Chapitre 6

PROGRAMMER AVEC DES OBJETS

Ce chapitre présente quelques techniques usuelles de programmation par objets et une ébauche de méthodologie. Il se termine par l'exemple complet des Tours de Hanoi, qui complète les classes *Pile* et *Tour* qui nous ont servi tout au long de ce livre. Bien que ce chapitre s'applique aux langages à objets typés aussi bien qu'aux langages non typés, l'exemple sera développé dans le langage que nous avons utilisé au chapitre 3. Un certain nombre de points seront donc spécifiques des langages typés.

Contrairement à la programmation impérative ou fonctionnelle classique, la programmation par objets est centrée sur les structures de données manipulées par le programme. Le développement d'un programme suit donc les trois phases suivantes :

- Identification des classes.
- Définition du *protocole* des classes, c'est-à-dire les en-têtes des méthodes publiques (visibles de l'extérieur des classes).

- Définition des champs et implémentation des corps des méthodes. Définition éventuelle de méthodes privées (visibles uniquement de l'intérieur des classes).

L'identification correcte des classes, l'utilisation correcte de l'héritage et la bonne définition des protocoles sont déterminants lorsque l'on souhaite créer des classes réutilisables. Aussi est-il important de comprendre et de pratiquer la programmation par objets avant de l'utiliser (pour des besoins professionnels), afin d'en percevoir clairement, par l'expérimentation, les limites et les subtilités intrinsèques.

6.1 IDENTIFIER LES CLASSES

L'identification des classes d'objets, qui semble souvent aisée, nécessite en réalité une bonne pratique de la programmation par objets. Il faut éviter de définir trop peu de classes, qui correspondent à des fonctionnalités trop complexes, mais aussi de définir trop de classes, qui entretiennent des relations complexes entre elles. Le juste milieu est affaire d'expérience. Les méthodes de conception pour les langages à objets sont encore balbutiantes. Les méthodes de conception par objets (à ne pas confondre avec les précédentes) sont plus répandues, mais elles ne sont pas toujours les mieux adaptées aux langages à objets. Certaines d'entre elles par exemple ne prennent pas en compte l'héritage. Une bonne approche consiste aussi à étudier les bibliothèques de classes fournies avec les langages ou disponibles pour ceux-ci. La bibliothèque de classes de Smalltalk est à ce titre très instructive. Elle contient l'ensemble des classes qui implémentent le système d'exploitation et l'environnement de programmation graphique du système Smalltalk.

Nous proposons de distinguer plusieurs catégories de classes. Sans être exhaustive, cette classification donne une idée des différents rôles que peut jouer une classe d'objets dans une application.

Les *classes atomiques* représentent des objets autonomes, c'est-à-dire dont l'état vu de l'extérieur ne dépend pas d'autres classes. Par exemple, des classes d'objets graphiques (rectangles, cercles, etc.) ou géométriques (points, vecteurs, etc.) sont des classes atomiques. La classe des piles n'est pas une classe atomique car une pile renferme des objets auxquels on peut accéder.

Les *classes composées* sont des classes dont les instances sont des assemblages de composants, ceux-ci étant accessibles de l'extérieur. Accessible signifie que l'on peut avoir connaissance des composants, même si la classe contrôle ou limite leur accès. Par exemple, l'accès aux composants peut être en lecture seule, ou bien par l'intermédiaire de noms symboliques (indice, chaîne de caractères). La classe des Tours de Hanoi est un exemple de classe composée ; dans l'exemple que nous donnons plus loin dans ce chapitre, nous verrons que l'accès aux tours se fait de manière symbolique, par un type énuméré.

Les *classes conteneurs* sont un cas particulier de classes composées. Une instance d'une classe conteneur (un conteneur) renferme une collection d'objets et fournit des méthodes pour ajouter, enlever, rechercher des objets de la collection. Souvent, une classe conteneur fournit également un moyen d'énumérer les objets de la collection, souvent par l'intermédiaire d'une classe active ou d'un itérateur (voir ci-dessous). La classe des piles est un exemple typique de classe conteneur.

Les *classes actives* sont des classes qui représentent un processus plutôt qu'un état. En Smalltalk, les blocs sont des classes actives, qui nous ont servi dans le chapitre 4 à définir des structures de contrôle. Un autre exemple courant de classe active sont les classes d'*itérateurs*. Un itérateur est un objet qui permet d'énumérer les composants d'un autre objet. En général, l'objet itéré est un conteneur. Une méthode de l'itérateur retourne le prochain objet de l'objet énuméré. L'itérateur sert à stocker l'état courant de l'énumération, ce qui permet à plusieurs itérateurs d'être actifs simultanément sur le même objet. Un exemple d'itérateur est présenté plus loin dans cette section.

Les *classes abstraites* sont des classes qui ne sont pas prévues pour être instanciées, mais seulement pour servir de racine à une hiérarchie d'héritage. En général, les classes abstraites n'ont pas de champs. Leurs méthodes doivent être redéfinies dans les classes dérivées, ou doivent appeler de telles méthodes. Une classe abstraite sert à définir un protocole général qui ne préjuge pas de l'implémentation des classes dérivées. Un bon gage d'extensibilité d'un ensemble de classes est d'insérer des classes abstraites en des points stratégiques de l'arbre d'héritage.

Par exemple, la classe abstraite *Collection* décrite ci-dessous contient des méthodes d'ajout, de retrait, et de recherche d'un élément. Ces méthodes doivent être virtuelles si le langage impose la déclaration explicite de la liaison dynamique, comme en C++. Les classes dérivées (*Ensemble*, *Liste*, *Fichier*, etc.), doivent redéfinir ces méthodes en fonction de leur implémentation de la collection.

```
Collection = classe {
    méthodes
        procédure Ajouter (Objet);
        procédure Retirer (Objet);
        fonction Chercher (Objet) : booléen;
        fonction Suivant (Objet) : Objet;
}
```

Une classe abstraite peut également contenir des méthodes dont le corps est défini dans la classe abstraite, comme la méthode *AjouterSiAbsent* ci-dessous :

```
procédure AjouterSiAbsent (o : Objet) {
    si non Chercher (o) alors Ajouter (o);
}
```

En imposant un protocole sur ses classes dérivées, une classe abstraite permet d'obtenir une plus grande homogénéité entre les classes. Par exemple, la classe *Collection* évite d'avoir une méthode *Ajouter* dans *Ensemble* et une méthode *Insérer* dans *Liste* : les deux méthodes devront s'appeler *Ajouter*.

Une classe abstraite sert également à définir des classes générales (à défaut de génériques) : soit une classe abstraite *A* et une classe quelconque *C* ; en déclarant dans *C* des champs ou des arguments de méthodes de type *A*, on pourra utiliser C avec une plus grande gamme d'objets que si l'on avait utilisé une classe concrète. À titre d'exemple, et à partir de la classe *Collection* définie ci-dessus, on peut construire une classe générale *Itérateur*, alors qu'en l'absence de classe abstraite, on serait contraint de définir une classe d'itérateurs pour chaque classe conteneur.

```
Itérateur = classe {
    champs
        coll : Collection;
        courant : Objet;
    méthodes
        procédure Initialiser (c : Collection) {
            c := coll;
            courant := coll.Suivant (NUL);
        }
        fonction Suivant () : Objet {
            o : Objet;
            o := courant;
            si o ≠ NUL alors courant := coll.Suivant (courant);
            retourner o;
        }
}
```

Nous avons supposé ici que *Collection.Suivant(NUL)* retourne le premier objet de la collection, et que *Collection.Suivant(o)* retourne *NUL* lorsque *o* est le dernier élément de la collection. *NUL* est un objet distingué qui sert ici à simplifier l'écriture.

Héritage ou imbrication ?

Le principal problème dans l'identification des classes est le choix de la hiérarchie d'héritage. Il s'agit de déterminer si une classe doit hériter d'une autre, et si oui de laquelle. Ici, les langages typés sont plus contraignants que les langages non typés car le choix de l'héritage déterminera ce que l'on peut

faire des instances de la classe. Dans l'exemple de la classe *Collection* ci-dessus, si l'on définit une classe qui n'hérite pas de *Collection* mais qui définit la méthode *Suivant*, on ne pourra pas utiliser la classe *Itérateur* sur les objets de cette classe. Ce serait possible dans un langage non typé, car tout ce que demande la classe *Itérateur* à l'objet itéré est de répondre au message *Suivant*. En conséquence, le choix de l'arbre d'héritage est à la fois plus difficile et plus déterminant dans les langages à objets typés.

L'héritage est un mécanisme puissant, ce qui signifie qu'il peut être utilisé dans différents contextes. L'héritage peut servir à représenter la spécialisation et l'enrichissement : c'est ce pour quoi il est utilisé le plus souvent. Ainsi, dans les chapitres précédents, nous avons fait hériter la classe *Tour* de la classe *Pile* car une tour est une pile « spéciale ». Par contre, nous nous sommes gardés de faire hériter *Pile* d'une hypothétique classe *Tableau*, et nous avons préféré mettre le tableau *dans* la pile.

La relation d'ordre entre les classes qui est induite par l'héritage doit nous inciter à utiliser l'héritage lorsque les *protocoles* des classes sont compatibles, et nous en dissuader lorsque seulement les *structures* des classes sont compatibles. Le protocole d'une classe est compatible avec celui d'une autre classe s'il est inclus dans celui-ci. De même, la structure d'une classe est compatible avec celle d'une autre classe si elle est incluse dans celle-ci. C'est la compatibilité des protocoles qui permet d'utiliser l'héritage non seulement pour la spécialisation (cas d'égalité), mais aussi pour l'enrichissement. Une pile et une tour ont le même protocole : empiler, dépiler, lire le sommet. Par contre un tableau a un protocole qui permet d'accéder à un élément quelconque, ce qui est incompatible avec le protocole des piles.

La sémantique de l'héritage est telle qu'une classe dérivée doit avoir un protocole compatible, mais aussi une structure compatible, puisque les champs de la classe de base sont hérités. Cette contrainte est une source de problèmes lorsque l'on définit la hiérarchie des classes. En effet, le choix de la

hiérarchie, qui est initialement guidé uniquement par la compatibilité des protocoles, peut être invalidé plus tard à cause d'une incompatibilité de structure. La solution consiste en général à créer des classes abstraites dont les classes de structures incompatibles sont des sous-classes.

Considérons par exemple la classe *Polygone*, avec comme méthodes le dessin, la rotation et la translation. La classe *Rectangle*, qui représente des rectangles dont les côtés sont horizontaux et verticaux, est une candidate pour l'héritage, car son protocole est compatible. Mais l'on peut, pour des raisons d'efficacité, vouloir représenter le rectangle par deux points diagonaux alors que le polygone nécessite une liste de points. L'héritage devient impossible, et l'on doit introduire une classe abstraite *Forme* comme suit :

```
Forme = classe {
    méthodes
        procédure Dessiner (f : Fenêtre);
        procédure Rotation (centre : Point; angle : réel);
        procédure Translation (v : Vecteur);
}
```

```
Polygone = classe Forme {          | Rectangle = classe Forme {
    champs                         |     champs
        points : Liste [Point];    |         p1, p2 :  Point;
    méthodes                       |     méthodes
        -- idem Forme              |         -- idem Forme
}                                  |
```

Le même phénomène se reproduit si l'on veut définir une classe *Carré*. Celle-ci devrait en toute logique hériter de *Rectangle*, mais si l'on veut représenter un carré par un point et une dimension, il faut définir une classe *Quadrilatère*, sous-classe de *Forme*, dont *Rectangle* et *Carré* sont des sous-classes. Cela conduit à alourdir inutilement la hiérarchie d'héritage.

Les utilisations de l'héritage autres que la spécialisation et l'enrichissement sont généralement vouées sinon à l'échec, du moins à des solutions de compromis. L'utilisation de l'héritage

entre classes de structures compatibles mais de protocoles incompatibles, comme *Tableau* et *Pile*, peut être acceptable si le langage permet de masquer la relation d'héritage du point de vue de l'inclusion de types. C'est le cas par exemple en C++ avec l'héritage privé. Dans les autres cas, il vaut mieux y renoncer, même si cela alourdit la programmation.

Héritage multiple

L'héritage multiple est source de nombreux problèmes. Nous avons déjà évoqué les conflits de noms et l'héritage répété. Mais l'héritage multiple pose aussi des problèmes d'ordre sémantique et méthodologique. Selon notre approche de compatibilité de protocoles, on peut décider que l'héritage multiple est justifié si la sous-classe a un protocole compatible avec chacune de ses superclasses. Des conflits de protocoles peuvent apparaître si une partie du protocole de la sous-classe est incluse dans les protocoles de plus d'une de ses superclasses : nous avons vu au chapitre 3 l'exemple de la méthode *Écrire* dans le cas de la classe *TourGM* héritant de *Tour* et *Fenêtre*. Dans ce cas, il faut impérativement redéfinir dans la sous-classe la partie du protocole qui crée des conflits. Si l'héritage multiple est justifié, le protocole redéfini devrait faire appel aux protocoles des superclasses.

Comme l'héritage simple, l'héritage multiple impose l'héritage de *structure* des classes parentes. Cet héritage de structure soulève le problème de l'héritage répété : doit-on dupliquer les champs hérités d'une même classe par plusieurs chemins ? Bien que les langages fournissent divers mécanismes de contrôle, comme nous l'avons vu, il est plus sain de ne pas utiliser l'héritage multiple dans une telle situation, car les risques sont grands de rendre la hiérarchie des classes inutilisable. Notons toutefois que l'héritage répété ne provoquera pas de conflit d'héritage de structure si la classe héritée plusieurs fois est une classe abstraite sans champ : c'est la seule situation dans laquelle l'héritage répété est sans risque.

Dans le cas où l'héritage multiple n'engendre pas de conflit de protocole, et si les classes héritées n'ont pas d'ancêtre commun contenant des champs, alors on peut envisager l'utilisation de l'héritage multiple. On obtient alors une *classe agglomérée*, proche d'une classe composée qui aurait un champ par classe héritée. La différence entre classe agglomérée et classe composée est qu'une instance d'une classe agglomérée est d'un type compatible avec chacune des classes dont elle hérite. Chaque classe héritée donne une facette différente à la classe agglomérée, et les conditions que nous avons imposées assurent l'indépendance de ces facettes. Le polymorphisme d'héritage sur une classe agglomérée revient à utiliser une instance de cette classe sous l'une de ses facettes.

La similarité entre agglomération et composition nous indique que, si l'on ne souhaite pas mettre en œuvre l'héritage multiple pour l'une des raisons décrites ci-dessus, on peut lui substituer la composition. On ne dispose plus des facettes et de la facilité de programmation associée, mais on obtient un ensemble de classes plus facile à maîtriser.

Si la composition peut remplacer l'héritage multiple, l'héritage multiple ne doit pas remplacer la composition : ce n'est pas parce qu'une voiture est constituée d'un moteur, d'une carrosserie et de quatre roues qu'il faut faire hériter la classe *Voiture* de la classe *Moteur*, de la classe *Carrosserie* et quatre fois de la classe *Roue* ! C'est le protocole, et non pas la structure, qui détermine l'héritage.

6.2 DÉFINIR LES MÉTHODES

Nous venons de voir comment les classes et l'arbre d'héritage sont définis. Il est apparu, en particulier, que la notion de protocole était cruciale dans la détermination de l'héritage. Nous allons maintenant fournir des éléments afin d'aider à la définition des protocoles, c'est-à-dire des méthodes publiques des classes. Comme pour les classes, nous allons proposer une classification des méthodes.

Les *méthodes d'accès* servent à obtenir des informations sur le contenu d'un objet, et à modifier son état, sans autre effet de bord. L'accès peut consister simplement à retourner ou affecter la valeur d'un champ, ou bien à effectuer un calcul qui utilise ou modifie la valeur des champs de l'objet. Dans ce dernier cas, on parlera plutôt de *méthode de calcul*. La plupart des langages de la famille Smalltalk engendrent automatiquement une méthode d'accès en lecture et une méthode d'accès en écriture pour chaque champ. Ceci va à l'encontre de l'encapsulation car toute classe est alors complètement exposée à ses clients.

Les *méthodes de construction* permettent d'établir des relations entre les objets. Les objets doivent en effet se connaître afin de pouvoir s'envoyer des messages. Pour cela, les objets stockent des références vers d'autres objets, références qu'il est nécessaire de maintenir. Déterminer les bonnes méthodes de construction est une tâche délicate car les relations entre objets sont souvent complexes.

Par exemple, une fenêtre doit connaître les objets graphiques qu'elle contient, et un objet graphique doit savoir dans quelle fenêtre il se trouve. Deux problèmes se posent : où mettre la méthode de construction, et comment établir le lien, ici bidirectionnel, entre les objets. La méthode d'ajout peut être dans la classe des fenêtres ou dans la classe des objets graphiques. On peut aussi décider de fournir les deux méthodes. Dans tous les cas, la méthode de l'une des classes devra faire appel à une méthode de l'autre classe pour établir le lien réciproque, comme dans cet exemple :

```
procédure Fenêtre.Ajouter (og : OGraphique) {
    …                           -- ajouter og dans la fenêtre
    og.AjoutéDans (moi);        -- prévenir og
}
```

On voit ici que les deux classes *Fenêtre* et *OGraphique* entretiennent un lien privilégié : si une autre classe appelle directement *OGraphique.AjoutéDans*, la relation de réciprocité entre la fenêtre et l'objet graphique ne sera pas respectée et le système sera dans un état erroné. La seule solution est de faire

en sorte que seule la classe *Fenêtre* puisse appeler *OGraphique.AjoutéDans*. Les mécanismes de contrôle de visibilité tels que les amis de C++ et les listes d'exportation d'Eiffel permettent de réaliser cela.

Les *méthodes de contrôle* utilisent le graphe des objets qui résulte de l'application des méthodes de construction pour réaliser un calcul qui met en jeu plusieurs objets. Une méthode de contrôle ne réalise pas de calcul par elle-même, elle détermine les objets compétents et leur retransmet toute ou partie du calcul. Par exemple, le réaffichage d'une fenêtre est une méthode de contrôle qui demande à chacun des objets de la fenêtre de se redessiner.

De la même façon que pour les méthodes de construction, les méthodes de contrôle ont souvent besoin de faire appel à des méthodes spécifiques des objets qui ne doivent pas être accessibles par d'autres clients. Dans l'exemple suivant, la méthode de réaffichage d'une fenêtre doit invoquer la méthode privée *InitDessin* de la fenêtre afin de mettre en place l'environnement nécessaire pour que les objets puissent se redessiner :

```
procédure Fenêtre.Redessiner () {
    InitDessin ();              -- mettre en place l'environnement
    pour chaque objet graphique o faire
        o.Redessiner ();
}
procédure OGraphique.Redessiner () {
    …                           -- dessiner l'objet dans sa fenêtre
}
```

Les méthodes de dessin des objets graphiques doivent donc être visibles seulement par les classes capables de mettre en place cet environnement avant de les appeler, *OGraphique.Redessiner* ne doit donc être visible que de *Fenêtre*.

Les *méthodes de classe* sont des méthodes globales à une classe. Elles jouent le rôle de procédures et fonctions globales, mais bénéficient des mêmes règles de visibilité que les méthodes

normales. Dans les langages qui disposent de métaclasses, les méthodes de classes sont définies dans celles-ci ; dans les autres langages, un mécanisme spécifique permet de déclarer des champs et des méthodes de classe. Les méthodes de classe sont utilisées pour accéder aux champs de classe pour contrôler le fonctionnement de l'ensemble des instances. Nous avons déjà vu une utilisation de méthodes de classe pour numéroter les instances d'une classe. Un autre exemple d'utilisation est le contrôle, par un champ et des méthodes de classe, du type de traces émises par les méthodes pour l'aide à la mise au point.

Définir la visibilité des méthodes

Les exemples précédents ont montré l'importance de la visibilité des méthodes pour la sécurité de la programmation. Les langages non typés n'offrent pas en général de contrôle de visibilité : toute méthode, et même tout champ (grâce aux méthodes d'accès créées automatiquement), est visible de toute classe. Au contraire, les langages typés offrent différents domaines de visibilité. Cette distinction est révélatrice des différences dans l'utilisation des deux familles de langages. Avec un langage typé, on souhaite une encapsulation importante pour assurer une programmation plus sûre en effectuant le maximum de contrôles de manière statique. Les langages non typés, de leur côté, sont souvent utilisés pour le prototypage, et l'on souhaite alors un accès ouvert aux objets afin de faciliter le développement incrémental du prototype.

Il est souvent délicat de déterminer le bon domaine de visibilité de chaque méthode, même lorsque le contrôle de visibilité est sophistiqué, comme dans Eiffel. En particulier, si l'on définit une classe réutilisable, les différentes utilisations de la classe conduiront en général à modifier l'interface, le plus souvent en rendant visible un plus grand nombre de méthodes. Les domaines de visibilité dont on a besoin sont les suivants : le domaine privé à la classe, le domaine visible par les sous-classes, les domaines visibles par des classes privilégiées, et le domaine public à toute classe.

Les domaines visibles par des classes privilégiées sont les plus délicats à définir, car il faut éviter la prolifération des classes privilégiées d'une classe donnée. Dans un système bien conçu, les classes fonctionnent par groupes, et chaque classe a pour classes privilégiées les autres classes du groupe. Cela circonscrit les dépendances entre classes et facilite la réutilisation.

La double distribution

La sémantique de l'envoi de message dans les langages à objets consiste à déterminer la méthode invoquée selon la classe de l'objet receveur du message. Il arrive fréquemment que le seul receveur ne suffise pas à déterminer la bonne méthode, car celle-ci peut dépendre également de la classe effective des paramètres du message. Dans les langages typés, le polymorphisme d'héritage permet en effet de passer comme paramètres effectifs des objets d'une sous-classe de la classe déclarée pour le paramètre formel. Dans les langages non typés, la classe des paramètres n'entre pas en jeu dans la recherche de méthode.

Dans les deux cas, la technique de la *double distribution* (« double-dispatching ») permet de résoudre le problème. Nous allons l'illustrer avec l'exemple suivant : deux classes abstraites *Afficheur* et *OGraphique* fournissent des méthodes pour la représentation d'objets graphiques sur des périphériques. La méthode de dessin d'un objet graphique sur un afficheur dépend à la fois de la classe de l'afficheur et de celle de l'objet graphique. Avec les classes *Fenêtre* et *Imprimante*, la double distribution de la méthode de dessin se réalise comme suit :

```
Afficheur = classe {
    méthodes
        procédure Dessiner (o : OGraphique);
}
Fenêtre = classe Afficheur {
    méthodes
        procédure Dessiner (o : OGraphique) {
            o.AfficherFenêtre (moi);
        }
}
```

```
Imprimante = classe Afficheur {
    méthodes
        procédure Dessiner (o : OGraphique) {
            o.AfficherImpr (moi);
        }
}
```

La méthode *Dessiner* est redéfinie dans chaque classe dérivée, et appelle une méthode de *OGraphique*, dont le nom encode la sous-classe émettrice : c'est la première étape de la double distribution. La deuxième étape a lieu dans les sous-classes de *OGraphique* : chaque méthode d'affichage sur un périphérique donné y est redéfinie.

```
OGraphique = classe {
    méthodes
        procédure AfficherFenêtre (aff : Fenêtre);
        procédure AfficherImpr (aff : Imprimante);
}
Rectangle = classe OGraphique {
    …
    méthodes
        procédure AfficherFenêtre (aff : Fenêtre) {
            … -- dessiner un rectangle dans une fenêtre
        }
        procédure AfficherImpr (aff : Imprimante) {
            … -- dessiner un rectangle sur une imprimante
        }
}
Cercle = classe OGraphique {
    …
    méthodes
        procédure AfficherFenêtre (aff : Fenêtre) {
            … -- dessiner un cercle dans une fenêtre
        }
        procédure AfficherImpr (aff : Imprimante) {
            … -- dessiner un cercle sur une imprimante
        }
}
```

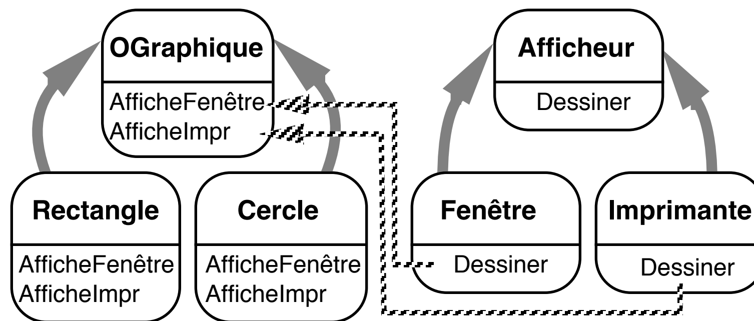

Figure 29 - Double distribution

La figure 29 illustre le mécanisme : la première distribution a lieu dans les sous-classes de *Afficheur*, et la deuxième dans les sous-classes de *OGraphique*. Étant donné un objet graphique et un afficheur, c'est finalement l'une des méthodes d'affichage des sous-classes de *OGraphique* qui sera appelée.

Si l'on rajoute une sous-classe à *Afficheur*, il faut définir la méthode *Dessiner* dans cette sous-classe ; il faut de plus ajouter la méthode d'affichage correspondante dans *OGraphique*, et une implémentation de cette méthode dans chaque sous-classe de *OGraphique*. Si l'on ajoute une sous-classe à *OGraphique*, il faut implémenter les méthodes d'affichage sur chaque périphérique dans cette nouvelle classe. On peut noter que l'ajout d'une nouvelle classe d'objets graphiques peut se faire sans toucher aux classes d'afficheurs, tandis que l'inverse n'est pas vrai. Ce critère peut aider à choisir dans quel sens doit se faire la double-distribution.

Si l'on a *n* sous-classes de *Afficheur* et *p* sous-classes de *OGraphique*, il faut implémenter *n* méthodes dans chaque sous-classe de *OGraphique*, soit *n*p* méthodes. Ceci n'est pas surprenant puisque l'affichage dépend du type d'afficheur *et* du type d'objet graphique. Mais il faut également implémenter une méthode de distribution pour chaque sous-classe de *Afficheur*, soit *n* méthodes de plus. De par les services qu'elle rend, la double distribution est d'un coût acceptable. Notons

également que, dans un langage typé qui autorise la surcharge des méthodes (méthodes de même nom avec des listes de paramètres différentes), les méthodes de distribution peuvent porter le même nom (ici se serait *Afficher*).

6.3 RÉUTILISER DES CLASSES

La réutilisation de classes est certainement l'un des avantages importants des langages à objets. La définition de classes réutilisables n'en est pas moins un travail difficile. On se trouve confronté à la définition de classes réutilisables lorsque l'on conçoit une bibliothèque, c'est-à-dire un ensemble de classes fournissant un service particulier. Une telle bibliothèque contient en général des classes à utiliser telles quelles, et d'autres classes prévues pour être dérivées : c'est la réutilisation par héritage. La généricité offre également un moyen de réutilisation puissant, mais comme elle n'est pas disponible dans tous les langages, nous ne l'évoquerons pas dans cette partie.

Lorsque l'on conçoit une bibliothèque, on a une idée du type de réutilisation qui sera employé. Mais dans la pratique, les classes sont rarement réutilisées de la façon que l'on avait imaginé : les besoins des utilisateurs ne correspondent pas exactement au service offert par la bibliothèque, ou bien le mode de réutilisation prévu ne s'adapte pas à l'application, ou bien encore les utilisateurs utilisent mal le mode de réutilisation prévu. La puissance d'une bibliothèque de classes sera d'autant plus grande qu'elle pourra être utilisée de manière non anticipée. Nous allons voir quelques techniques qui permettent d'atteindre cet objectif.

Certaines classes peuvent être prévues pour être réutilisées directement, mais la plupart du temps, la réutilisation se fait par l'intermédiaire de l'héritage. C'est notamment le cas pour les classes abstraites. La réutilisation par héritage consiste à redéfinir des méthodes de la classe de base, et à ajouter de nouveaux champs et de nouvelles méthodes. C'est la redéfinition qui pose bien sûr le plus de problèmes. La classe de

base doit définir quelles méthodes *doivent* être redéfinies, celles qui *peuvent* être redéfinies, et celles qui ne *doivent pas* l'être. Ces trois types de méthodes dépendent du protocole de la classe de base. Sans cette information, on ne peut pas réutiliser la classe de base correctement.

Une technique particulièrement sûre consiste à autoriser seulement la redéfinition de méthodes privées, comme le montre l'exemple suivant :

```
PileAbstraite = classe {
    méthodes privées
        procédure Ajouter (o : Objet);      -- à redéfinir
        procédure Retirer ();               -- à redéfinir
        fonction EstVide () : booléen;      -- à redéfinir
        fonction Dernier : Objet;           -- à redéfinir
        procédure PileVide ();              -- à redéfinir
    méthodes    -- méthodes publiques
        procédure Empiler (o : Objet) {
            Ajouter (o);
        }
        procédure Dépiler () {
            si non EstVide () alors Retirer ();
        }
        fonction Sommet : Objet {
            si non EstVide ()
                alors retourner Dernier ()
                sinon { PileVide (); retourner NUL; }
        }
}
```

Parce qu'il est très simple, cet exemple est un peu caricatural. Il montre néanmoins que, en interdisant la redéfinition des méthodes publiques, on ne peut créer une sous-classe qui ne respecte pas la sémantique d'une pile. La méthode *PileVide* permet de redéfinir la façon de signaler ou de traiter l'erreur qui consiste à accéder au sommet d'une pile vide.

De manière générale, le protocole public assure les contrôles de manière à respecter la sémantique de la classe, tandis que le protocole privé définit les opérations atomiques à définir dans chaque sous-classe. Cela n'empêche pas, le cas échéant, de redéfinir une méthode du protocole public dans une sous-classe, en particulier pour des raisons d'efficacité.

Classes dépendantes

L'utilisation de classes composées ou agglomérées conduit en général à des ensembles de classes qui sont prévus pour fonctionner ensemble. La réutilisation de ces classes doit se faire en les dérivant en parallèle, ce qui pose des problèmes spécifiques. Reprenons l'exemple des classes *Fenêtre* et *OGraphique*. Une fenêtre contient une liste d'objets à afficher et un objet graphique contient la fenêtre dans laquelle il s'affiche. La dérivation parallèle a généralement pour objectif de définir deux nouvelles classes qui, comme leurs classes de base, doivent fonctionner ensemble.

Par exemple, on cherche à définir les classes *Fenêtre3D* et *OGraphique3D* pour l'affichage d'objets à trois dimensions :

```
Fenêtre3D = classe Fenêtre {
    méthodes publiques
        procédure Ajouter (o : OGraphique);        -- héritée
}

OGraphique3D = classe OGraphique {
    méthodes privées
        procédure AjoutéDans (f : Fenêtre);        -- héritée
}
```

Une fenêtre à trois dimensions ne peut contenir que des objets graphiques à trois dimensions. Malheureusement, ceci n'est pas reflété par les méthodes héritées : la procédure *Fenêtre.Ajouter* prend un objet graphique quelconque en paramètre, de même que la procédure *OGraphique.AjoutéDans* prend une fenêtre quelconque en argument.

Dans les langages non typés, il est facile de tester la classe effective d'un objet, comme nous l'avons montré au chapitre 4 avec la classe *HPile*. Par contre, il n'y a pas de solution satisfaisante à ce problème dans les langages typés. Il faudrait redéfinir la méthode *Ajouter* avec un paramètre de type *OGraphique3D*.

Certains langages, notamment Eiffel, autorisent la redéfinition d'une méthode dans une sous-classe avec des paramètres dont les types sont inclus dans les types des paramètres correspondants dans la classe de base. Malheureusement, le contrôle de type ne peut plus être réalisé statiquement, ce qui fait d'Eiffel un langage faiblement typé, comme le montre cet exemple :

```
U = classe {                      | V = classe U {
        procédure g ();           |         procédure h ();
}                                 | }
A = classe {                      | B = classe A {
        procédure f (p : U) {     |         procédure f (p : V) {
                p.g ();           |                 p.h ();
        }                         |         }
}                                 | }
a : A; b : B; u: U;
a = b;      -- polymorphisme d'inclusion
a.f (u);    -- la liaison dynamique appelle B.f
```

Dans cet exemple, la méthode *f* est redéfinie dans la classe *B*, avec un paramètre appartenant à une sous-classe de celui déclaré pour *A.f*. L'appel *a.f(u)* est correct du point de vue du typage statique. À l'exécution, *a* contient un objet de classe *B* donc, par liaison dynamique, c'est *B.f* qui sera appelée. Malheureusement, *B.f* attend un paramètre de type *V* alors que l'on a passé un paramètre de type *U*. Pour éviter une erreur à l'exécution, le compilateur doit engendrer du code pour contrôler le type effectif des objets à l'exécution.

Dans les langages qui n'offrent pas le mécanisme d'Eiffel, la seule solution sûre consiste à fournir une méthode qui permette de connaître et de tester la classe d'un objet. Cela revient à

définir des objets qui représentent des classes, comme les métaclasses des langages à objets de la famille Smalltalk.

Dans tous les cas, la dérivation parallèle oblige à abandonner l'idée d'un typage statique, ce qui implique une attention plus grande lors de la conception du système pour limiter les situations qui font intervenir le contrôle de types dynamique.

6.4 EXEMPLE : LES TOURS DE HANOI

Nous présentons dans cette section l'exemple complet des Tours de Hanoi. Nous partons de la classe *Tour* définie dans le chapitre 3, et nous définissons la classe *Hanoi* qui représente le jeu des Tours de Hanoi.

```
TourPos = (gauche, centre, droite);
Hanoi = classe {
    champs
        tours : tableau [TourPos] de Tour;
    méthodes
        procédure Construire ();
        procédure Initialiser (n : entier);
        procédure Déplacer (de, vers : TourPos);
        procédure Jouer (de, vers, par : TourPos; n : entier);
}
```

Le type *TourPos* sert à identifier les trois tours. *Hanoi* est une classe composée offrant un accès contrôlé à ses composants. Les corps des méthodes sont les suivants :

```
procédure Hanoi.Construire () {
    tours [gauche] := allouer (Tour);
    tours [centre] := allouer (Tour);
    tours [droite] := allouer (Tour);
}

procédure Hanoi.Initialiser (n : entier) {
    tours [gauche] . Initialiser (n);
    tours [centre] . Vider ();
```

```
    tours [droite] . Vider ();
}

procédure Hanoi.Déplacer (de, vers : TourPos) {
    d : entier;
    d := tours [de] . Sommet ();
    si tours [vers] . PeutEmpiler (d) alors {
        tours [de] . Dépiler ();
        tours [vers] . Empiler (d);
    } sinon
        erreur.Écrire ("Déplacer : coup impossible");
}

procédure Hanoi.Jouer (de, vers, par : TourPos; n : entier) {
    si n > 0 alors {
        Jouer (de, par, vers, n -1);
        Déplacer (de, vers);
        sortie.Écrire (de, " -> ", vers);
        Jouer (par, vers, de, n -1);
    }
}
```

Les objets *erreur* et *sortie* sont des objets globaux qui permettent d'afficher des messages à l'écran. *allouer* permet de créer un objet dynamiquement (comme le *new* de Pascal).

On peut utiliser le jeu des Tours de Hanoi comme suit :

```
hanoi : Hanoi;
hanoi.Construire ();
hanoi.Initialiser (4);
hanoi.Déplacer (gauche, centre);
hanoi.Déplacer (gauche, droite);
hanoi.Déplacer (droite, centre);
    -> jouer : coup impossible
```

La résolution automatique du jeu se fait de la façon suivante :

```
hanoi.Initialiser (3);        -- revenir à la position initiale
hanoi.Jouer ();
    gauche -> centre
```

```
    gauche -> droite
    centre -> droite
```

Les Tours de Hanoi graphiques

Essayons maintenant d'utiliser la classe *TourG* définie au chapitre 3 pour définir la classe *HanoiG* des Tours de Hanoi graphiques. Il nous suffit de redéfinir la méthode *Construire* pour allouer des tours graphiques au lieu de tours normales. Comme les tours graphiques nécessitent une fenêtre pour l'affichage, nous allons ajouter un champ dans le nouvelle classe.

```
HanoiG = classe Hanoi {
    champs
        f : Fenêtre;
    méthodes privées
        procédure ConstruireTour (t : TourPos; x, y : entier);
    méthodes
        procédure Construire ();
}

procédure HanoiG.ConstruireTour (t : TourPos; x, y : entier) {
    tours [t] := allouer (TourG);
    tours [t] . Placer (f, x, y);
}

procédure HanoiG.Construire () {
    f := allouer (Fenêtre);
    ConstruireTour (gauche,10, 100);
    ConstruireTour (centre, 50, 100);
    ConstruireTour (droite, 90, 100);
}
```

La méthode *ConstruireTour* est une méthode auxiliaire de *Construire*. Elle doit donc être privée.

L'exemple d'utilisation de la classe *Hanoi* s'applique à un objet de la classe *HanoiG*. Toutefois, les déplacements des

disques ne seront pas visualisés graphiquement. Si l'on veut ajouter cette animation, il faut redéfinir la méthode *Déplacer*.

```
procédure HanoiG.Déplacer (de, vers : TourPos) {
    Hanoi.Déplacer (de, vers);
    -- animation
    …
}
```

Cette solution n'est pas satisfaisante car si le déplacement est invalide, *Hanoi.Déplacer* émet un message d'erreur, et l'animation ne devrait pas avoir lieu. Le seul moyen de tester la validité du déplacement dans *HanoiG.Déplacer* est de répéter le code de *Hanoi.Déplacer*, ce qui rend la classe dérivée *HanoiG* dépendante de l'implémentation de la classe *Hanoi*.

Un meilleure solution consiste à modifier la classe *Hanoi* pour la rendre plus flexible du point de vue de la réutilisation, comme nous l'avons illustré avec la classe *PileAbstraite* de la section 6.3 ci-dessus. Définissons pour cela une méthode privée *Bouger*, appelée depuis *Déplacer* lorsque le déplacement est licite :

```
Hanoi = classe {
    …
    méthodes privées
        …
        procédure Bouger (d : entier; de, vers : TourPos);
    méthodes
        procédure Déplacer (de, vers : TourPos);
        …
}

procédure Hanoi.Déplacer (de, vers : TourPos) {
    d : entier;
    d := tours [de] . Sommet ();
    si tours [vers] . OK (d) alors {
            tours [de] . Dépiler ();
            tours [vers] . Empiler (d);
            Bouger (d, de, vers);     -- notifier le déplacement
```

```
    } sinon
        erreur.Écrire ("Déplacer : coup impossible");
}

procédure Hanoi.Bouger (d : entier; de, vers : TourPos) {
    -- rien par défaut
}
```

La classe *HanoiG* devient :

```
HanoiG = classe Hanoi {
    champs
        f : Fenêtre;
    méthodes privées
        procédure ConstruireTour (t : TourPos; x, y : entier);
        procédure Bouger (d : entier; de, vers ; TourPos);
    méthodes
        procédure Construire ();
}

procédure HanoiG.Bouger (d : entier; de, vers : TourPos) {
    -- animation du disque d de la tour de vers la tour vers
    …
}
```

Il n'est plus nécessaire de redéfinir *Déplacer*. On a rendu la classe extensible en définissant une méthode privée (ici *Bouger*) qui notifie les classes dérivées d'un changement d'état significatif. Il faut bien reconnaître que, sans la tentative de dérivation, ceci ne serait pas apparu spontanément. L'écriture de classes réutilisables nécessite donc une connaissance a priori des contextes de réutilisation.

6.5 CONCLUSION

Ce chapitre nous a montré les possibilités mais aussi les limites des langages à objets. Par rapport aux autres langages, les langages à objets favorisent la modularité et la réutilisation, sans

pour autant résoudre complètement les problèmes liés à ces aspects.

Les exemples de développement d'applications avec un langage à objets ont montré que les classes aident à maîtriser de gros systèmes, à condition de les spécifier soigneusement, et de s'adapter au langage en faisant des concessions au modèle idéal des objets. En d'autres termes, les langages à objets sont un outil puissant et général, mais ne sont pas la panacée : un problème ne s'est jamais résolu par la seule vertu des objets.