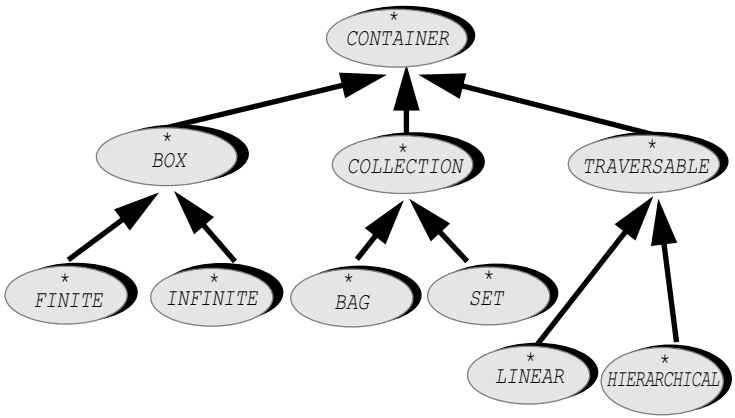
## Critères d'adoption de l'héritage de vue

La question de l'héritage de vue peut se présenter relativement tôt lors de l'analyse d'un domaine de problèmes, alors même que vous hésitez entre plusieurs critères de classification possibles. Au fur et à mesure que vous comprenez mieux le domaine d'application, un des critères se dégage fréquemment des autres, s'imposant comme guide primordial pour la conception de la structure d'héritage. Dans ce cas, l'étude précédente vous suggère fortement de renoncer à l'héritage de vue au profit de techniques plus simples.

Néanmoins, on peut considérer que l'héritage de vue est utile quand les trois conditions suivantes sont réunies :

- les divers critères de classification sont d'égale importance, et donc tout choix d'un critère primaire serait arbitraire ;
- plusieurs combinaisons possibles sont nécessaires (comme, dans l'exemple précédent, superviseur permanent, ingénieur temporaire, ingénieur permanent et ainsi de suite) ;
- les classes envisagées sont tellement importantes qu'elles justifient de passer un temps significatif à concevoir la meilleure structure possible d'héritage. Cela s'applique, en particulier, au cas des classes d'une **bibliothèque réutilisable** offrant un grand potentiel de réutilisation.

Un exemple d'application de ces critères est la structure la plus externe des bibliothèques Base dans l'environnement décrit dans le dernier chapitre de ce livre. Les classes résultantes ont été obtenues en s'efforçant d'appliquer, comme le décrit en détail le livre [M 1994a], les principes taxonomiques à la classification systématique des structures de base de l'informatique, suivant en cela la tradition propre aux sciences naturelles. La partie supérieure de la structure "container" ressemble à ceci :




***Une classification par vues des structures fondamentales de l'informatique***

La classification de premier niveau (`BOX`, `COLLECTION`, `TRAVERSABLE`) est fondée sur les vues : le niveau en dessous (tout comme nombre de ceux plus bas dans la hiérarchie et non représentés ici) est une classification de sous-type. Une structure conteneur est caractérisée par trois critères :

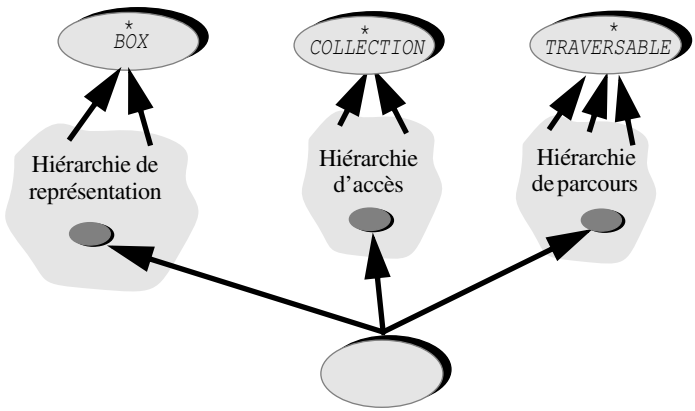
- la manière dont on accédera aux éléments : `COLLECTION`. Un ensemble `SET` permet de déterminer si un élément est présent, alors que `BAG` permet également au client de déterminer le nombre d'occurrences d'un élément donné. Parmi les raffinements qui suivent, on trouve d'autres

abstractions d'accès comme *SEQUENCE* (les éléments sont accessibles de manière séquentielle), *STACK* (éléments accessibles dans l'ordre inverse de celui de leur insertion) et ainsi de suite ;

- la manière dont seront représentés les éléments : *BOX*. Parmi les variantes, on trouve les structures finies et infinies. Une structure finie peut être bornée ou non ; une structure bornée peut être de taille fixe ou modifiable ;
- la manière dont seront traversées les structures : *TRAVERSABLE*.

Il est intéressant de remarquer que la hiérarchie n'a pas été, au départ, envisagée sous l'angle de l'héritage de vue. L'idée initiale était de définir *BOX*, *COLLECTION* et *TRAVERSABLE* comme des classes sans lien, se trouvant chacune au sommet d'une hiérarchie séparée ; puis, quand il s'agirait de décrire une implémentation particulière de structures de données, d'utiliser l'héritage multiple en choisissant un parent dans chacune des trois parties. Par exemple, une liste chaînée est finie et non bornée (représentation), permet l'accès séquentiel (accès) et peut être traversée de manière linéaire (parcours) :

***Construction d'une classe de structures de données par combinaison d'abstractions par héritage multiple***




Mais nous avons alors réalisé qu'il n'était pas convenable de maintenir séparées *BOX*, *COLLECTION* et *TRAVERSABLE* : elles avaient toutes besoin de quelques caractéristiques communes, en particulier *has* (test d'appartenance) et *empty* (test d'absence d'éléments). La nécessité d'introduire un ancêtre commun se faisait clairement sentir — *CONTAINER*, où apparaissent dorénavant ces caractéristiques communes. Ainsi, une structure qui était, au départ, conçue pour un héritage multiple pur, avec trois hiérarchies disjointes au sommet, s'est transformée en une hiérarchie d'héritage avec vue contenant une quantité considérable d'héritage répété.

Bien que difficile à concevoir correctement au début, cette structure s'est avérée utile, souple et stable, confirmant les deux conclusions de cette étude : l'héritage de vue n'est pas fait pour les peureux ; et il peut jouer, quand on peut l'appliquer, un rôle clé dans les domaines de problèmes complexes dans lesquels interagissent de nombreux critères — si l'effort est justifié, comme c'est le cas avec une bibliothèque fondamentale de composants réutilisables, qui se doit, tout simplement, d'être faite correctement.

# 24.11 COMMENT DÉVELOPPER DES STRUCTURES D'HÉRITAGE

*Une partie du contenu de cette section est tirée de* [M 1995].

Quand vous lisez un livre ou un article pédagogique sur la méthode orientée objet, ou quand vous découvrez une bibliothèque de classes, les hiérarchies d'héritage que vous voyez ont toujours été conçues précédemment, et l'auteur ne vous dit pas comment il a fait pour y parvenir. Comment allez-vous donc vous y prendre pour concevoir vos propres structures ?

## Spécialisation et abstraction

Que ce soit voulu ou non, de nombreuses présentations pédagogiques laissent supposer que les structures d'héritage devraient être conçues en allant du plus général (la partie supérieure) au plus spécifique (les feuilles). Cela vient, en partie, du fait que c'est souvent la meilleure manière de *décrire* une bonne structure, une fois qu'elle existe : du général au particulier ; en allant des figures aux figures fermées, aux polygones, aux rectangles, aux carrés. Mais la meilleure manière de décrire une structure n'est pas nécessairement la meilleure manière de la *produire*.

> Un commentaire similaire, dû à Michael Jackson, a été évoqué lors de l'étude de la conception descendante.

*Voir "Production et description", page 117.*

Dans un monde idéal peuplé de gens parfaits, nous pourrions toujours reconnaître immédiatement les bonnes abstractions et en tirer les catégories, leurs sous-catégories et ainsi de suite. Dans le monde réel, cependant, nous voyons souvent un cas spécifique avant de découvrir l'abstraction générale dont il n'est qu'une variante.

Dans de nombreux cas, l'abstraction n'est pas unique ; la meilleure manière de généraliser une certaine notion dépend très probablement de ce que vous ou vos clients voudrez faire de cette notion et de ses variantes. Prenons l'exemple d'une notion que nous avons souvent rencontrée lors des discussions précédentes : celle des points dans un espace à deux dimensions. On peut envisager au moins quatre généralisations possibles :

- des points dans un espace de dimension arbitraire — ce qui conduit à une structure d'héritage où les soeurs de la classe `POINT` seront les classes `POINT_3D` et ainsi de suite ;
- des figures géométriques — les autres classes de la structure étant du type `FIGURE`, `RECTANGLE`, `CIRCLE` et ainsi de suite ;
- des polygones — avec d'autres classes comme `QUADRANGLE` (quatre nœuds), `TRIANGLE` (trois nœuds) et `SEGMENT` (deux nœuds), `POINT` étant le polygone spécial n'ayant qu'un nœud ;
- des objets qui sont complètement définis par deux coordonnées — les autres prétendants étant ici `COMPLEX` et `VECTOR_2D`.

Bien que certaines de ces généralisations soient plus attrayantes que d'autres, il est impossible de dire dans l'absolu laquelle d'entre elles est la meilleure. La réponse dépendra de la manière dont évolue votre base logicielle et de ce dont elle aura besoin. Ainsi, un processus prudent dans lequel vous abstrayez un petit peu trop tard, parce que vous avez attendu d'être sûr d'avoir trouvé le chemin de généralisation le plus utile, peut se révéler préférable à celui où vous auriez introduit trop d'abstraction trop tôt, sans la valider.

## Le caractère arbitraire des classifications

L'exemple `POINT` est typique. En présence de deux classifications concurrentes pour un ensemble donné d'abstractions, vous pourrez souvent déterminer, sur la base d'arguments rationnels, laquelle est *meilleure* ; mais on est rarement capable de déterminer qu'une structure d'héritage donnée est *la meilleure* possible.

Cette situation n'est pas propre au logiciel. Ne croyez pas, par exemple, que les classifications de Linné des sciences naturelles soient universellement acceptées et éternelles. Les responsables de l'archive Internet "Tree of Life" évoquée précédemment indiquent, dès le début, que la classification du projet — toute interdisciplinaire qu'elle soit — est controversée. Et il ne s'agit pas seulement du cas de minuscules créatures trop visqueuses pour être évoquées au moment du déjeuner : la classification Web du docteur Everham pour les oiseaux, citée précédemment, est accompagnée du commentaire :

*Voir page 813. Les références bibliographiques ont été omises.*

*Il y a 174 familles, 2 044 genres et 9 021 espèces d'oiseaux dans le monde ! Les espèces les plus abondantes se trouvent dans l'ordre des passereaux avec 5 276 espèces. Le plus petit nombre d'espèces dans un ordre est 1 : l'autruche dans les struthioniformes. (Je pensais que l'autruche serait dans un ordre commun aux émeus, kiwis et moas, qui sont tous éteints, car ils sont tous incapables de voler, ont des jambes solides et de longs cous.) Le système de Linné regroupe les organismes selon leur similarité morphologique. Une autre classification des animaux est fondée sur l'hybridation ADN-ADN. Elle est très complexe ; par exemple, un coucou américain serait classé sous : règne, animalia ; phylum, chordata ; classe, aves ; sous-classe, neornithes ; infra-classe, neoaves ; parv-classe, passerae ; ordre supérieur, cuculimorphae ; ordre, cuculiformes ; infra-ordre, cuculides ; parv-ordre, coccyzida ; famille, coccyzidae.*

*On trouvera plus d'informations sur les méthodes concurrentes de classification à la fin de ce chapitre.*

Ceci illustre la compétition entre les deux systèmes : le modèle traditionnel, fondé sur la morphologie (et l'évolution) ; et un système plus inductif fondé sur l'analyse de l'ADN. Ils conduisent à des résultats radicalement différents. Remarquez également, par ailleurs, que le zoologiste cité considère que le fait de ne pas voler devrait être un critère taxonomique significatif — en désaccord avec la classification officielle.

## Induction et déduction

Un bon processus de conception de hiérarchies logicielles combine aspects déductif et inductif, aspects de spécialisation et de généralisation : vous découvrirez parfois d'abord l'abstraction pour, ensuite, en inférer les cas spéciaux ; parfois, vous commencerez par construire ou trouver une classe utile pour, ensuite, réaliser qu'il existe un concept sous-jacent plus abstrait.

S'il s'avère que vous n'utilisez pas toujours le premier schéma, mais si, de temps en temps, vous ne découvrez l'abstrait qu'après avoir vu le concret, *cela peut être tout à fait normal*. Vous utilisez simplement une approche "yoyo" de la classification.

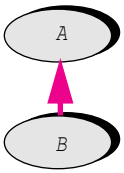
Au fur et à mesure que vous accumulerez expérience et compréhension, vous verrez croître la part des décisions correctes prises a priori. Mais une composante a posteriori subsistera toujours.

## Variétés d'abstractions de classe

*Ce principe de retour en arrière est l'un des plus admirables attributs de l'héritage.*
Charles Darwin

Il existe deux formes fréquentes et utiles de construction a posteriori de parents.

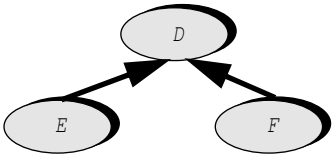
*Abstraire* découle de la reconnaissance tardive d'une abstraction de plus haut niveau. Vous découvrez une classe `B` qui correspond à une notion utile, mais dont le développeur n'avait pas remarqué qu'il s'agissait d'un cas spécial d'une notion plus générale `A`, justifiant un lien d'héritage :




***Abstraction***

Que cette observation ait été ignorée au début — c'est-à-dire que `B` avait été construite sans `A` — n'est pas une raison pour renoncer à l'utilisation de l'héritage. Une fois que la nécessité de `A` a été reconnue, vous pouvez et, dans la plupart des cas, devez écrire cette classe et adapter `B` de manière à ce qu'elle devienne une de ses héritières. Ce n'est pas aussi bien que d'avoir écrit `A` en premier, mais c'est mieux que ne rien écrire du tout.

*Factoriser* est le cas où vous détectez que deux classes `E` et `F` représentent, en fait, des variantes d'une même notion générale :



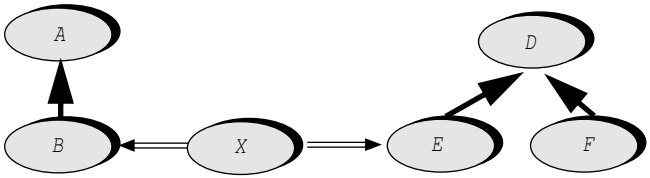

***Factorisation***

Si vous reconnaissez ce facteur commun par la suite, l'étape de généralisation vous permettra d'ajouter une classe parent commune `D`. Il eût été préférable, là encore, d'obtenir la bonne hiérarchie du premier jet, mais mieux vaut tard que jamais.

## Indépendance de client

Abstraire et factoriser peuvent, dans de nombreux cas, s'effectuer sans effets négatifs sur les clients existants (une application du principe ouvert-fermé).

Cette propriété découle de l'utilisation de la rétention d'information par la méthode. Considérez à nouveau les cas schématiques précédents, mais en ajoutant au tableau une classe client typique `X` :




***Abstraction, factorisation et clients***

Quand `B` est abstrait en `A` ou quand les caractéristiques de `E` sont factorisées avec celles de `F` en `D`, une classe `X` client de `B` ou `E` (dans la figure, il s'agit d'un client des deux) ne ressentira, dans la majorité des cas, aucun effet dû à ce changement. Les ancêtres d'une classe n'affectent pas ses

clients si ceux-ci ne font qu'appliquer des caractéristiques de la classe à des entités du type correspondant. En d'autres termes, si `X` utilise `B` et `E` comme fournisseur selon le schéma :

```
b1: B; e1: E
...
b1.some_feature_of_B
...
e1.some_feature_of_E
```

`X` n'est en rien affecté par un changement de parent de `B` ou `E` provenant d'un processus d'abstraction ou de factorisation.

## Élever le niveau d'abstraction

Abstraire et factoriser sont représentatifs du processus continuel d'amélioration qui caractérise un processus de construction logicielle orientée objet réussi. Pour moi, il s'agit d'un des aspects les plus exaltants qu'apporte la pratique de la méthode : savoir que, même si on ne s'attend pas à ce que vous atteigniez la perfection du premier coup, on vous donne la chance d'améliorer sans cesse votre conception jusqu'à ce que tout le monde soit satisfait.

Dans un groupe de développement qui applique bien la méthode, cette élévation régulière du *niveau d'abstraction* du logiciel et, en conséquence, de sa qualité est clairement perceptible aux membres du projet, et sert d'incitation et de motivation constantes.

# 24.12 UNE VISION D'ENSEMBLE : BIEN UTILISER L'HÉRITAGE

L'héritage ne cessera jamais de nous surprendre par sa puissance et son caractère versatile. Nous avons essayé, dans ce chapitre, de mieux appréhender ce qu'est exactement l'héritage et la manière de l'utiliser à notre avantage. Quelques conclusions essentielles ont surgi.

Tout d'abord, nous ne devrions pas avoir peur de la variété des modes d'utilisation de l'héritage. Interdire l'héritage multiple ou l'héritage de service n'amène que des ennuis. Ces mécanismes sont là pour vous aider : il faut les utiliser, mais à bon escient.

Ensuite, l'héritage est, pour l'essentiel, une technique de *fournisseur*. C'est une arme de notre arsenal de techniques permettant de combattre nos adversaires (en particulier, la complexité, farouche ennemi du développeur logiciel). L'héritage peut être un facteur important dans le logiciel *client*, en particulier dans le cas des bibliothèques, mais son principal objectif est, avant tout, de nous aider à construire.

Bien évidemment, tout logiciel est conçu pour ses clients, et les besoins des clients orientent le processus. Un ensemble de classes est bon s'il offre un excellent service au logiciel client : des interfaces et des implémentations correspondantes complètes, exemptes de mauvaises surprises (comme des surcoûts de performance inattendus), simples à utiliser, faciles à se rappeler, extensibles. Pour atteindre ces objectifs, le concepteur est libre d'utiliser comme il le souhaite l'héritage et les autres techniques orientées objet. La fin justifie les moyens.

Rappelez-vous également, quand vous concevez une structure d'héritage, que le but est de construire du logiciel, pas de faire de la philosophie. Il y a rarement de solution unique, ni même de solution idéale dans l'absolu. La "meilleure" solution n'a un sens que par rapport à une

certaine classe d'applications clientes. C'est particulièrement vrai quand on s'éloigne des mathématiques et de l'informatique théorique, pour lesquels existe un ensemble communément accepté de résultats théoriques, et que l'on aborde les domaines d'applications proches de l'entreprise. Pour déterminer la hiérarchie de classe la mieux adaptée à la notion d'action d'une entreprise, il faut savoir au préalable par qui va être utilisé le logiciel : des investisseurs individuels, une entreprise cotée en bourse, un courtier ou une société de bourse.

D'une certaine manière, ce constat est réconfortant. Le naturaliste qui classe un ensemble donné de plantes et d'animaux doit concevoir des catégories absolues. Dans le logiciel, ce cas de figure ne se produit que si vous avez à produire des bibliothèques d'intérêt général (comme celles concernant des structures fondamentales de données, des bases de données). La plupart du temps, vos buts seront plus modestes. Il vous faudra concevoir une *bonne* hiérarchie qui réponde aux besoins d'un certain genre de logiciel client.

La dernière leçon de ce chapitre généralise un commentaire élaboré lors de l'étude de l'héritage de service : la principale difficulté que pose la construction des structures de classe n'est pas l'héritage en soi ; c'est la recherche des abstractions. Si vous avez identifié des abstractions valides, leur structure d'héritage apparaîtra d'elle-même. Pour découvrir les abstractions, le guide que vous utiliserez est celui que nous avons suivi tout au long de ce livre : la théorie des types abstraits de données.

## 24.13 CONCEPTS CLÉS INTRODUITS DANS CE CHAPITRE

- Chaque utilisation de l'héritage devrait correspondre à une certaine forme de relation "est" entre deux catégories d'objets, dans un domaine modélisé externe ou dans le logiciel même.
- N'utilisez pas l'héritage pour modéliser une relation "contient ce genre de composant" ; cela reste du domaine de la relation client. (Rappelez-vous `CAR_OWNER`.)
- Quand on peut appliquer l'héritage, la relation client est souvent applicable également. Si la vue correspondante peut changer, utilisez la relation client ; si vous envisagez des utilisations polymorphes, utilisez l'héritage.
- N'introduisez des noeuds d'héritage intermédiaires que s'ils décrivent une abstraction bien identifiée et dotée de caractéristiques spécifiques.
- On a défini une classification de l'héritage, fondée sur douze genres divisés en trois catégories générales : l'héritage de modèle (décrivant des relations qui existent dans le domaine modélisé), l'héritage logiciel (décrivant des relations présentes dans le logiciel même) et l'héritage de variation (pour l'adaptation de classes, que ce soit dans le modèle ou le logiciel).
- La puissance de l'héritage provient de la combinaison d'une spécialisation de type et d'un mécanisme d'extension de module. Utiliser des mécanismes de langage différents ne semble ni sage ni utile.
- Les héritages d'implémentation et de service doivent être abordés avec précaution, mais peuvent se révéler de puissantes techniques pour le fournisseur.
- L'héritage de vue, technique délicate mettant en jeu l'héritage répété, permet de classer les objets selon divers critères concurrents. Il est utile dans les bibliothèques professionnelles. Dans de nombreux cas, on préférera la technique plus simple fondée sur les handles.

- Bien que cela ne soit pas, en théorie, l'idéal, le processus de conception des hiérarchies d'héritage est, en pratique, souvent de type yoyo — de l'abstrait au concret et vice versa.
- L'héritage est, avant tout, une technique de fournisseur.

## 24.14 NOTES BIBLIOGRAPHIQUES

La principale référence concernant la taxonomie de l'héritage est [Girod 1991]. Un livre consacré à la méthodologie OO [Page-Jones 1995], l'un des rares à fournir des conseils utilisables de méthodologie de conception orientée objet, contient des conseils précieux sur les bonnes et mauvaises utilisations de l'héritage. Une autre référence utile est [McGregor 1992] ; John McGregor a, en particulier, exploré la technique introduite, dans ce chapitre, sous le nom d'héritage de vue.

[Breu 1995] propose également d'intéressants concepts fondés sur une vision de l'utilisation de l'héritage plus restrictive que celle de ce chapitre.

Une technique semblable à celle des "handles" est décrite dans [Gil 1994].

La préparation de ce chapitre a bénéficié des commentaires de plusieurs biologistes gérant les ressources accessibles sur le Web concernant la taxonomie des êtres vivants, en particulier : le "tree of life" de l'Université d'Arizona (*phylogeny.arizona.edu/tree/life.html*), avec la permission des professeurs David Maddison et, pour les oiseaux, Michel Laurin (ce dernier de Berkeley). Le professeur Edwin Everham de l'Université de Radford a également été très utile.

Des références générales sur la théorie de la classification, ou *systématique*, se trouvent à la fin de la prochaine section.

## 24.15 UNE BRÈVE HISTOIRE DE LA TAXONOMIE

Voici quelques informations supplémentaires, qui ne sont pas utilisées dans le reste de ce livre. L'étude des préoccupations taxonomiques des autres disciplines est pour nous, développeurs logiciels, riche de leçons potentielles. J'espère aiguillonner l'intérêt pour ce sujet fascinant — un sujet interdisciplinaire qui peut faire l'objet d'une thèse ou d'un mémoire de maîtrise.

### D'Aristote à Darwin

On fait remonter l'étude de la classification des espèces à Aristote (384-322 av. J.-C.), dont la taxonomie des animaux, Historia Animalium, enrichie, sous le titre Historia Plantarum, par celle des plantes grâce à son étudiant Theophrastus d'Eresos (370-288 av. J.-C.), a été considérée pendant des siècles comme le nec plus ultra sur le sujet. Parmi les critères utilisés par Aristote pour classer les animaux, on trouve la manière dont ceux-ci se reproduisent et vivent ; en adoptant un point de vue plus moderne, seul le premier critère serait considéré comme pertinent puisque nous en sommes venus à accepter qu'un dauphin soit, indépendamment de considérations d'habitat, plus proche d'un lama que d'un requin. La classification de Theophrastus était plus systématiquement structurelle. La terminologie botanique moderne découle largement des termes utilisés par Aristote et de Theophrastus via la traduction latine qu'en a donnée Pline l'Ancien

(23-79 apr. J.-C.) dans son Histoire naturelle (Pline était pleinement conscient de la nécessité d'éviter d'être trompé par les apparences : "Le projet [de certains naturalistes grecs] consistait, pour une part, à dessiner en couleur les diverses plantes, puis à ajouter par écrit une description de leurs propriétés. Cependant, les images sont souvent trompeuses ; … par ailleurs, il n'est pas suffisant de représenter une seule image d'une plante, prise à un instant donné, car celle-ci prend une forme différente dans chacune des quatre saisons de l'année.") Dioscorides d'Anazarbus (1er siècle apr. J.-C.), docteur de Néron, a, par la suite, apporté une contribution significative à ce projet en classant les plantes selon leurs propriétés médicinales.

Plusieurs érudits ont repris ce travail au moment de la Renaissance, en particulier Gessner, qui devait influencer Linné et Cuvier par ses *Opera Botanica* et *Historia Plantarum* (1541-1571), où il distingue genre, espèce et classe, et Caspar Bauhin, qui conçut un système binomial pour la classification des plantes dans son *Pinax* (1596). Au siècle suivant, John Ray (1628-1705) a éliminé certains aspects arbitraires des classifications existantes en prenant simultanément en compte plusieurs propriétés de la morphologie des plantes. Il a établi la division de base des plantes (division dont Theophrastus avait eu l'intuition) qui fleurissent en monocotylédones et dicotylédones. Cette division, qui reste d'actualité, est un autre exemple du flou qui existe encore dans les critères fondamentaux de classification de la biologie ; le musée de paléontologie de l'Université de Berkeley (voir les références bibliographiques à la fin de cette section) donne une liste de sept facteurs permettant de distinguer monocotylédones et dicotylédones — un ou deux cotylédons dans l'embryon, partie fleurie en multiple de trois ou de quatre/cinq, etc. — mais ajoute qu'aucun facteur unique de cette liste n'identifie sans ambiguïté si une plante à fleurs donnée est une monocotylédone ou une dicotylédone.

Ce n'est qu'au XVIIIe siècle, avec le développement d'une science de la biologie et la croissance rapide du nombre d'espèces connues, que le problème de la classification biologique commence à prendre un caractère d'urgence. Alors que Theophrastus n'avait identifié que cinq cents espèces de plantes, Bauhin en connaissait six mille et Linné en avait catalogué dix-huit mille ; moins d'un siècle plus tard, la liste de Cuvier ne comportait pas moins de cinquante mille entrées ! Les scientifiques-philosophes des Lumières, motivés par la classification des corps célestes qu'avait élaborée Newton dans ses *Principia Mathematica* (1687), ne se contentaient désormais plus de répertorier les espèces, mais commençaient à rechercher des principes pertinents pour les grouper en catégories — via des mécanismes d'abstraction adéquats, comme diraient les informaticiens que nous sommes. Les prémisses de la taxonomie moderne remontent à cet effort collectif du début de l'ère moderne.

La contribution majeure fut celle du botaniste suédois Carl Linné (1701-1778), connu également sous le nom latin de Carolus Linnaeus, qui publia en 1737 son système taxonomique, qui reste à la base de tous les systèmes taxonomiques

utilisés aujourd'hui. Une des innovations principales fut — pour utiliser à nouveau la terminologie du génie logiciel — d'éliminer l'approche *descendante* utilisée par les taxonomistes précédents (qui définissaient des catégories abstraites de base pour les diviser successivement en groupes plus petits) pour adopter une approche *ascendante*, en phase avec l'accent mis sur le pragmatisme et l'expérimentation qui a caractérisé les débuts de la méthode scientifique ; il a commencé par les espèces elles-mêmes et les a groupées en catégories.

Ray et Linné cherchaient, tous deux, un "système naturel", c'est-à-dire une classification idéale qui mettrait en lumière des intentions divines.

Le progrès qui sépare Linné de Darwin était dû, en grande partie, à une époustouflante succession de naturalistes au Jardin des Plantes de Paris :

- Georges-Louis de Buffon (1707-1788) a écrit une magnifique *Histoire naturelle* en quarante-quatre volumes, suffisamment innovante pour envisager un ancêtre commun aux hommes et aux singes ;
- Antoine-Laurent de Jussieu (1748-1836) a cherché un système de classification des plantes plus naturel et plus global que celui de Linné. Les taxonomies modernes découlent, en fait, des travaux de Jussieu, fondés sur ceux de Ray (bien que les systèmes modernes de classification s'appuient sur les idées de Linné, sa classification même a été mise de côté — en partie, initialement, pour des raisons morales, car il insistait beaucoup sur les caractéristiques sexuelles) ;
- Jean-Baptiste Lamarck (1744-1829), dont la théorie de l'évolution préfigurait celle de Darwin, a publié sa *Flore française* en 1778 et a introduit, presque seul, la classification des "invertébrés", terme qu'il a inventé. Dans son *Histoire naturelle des animaux sans vertèbres*, il fut le premier à séparer les crustacés des insectes ;
- Georges Cuvier (1769-1832) fit pour les vertébrés ce qu'accomplit Lamarck pour les invertébrés. Il était célèbre pour sa capacité à reconstruire des organismes complets sur la seule base de fragments fossiles. Il a classé les animaux en quatre branches ;
- Étienne Geoffroy Saint-Hilaire (1772-1844), autre grand taxonomiste, a été l'adversaire de Cuvier (qu'il avait amené à Paris) lors d'un débat public célèbre sur la question de l'unité et de la diversité des formes de vie. L'argument reflétait des questions plus profondes : visions évolutionniste ou fixe des espèces, et la question, toujours d'actualité, du formalisme et du fonctionnalisme. Quand nous voyons Cuvier écrire, en 1828, "*S'il y a des ressemblances entre les organes des poissons et ceux d'autres classes de vertébrés, cela ne vient que des ressemblances de leurs fonctions*" et Geoffroy répondre, en 1829, "*Les animaux n'ont d'autres usages que ceux qui découlent de la structure de leurs organes*", il est difficile à un professionnel du logiciel d'éviter de penser "type abstrait de données" et "implémentation".

La révolution suivante de la pensée taxonomique a été déclenchée par Charles Darwin (1809-1882), dont *De l'origine des espèces* (1859) a introduit un principe simple de taxonomie : utiliser l'histoire évolutionniste. La classification des organismes selon leur position dans l'évolution est appelée **cladistique**. Pour certains biologistes, c'est le *seul* critère valable. Citons, à nouveau, le musée de paléontologie de Berkeley :

*Durant de nombreuses années, même avant Darwin, il était fréquent de raconter des "histoires" sur la manière dont étaient apparues certaines caractéristiques des organismes. Avec la cladistique, il est possible de déterminer si ces histoires sont fondées et, sinon, de les abandonner au profit d'autres hypothèses. Par exemple, on a longtemps dit que les araignées qui tissent leurs toiles sous forme de sphères complexes et organisées avaient évolué à partir d'araignées qui tissaient des toiles plus simples. L'analyse cladistique de ces araignées a, en fait, montré que le tissage des sphères était l'état primitif et que le tissage de toiles était le résultat d'une évolution débutant avec des araignées utilisant des toiles plus organisées.*

Les biologistes qui utilisent ce critère unique et incontestable ont, d'une certaine manière, plus de chance que nous autres, pauvres modélisateurs logiciels : ils peuvent supposer, ou prétendre, qu'il existe une unique vérité taxonomique et que le seul problème est de la reconstruire. (En d'autres termes, ils ont achevé la quête du système naturel que menaient Ray, Linné et Jussieu.) Dans la modélisation logicielle, nous ne pouvons postuler, et encore moins découvrir, l'existence d'une telle vérité sous-jacente.

## Le théâtre moderne

Vous pourriez croire que la taxonomie biologique serait, de nos jours, de par sa longue et prestigieuse histoire s'étirant d'Aristote à Darwin et Huxley, un domaine stable. Il n'en est rien. Depuis les années soixante, la controverse fait rage. Il y a trois grandes écoles, dont l'ardeur des débats serait familière à toute personne ayant entendu des ingénieurs logiciels discuter de leurs langages de programmation favoris. Voici — après la taxonomie des taxonomies du début de ce chapitre — la taxonomie des taxonomistes :

- les *phénotypiciens numériques* dérivent leurs classifications de l'étude des caractères individuels des organismes, en utilisant des mesures numériques de distance (et en s'appuyant largement sur des algorithmes informatiques) pour grouper les organismes ayant le plus de caractéristiques en commun. Sokal et Sneath sont les fondateurs reconnus de cette approche ;
- les *cladistes* utilisent comme seul critère l'histoire évolutionniste. L'extrait de Berkeley illustre cette approche (voir ci-dessous pour plus de détails). La cladistique tire son inspiration des travaux d'un scientifique allemand, Willi Hennig, publiés tout d'abord en allemand en 1950, puis en anglais en 1965 ;
- les *taxonomistes évolutionnistes*, conduits par G.G. Simpson et Ernst Mayr, et se réclamant de l'héritage direct de Darwin, "*fondent* [leur] *classifications sur les similarités et différences observées parmi les groupes d'organismes, évaluées à la lumière de leur histoire évolutionniste inférée*" (comme l'indique Mayr, 1981, dont on trouvera la référence ci-dessous).

Il est presque impossible de trouver dans la littérature une présentation neutre des arguments de chaque approche. (Cela vous rappelle peut-être quelque chose.) Il ne reste au profane qu'à essayer de se former une vue impartiale. Dans ce bref survol, nous essaierons de rester aussi proches que possible des analogies logicielles.

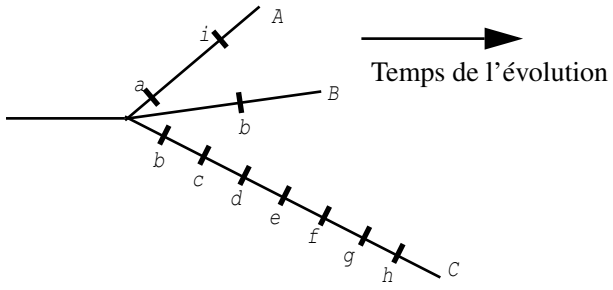
La phénotypie numérique — ce que nous appellerions l'approche ascendante — présente l'avantage d'être fondée sur des mesures précises et reproductibles. Mais le choix des caractéristiques mesurées et leurs poids sont subjectifs. Une mesure purement externe risque d'être influencée par des facteurs de hasard ; il est bien connu, depuis Darwin, que l'évolution ne met pas en jeu que des divergences (espèces évoluant d'un ancêtre commun en développant des caractères différents), mais aussi des convergences (espèces complètement différentes développant des caractéristiques similaires pour s'adapter à des environnement similaires ou par simple hasard). Il y a donc un grand risque d'arbitraire. On peut également craindre l'instabilité : la découverte de nouvelles espèces — qui se produit constamment en biologie — pourrait, plus fréquemment que dans les autres approches, remettre en question les classifications tirées d'une analyse statistique des espèces déjà connues.

Les deux autres écoles pourraient, superficiellement, sembler très proches l'une de l'autre. Pourquoi continuent-elles, alors, à débattre entre elles du haut de leurs journaux et conférences respectifs ? La raison en est que les cladistes sont, selon eux, plus rigoureux ou, selon les deux autres écoles, dogmatiques. Ils ne reconnaissent comme seul critère de classification que l'évolution, et elle seule. La méthode est particulièrement stricte : elle examine l'histoire évolutionniste, indiquée par un fossile, pour distinguer les caractères *synapomorphes* et *plésiomorphes*. Une caractéristique est plésiomorphe si elle est déjà présente dans un ancêtre commun ; pour le cladiste, elle n'a aucun intérêt ! Les seules caractéristiques utiles sont celles qui sont synamorphes, c'est-à-dire présentes dans deux organismes sans l'être dans leur ancêtre. Les synapomorphies constituent le principal outil de positionnement des nouveaux groupes (*taxa*, le pluriel de *taxon*).

Dans la situation suivante, les cladistes ne verront donc que deux taxa :

***Un cladogramme***

D'après Mayr, 1961.




C'est un *cladogramme*, ou enregistrement de l'apparition des caractères dans l'histoire évolutionniste. Les marques indiquent de nouveaux caractères. `B` et `C` présentent une synapomorphie, le caractère **b**, qui n'était pas dans l'ancêtre et qui ne se trouve pas dans `A` ; ainsi, pour un cladiste, `B` et `C` formeront un taxon, et `A` un autre. Pour un taxonomiste évolutionniste, il y aura trois taxa, puisque `C` diffère de `B` par de nombreux autres caractères (**c** à **h**). Dans sa forme pure, la cladistique est

encore plus restrictive : comme la phonologie de Roman Jakobson, elle ne prend en considération que des caractères *binaires* ; et elle insiste, quand des taxa évoluent d'un ancêtre commun, pour que l'ancêtre disparaisse.

La taxonomie évolutionniste semble une approche plus modérée, essayant de s'inspirer à la fois de la cladistique et de la phénotypique : l'évolution est la classification primordiale, mais elle est complétée par l'analyse d'autres caractéristiques, qui ne sont pas forcément synapomorphes.

D'où vient le caractère si strict de la cladistique ? L'argument essentiel est épistémologique : essayer de se plier aux règles de réfutabilité de Karl Popper. Les cladistes affirment que leur approche est la seule à ne pas être circulaire ; alors que les deux autres supposent, plus ou moins (selon cette vue), ce qu'ils essaient de déduire, une hypothèse cladistique peut être réfutée, de même qu'une unique expérience peut anéantir une théorie physique, alors qu'on ne peut *prouver* une théorie par la seule expérimentation.

Le débat entre ces approches n'est pas clos. Le progrès de la biologie moléculaire l'influencera sans aucun doute ; en particulier, en fournissant un lien entre les caractères observés et l'enregistrement évolutionniste, il peut faciliter ne serait-ce qu'une réconciliation partielle entre la phénotypique et les deux autres méthodes.

Nous allons, avec regret, nous arrêter ici (des considérations plus terre à terre de génie logiciel nous appellent). Pour un développeur logiciel OO, lire la littérature de la taxonomie, bien que demandant parfois une attention pour le moins soutenue ("*Une définition phylogénétique de l'homologie peut être considérée comme plus réfutable qu'une définition phénotypique et est, donc, préférable si elle conduit à une hypothèse d'homologie contenant, à la fois, toutes les réfutations potentielles fournies par les comparaisons phénotypiques et les réfutations potentielles fournies par la phylogénie*…"), est riche d'enseignements. Notre propre travail nous soumet constamment, comme nos collègues du département de biologie ou de l'herbarium, aux chants des sirènes des différents bords : la forme de classification a priori, descendante, déductive et fondée sur l'ordre "naturel" des choses, nous venant, via les cladistes, de Linné ; et la vision empirique, inductive, ascendante des phénotypicistes, nous disant d'observer et de collecter. Peut-être voudrons-nous, comme les taxonomistes évolutionnistes, prendre un peu des deux côtés.

## Bibliographie sur la taxonomie

Les références suivantes — qui ont été séparées de la bibliographie principale de ce livre pour éviter un trop gros mélange des genres — sera utile comme point de départ sur le sujet de l'histoire de la taxonomie :

- les informations en ligne du musée de l'université de Californie à Berkeley sur l'évolution : http://www.ucmp.berkeley.edu/clad/clad4.html (auteurs : Allen G. Collins, Robert Guralnick, Brian R. Speer). Résolument cladiste. Une partie de la présentation ci-dessus est tirée des pages de l'UCMP et des suggestions de leurs auteurs ;

- une biographie de Jussieu : *Antoine-Laurent de Jussieu*, *Nature and the Natural System* par Peter F. Stevens, Columbia University Press, New York, 1994 (je remercie le professeur Stevens pour plusieurs suggestions importantes) ;
- un ensemble d'articles sur la cladistique : *Cladistic Theory and Methodology*, édité par Thomas Duncan et Tod F. Stuessy, Van Nostrand Reinhold, 1985. Très cladiste, mais la fin du volume contient quelques articles critiques intéressants, en particulier par Ernst Mayr (*Cladistic analysis or cladistic classifications*?, pages 304-308, auparavant dans *Zeitung Zool. Syst. Evolut.-Forsch.*, 19, 94-128, 1974) ;
- un autre volume de contributions : *Prospects in Systematics*, ed. D.L. Hawksworth, Systematics Association, Clarendon Press, Oxford, 1988 ;
- un livre éducatif : *Biological Systematics* par Herbert H. Ross, Addison-Wesley, Reading (Mass.), 1973 ;
- le livre fondateur de la cladistique : *Phylogenetic Systematics* par Willi Hennig, traduction anglaise, University of Illinois Press, Urbana (Ill.), 1966. Voir également une présentation plus courte par Hennig (adaptée de son article original de 1950) dans Duncan et Stuessy ;
- un traité cladiste, débutant avec la photo de Hennig : *Phylogenetics — The Theory and Practice of Phylogenetic Systematics* par E.O. Wiley, publié par John Wiley and Sons, New York, 1981. Du même auteur, un argument poppérien en faveur de la cladistique, *Karl R. Popper, Systematics, and Classification: A Reply to Walter Bock and Other Evolutionary Taxonomists*, pages 37-47 de Duncan et Stuessy, à l'origine dans *Syst. Zool* 24:233-243, 1975 ;
- un article clair de Ernst Mayr, penchant vers la taxonomie évolutionniste tout en évoquant les autres approches avec bienveillance : *Biological Classification: Towards a Synthesis of Opposing Methodologies*, dans *Science*, vol. 214, 1961, pages 510-516 ;
- le texte fondateur des phénotypiciens : *Principles of Numerical Taxonomy*, par Robert P. Sokal et Peter H.A. Sneath, Freeman Publishing, San Francisco, 1963, révisé en 1973 ;
- un petit livre plus récent en faveur de *Transformed Cladistics* (sous-titre : *Taxonomy and Evolution*) par N.R. Scott-Ram, Cambridge University Press, 1990.

# EXERCICES

## E24.1 Piles en tableau

Écrivez en entier la classe `STACK` et son héritière `ARRAYED_STACK`, ébauchée dans ce chapitre, en utilisant la technique du "mariage d'intérêt".

## E24.2 Métataxonomie

*Voir "Taxonomie générale", page 795.*

Imaginez que la classification des formes d'héritage donnée dans ce chapitre soit une hiérarchie d'héritage. Quel(s) genre(s) mettrait-elle en jeu ?

## E24.3 Les piles de Hanoï

(Cet exercice vient d'un exemple utilisé, à la fin de 1995 et au début de 1996, par Philippe Drix sur la liste de courrier électronique française GUE.)

*Le problème des tours de Hanoï, utilisé dans de nombreux livres informatiques comme exemple de procédure récursive, vient de Édouard Lucas, "Récréations mathématiques", Paris, 1883, repris par Albert Blanchard, Paris, 1975.*

Soit une classe retardée *STACK*, munie d'une procédure *put* pour empiler un élément au sommet et d'une précondition mettant en jeu la fonction à valeur booléenne *full* (qui aurait pu également être appelée *extendible* ; pendant que vous étudierez cet exercice, vous remarquerez que le choix du nom peut changer l'attrait des différentes solutions possibles).

Considérez maintenant le célèbre problème des tours de Hanoï, dans lequel des disques sont placés sur des piles — les tours — suivant la règle qu'un disque donné ne peut être mis que sur un disque plus grand.

Est-il utile de définir la classe *HANOÏ_STACK*, qui représente de telles piles, comme héritant de *STACK* ? Si tel est le cas, comment devrait-on écrire une telle classe ? Sinon, est-ce que *HANOÏ_STACK* peut néanmoins utiliser *STACK* ? Écrivez la classe en entier pour les diverses solutions possibles ; étudiez les avantages et inconvénients de chacune, indiquez celle que vous préférez et expliquez la raison de votre choix.

## E24.4 Les polygones sont-ils des listes ?

L'implémentation de notre premier exemple d'héritage, la classe *POLYGON*, utilise un attribut de liste chaînée *vertices* pour représenter les noeuds d'un polygone. *POLYGON* ne devrait-elle pas plutôt hériter de *LINKED_LIST* [ *POINT*] ?

*"POLYGONES ET RECTANGLES", 14.1, page 446.*

## E24.5 Héritage de variation fonctionnelle

Fournissez un ou plusieurs exemples d'héritage de variation fonctionnelle. Pour chacun d'eux, dites si ce sont des applications légitimes du principe ouvert-fermé ou des exemples de ce que nous avons appelé "bidouillage organisé".

## E24.6 Exemples de classification

Dans chacun des cas suivants, indiquez quel genre d'héritage est appliqué :

- *SEGMENT* de *OPEN_FIGURE*,
- *COMPARABLE* (objets munis d'une relation d'ordre total) héritant de *PART_COMPARABLE* (objets munis d'une relation d'ordre partiel),
- Une classe d'*EXCEPTIONS*.

*"TRAITEMENT AVANCÉ DES EXCEPTIONS", 12.6, page 418.*

## E24.7 Où doivent se trouver les itérateurs ?

Serait-ce une bonne idée d'introduire des caractéristiques d'itérateurs (*while_do* et autres) dans les classes décrivant les structures de données sur lesquelles ils itèrent, comme *LIST* ? Considérez les points suivants :

- la facilité d'appliquer les itérations à des routines *action* et *test* arbitraires, choisies par l'application ;
- extensibilité : la possibilité d'ajouter de nouveaux schémas d'itération à la bibliothèque ;
- plus généralement, le respect des principes orientés objet, en particulier l'idée que les opérations n'existent pas en soi, mais seulement en relation avec certaines abstractions de données.

## E24.8 Héritage de module et de type

Supposez que nous concevions un langage comprenant deux genres d'héritage : l'extension de module et le sous-typage. Où se placerait chacun des genres d'héritage identifiés dans ce chapitre ?

## E24.9 Héritage et polymorphisme

À propos des genres d'héritage évoqués dans ce chapitre entre un parent *A* et un héritier *B*, quels sont, d'après vous, ceux qui seront utilisés, en pratique, pour les attachements polymorphes, c'est-à-dire les affectations *x* := *y* ou les passages d'arguments correspondants où *x* est de type *A* et *y* de type *B* ?