Chapitre 3

LANGAGES À OBJETS TYPÉS

Ce chapitre présente les langages à objets typés, dont Simula est l'ancêtre. Ce dernier étant peu utilisé aujourd'hui, ce sont les langages plus récents C++, Eiffel et Modula3 qui nous serviront de base. La première version de C++ a été définie en 1983 par Bjarne Stroustrup aux Bell Laboratories, le même centre de recherches où sont nés Unix et le langage C. Eiffel est un langage créé à partir de 1985 par Bertrand Meyer de Interactive Software Engineering, Inc. Modula3 est une nouvelle version de Modula développée depuis 1988 au Systems Research Center de DEC sous l'impulsion de Luca Cardelli et Greg Nelson.

Nous utilisons pour les exemples un pseudo-langage dont la syntaxe, inspirée en grande partie de Pascal, se veut intuitive. Nous donnons ci-dessous la description de ce langage sous forme de notation BNF étendue, avec les conventions suivantes :

- les mots clés du langage sont en caractères gras ;
- les autres terminaux sont en italique ;

- les crochets indiquent les parties optionnelles ;
- la barre verticale dénote l'alternative ;
- les parenthèses servent à grouper des parties de règles ;
- + indique la répétition au moins une fois ;
- * indique la répétition zéro ou plusieurs fois ;
- les indicateurs de répétition peuvent être suivis d'une virgule ou d'un point-virgule qui indique le séparateur à utiliser lorsqu'il y a plus d'un élément dans la liste ;

```
prog        ::= ( classe | méthode )+
classe      ::= id-cls = classe [id-cls +,] {
                    [champs champs+]
                    [méthodes méthodes+]
                }
champs      ::= id-champ +, : type ;
méthodes    ::= procédure id-proc (param*;) ;
            |   fonction id-fonc (param*;) : type ;
type        ::= id-cls | entier | booléen
            |   tableau [ const .. const ] de type
param       ::= id-param +, : type
méthode     ::= procédure id-cls.id-proc (param*;) bloc
            |   fonction id-cls.id-fonc (param*;): type bloc
bloc        ::= { [decl+] instr*; }
decl        ::= id-var +, : type ;
instr       ::= var := expr
            |   [var.]id-proc (expr*,)
            |   tantque expr-bool faire instr
            |   si expr-bool alors corps [sinon instr]
            |   pour id-var := expr à expr faire instr
            |   retourner [expr]
            |   bloc
var         ::= id(.id-champ)* | var [ expr ]
id          ::= id-var | id-param | id-champ
```

```
expr         ::=  var | const
              |   [var.]id-fonc (expr*,)
              |   expr ( + | - | * | / ) expr
expr-bool    ::=  expr ( < | > | = | ≠ ) expr
              |   expr-bool ( et | ou ) expr-bool
              |   non expr-bool
```

Pour compléter cette description, il convient de préciser que les commentaires sont introduits par deux tirets et se poursuivent jusqu'à la fin de la ligne.

3.1 CLASSES, OBJETS, MÉTHODES

Définir une classe

La notion de classe d'objets est une extension naturelle de la notion de type enregistrement. En effet, une classe contient la description d'une liste de champs, complétée par la description d'un ensemble de méthodes.

Notre classe *Pile* peut s'écrire :

```
Pile = classe {
    champs
        pile : tableau [1..N] de entier;
        sommet : entier;
    méthodes
        procédure Empiler (valeur: entier);
        procédure Dépiler ();
        fonction Sommet () : entier;
}
```

La déclaration d'un objet correspond à l'instanciation :

```
p1 : Pile;
```

L'invocation des méthodes d'un objet utilise l'opérateur point (« . »), qui permet traditionnellement l'accès à un champ

d'un enregistrement. On peut donc considérer que les méthodes se manipulent comme des champs de l'objet :

```
p1.Empiler (10);
p1.Empiler (15);
p1.Dépiler ();
s := p1.Sommet ();          -- s vaut 10
```

Cette notation indique clairement quel est le receveur du message (ici *p1*), la méthode invoquée, et ses paramètres. Comme l'envoi de message nécessite impérativement un receveur, on ne peut invoquer les méthodes autrement que par cette notation pointée :

~~Empiler (10)~~

n'a pas de sens car on ne connaît pas le receveur du message, sauf s'il y a un receveur implicite, comme nous le verrons plus loin.

Cette même notation pointée permet d'accéder aux champs de l'objet :

```
p1.pile [5];
```

Les règles de visibilité empêchent généralement un tel accès aux champs. Comme nous l'avons vu au chapitre 2, les champs sont d'un accès privé tandis que les méthodes sont d'un accès public. Ceci signifie que les champs d'un objet sont accessibles seulement par cet objet, alors que les méthodes sont accessibles par tout objet grâce à l'envoi de message.

Définir des méthodes

La déclaration d'une classe contient les en-têtes des méthodes. Nous allons décrire leurs corps de façon séparée. Pour cela, nous utiliserons la notation *classe.méthode* qui permet une qualification complète de la méthode. En effet, deux méthodes de même nom peuvent être définies dans deux classes différentes. Rappelons que cela constitue la première forme de polymorphisme offerte par les langages à objets, le polymorphisme ad hoc.

Selon les langages, la façon de déclarer les corps des méthodes varie. La notation que nous avons choisie est inspirée de C++. Eiffel adopte une autre convention qui consiste à mettre les déclarations de méthodes dans un bloc associé à la classe où elles sont définies, ce qui pourrait être transcrit de la façon suivante dans notre pseudo-langage :

```
Pile = classe {
    procédure Empiler (valeur : entier) {
        -- corps de Empiler
    }
    -- etc.
}
```

La notation qualifiée que nous avons adoptée ici permet de séparer la déclaration de la classe de la déclaration des corps des méthodes, mais les deux mécanismes sont strictement équivalents.

Dans notre exemple, si l'on omet les tests de validité des opérations (pile vide, pile pleine), on a les définitions de corps de méthodes suivantes :

```
procédure Pile.Empiler (valeur: entier) {
    -- attention : pas de test de débordement
    sommet := sommet + 1;
    pile [sommet] := valeur;
}
procédure Pile.Dépiler () {
    -- attention : pas de test de pile vide
    sommet := sommet - 1;
}
fonction Pile.Sommet () : entier {
    retourner pile [sommet];
}
```

Une méthode est toujours invoquée avec un objet receveur, qui sert de contexte à son exécution. L'accès aux champs de l'objet receveur (*pile* et *sommet* dans notre exemple) se fait en mentionnant directement leurs noms.

Pour être plus précis, les entités accessibles depuis une méthode sont :

- l'objet receveur,
- les champs de l'objet receveur,
- les méthodes de l'objet receveur,
- les paramètres de la méthode,
- les variables locales de la méthode,
- les objets déclarés de façon globale au programme.

Les champs des objets et les paramètres étant eux-mêmes des objets, on peut invoquer leurs méthodes. Pour illustrer cela, supposons l'existence d'une classe *Fichier* et écrivons de nouvelles méthodes pour la classe *Pile* (ces méthodes doivent être ajoutées à la déclaration de la classe *Pile*) :

```
procédure Pile.Écrire (sortie : Fichier) {
    pour i := 1 à sommet faire sortie.Écrire (pile [i]);
}
procédure Pile.Vider () {
    tantque sommet > 0 faire Dépiler ();
}
```

Pile.Écrire invoque la méthode *Écrire* du paramètre *sortie*. Elle écrit sur ce fichier l'ensemble des éléments de la pile. Nous supposons ici que *Écrire* est une méthode de la classe *Fichier*.*Pile.Vider* invoque la méthode *Dépiler* sans la qualifier par un receveur, ce qui semble contraire à ce que nous avons dit plus haut. Mais ici on est dans le contexte d'un objet receveur de classe *Pile*, qui devient le receveur implicite de *Dépiler* (voir figure 7). C'est par un mécanisme identique que *sommet* représente le champ *sommet* de l'objet receveur du message.

La pseudo-variable *moi*

Bien que l'objet receveur soit implicite en ce qui concerne l'accès aux champs et aux méthodes, il est parfois nécessaire de le référencer explicitement, par exemple pour le passer en paramètre à une autre méthode. Selon les langages, il porte le

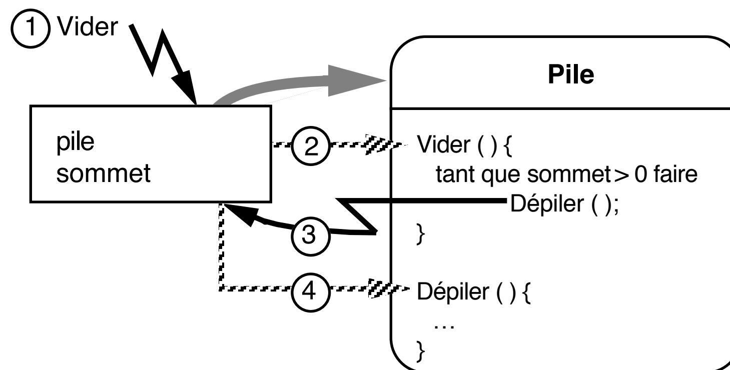

Figure 7 - Accès aux champs et aux méthodes.
Les flèches hachurées représentent l'invocation de méthode

nom réservé de « *self* » (Modula3, mais aussi Smalltalk), « *Current* » (Eiffel), « *this* » (C++). Nous l'appellerons « *moi* ». *Moi* n'est pas à proprement parler un objet, mais une façon de référencer le receveur de la méthode en cours d'exécution. On utilise pour cela le terme de *pseudo-variable*.

L'exemple suivant illustre l'utilisation de *moi*. Les classes *Sommet* et *Arc* permettent de représenter un graphe. Un sommet est relié à un ensemble d'arcs, et un arc relie deux sommets.

```
Sommet = classe {
    champs
        -- représentation des arcs adjacents
    méthodes
        procédure Ajouter (a : Arc);
}
Arc = classe {
    champs
        départ, arrivée : Sommet;
    méthodes
        procédure Relier (s1, s2 : Sommet);
}
```

Écrivons le corps de la méthode *Relier* de la classe *Arc* :

```
procédure Arc.Relier (s1, s2 : Sommet) {
    départ := s1;
    arrivée := s2;
    s1.Ajouter (moi);
    s2.Ajouter (moi);
}
```

Cet exemple montre qu'il est indispensable de pouvoir référencer explicitement le receveur du message dans le corps de la méthode invoquée : c'est l'arc qui reçoit le message *Relier* qui doit être ajouté aux sommets *s1* et *s2*.

On peut également utiliser la pseudo-variable *moi* pour qualifier les champs et les méthodes locales, mais cela n'apporte rien sinon une notation plus lourde. La méthode *Vider* de la classe *Pile* pourrait ainsi s'écrire comme suit :

```
procédure Pile.Vider () {
    tantque moi.sommet > 0 faire moi.Dépiler ();
}
```

Les classes primitives

Nous avons utilisé pour définir la classe *Pile* un champ de type entier et un champ de type tableau, considérant ces types comme prédéfinis dans le langage. Le statut de ces types prédéfinis varie d'un langage à l'autre. En général, les types atomiques (entier, booléen, caractère) ne sont pas des classes et on ne peut en hériter. Les types structurés comme les tableaux sont parfois accessibles comme des classes génériques.

On peut toujours construire une classe qui contient un champ d'un type prédéfini. Malheureusement, à moins de disposer de mécanismes de conversion implicite entre types atomiques et classes, on ne peut utiliser ces classes de façon transparente.

Par exemple, si l'on définit la classe *Entier*, contenant un champ de type *entier*, comme suit :

```
Entier = classe {
    champs
        valeur : entier;
    méthodes
        procédure Valeur (v : entier);
}
procédure Entier.Valeur (v : entier) {
    valeur := v;
}
```

et si l'on change le type *entier* par la classe *Entier* dans la classe *Pile*, on ne peut plus écrire

```
p1.Empiler (10);
```

car 10 est une valeur du type prédéfini *entier*, et non un objet de la classe *Entier*. Sans mécanisme particulier du langage, il faut écrire :

```
v : Entier;
v.Valeur (10);
p1.Empiler (v);
```

La différence de statut entre types atomiques et classes résulte d'un compromis dans l'efficacité de l'implémentation des langages à objets typés. Cette différence ne pose que peu de problèmes dans la pratique, bien qu'elle soit peu satisfaisante pour l'esprit.

3.2 HÉRITAGE

Nous allons maintenant présenter comment est mis en œuvre l'un des concepts de base des langages à objets : l'héritage. Nous allons pour cela présenter les deux principales utilisations de l'héritage, la spécialisation et l'enrichissement.

Spécialisation par héritage

Nous définissons maintenant une sous-classe de la classe *Pile*, la classe *Tour*. Rappelons qu'une tour est une pile dont les valeurs sont décroissantes.

```
Tour = classe Pile {
    méthodes
        procédure Initialiser (n : entier);
        fonction PeutEmpiler (valeur : entier) : booléen;
        procédure Empiler (valeur : entier);
}
```

L'héritage est mentionné dans l'en-tête de la déclaration (comparer avec la déclaration de la classe *Pile*). La procédure *Initialiser* empile *n* disques de tailles décroissantes :

```
procédure Tour.Initialiser (n : entier) {
    sommet := 0;
    pour i := n à 1 faire Empiler (i);
}
```

Initialiser invoque la méthode *Empiler*, qui est redéfinie dans la classe *Tour*. Définissons maintenant les corps des méthodes *PeutEmpiler* et *Empiler* :

```
fonction Tour.PeutEmpiler (valeur : entier) : booléen {
    si sommet = 0
        alors retourner vrai;
        sinon retourner valeur < Sommet ();
}
```

La méthode *PeutEmpiler* référence le champ *sommet* de sa classe de base ainsi que la méthode *Sommet* définie également dans la classe de base. Elle teste si la valeur peut être empilée, c'est-à-dire si la tour est vide ou sinon si la valeur est plus petite que le sommet courant de la tour. *Empiler* utilise *PeutEmpiler* pour décider de l'empilement effectif de la valeur :

```
procédure Tour.Empiler (valeur : entier) {
    si PeutEmpiler (valeur)
        alors Pile.Empiler (valeur);
        sinon erreur.Écrire ("impossible d'empiler");
}
```

On suppose ici l'existence d'un objet global *erreur*, de la classe *Fichier*, qui permet de communiquer des messages à l'utilisateur grâce à sa méthode *Écrire*.

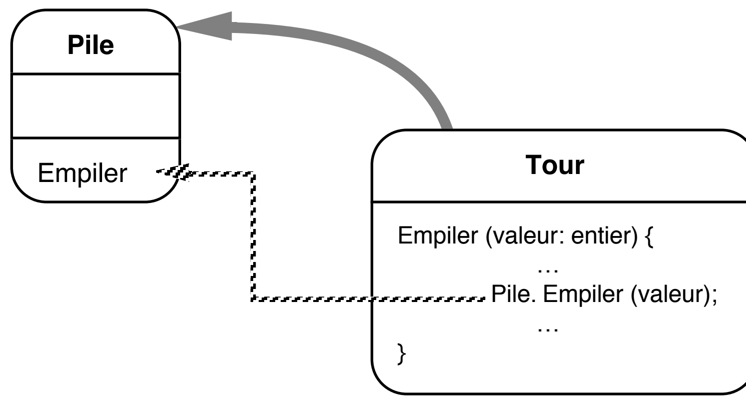

Figure 8 - Spécialisation de la méthode *Empiler*

L'appel de *Pile.Empiler (valeur)* mérite quelques explications. La classe *Tour* est une spécialisation de la classe *Pile*, c'est-à-dire que l'on a simplement redéfini une méthode de la classe *Pile*. Dans cette situation, la méthode redéfinie a souvent besoin de référencer la méthode de même nom dans la classe de base. Si l'on avait écrit

```
Empiler (valeur)
```

on aurait provoqué un appel récursif, puisque l'on est dans le corps de la méthode *Empiler* de la classe *Tour*. La notation

```
Pile.Empiler (valeur)
```

permet de qualifier le nom de la méthode appelée. Comme *Tour* hérite de *Pile*, la méthode *Empiler* de *Pile* est accessible dans le contexte courant, mais elle est cachée par sa redéfinition dans la classe *Tour* (voir figure 8). La notation qualifiée permet l'accès à la méthode de la classe de base, dans le contexte de l'objet receveur. Elle ne peut être utilisée que dans cette situation.

Une fois la classe *Tour* définie, on peut en déclarer des instances et invoquer des méthodes :

```
t : Tour;
...
t.Empiler (10);
```

```
t.Dépiler;
t.Empiler (20);
t.Empiler (5);     -- impossible d'empiler
```

Comme on l'a dit, les méthodes de la classe de base restent accessibles. Dans cet exemple, *t.Dépiler* invoque *Pile.Dépiler*.

Enrichissement par héritage

Nous allons maintenant définir une classe dérivée de la classe *Tour* en ajoutant la possibilité de représenter graphiquement la tour. Pour cela nous supposons l'existence des classes *Fenêtre* et *Rectangle*, avec les définitions partielles suivantes :

```
Fenêtre = classe {
    méthodes
        procédure Effacer ();
}

Rectangle = classe {
    méthodes
        procédure Centre (posX, posY : entier);
        procédure Taille (largeur, hauteur : entier);
        procédure Dessiner (f : Fenêtre);
}
```

TourG est une sous-classe de *Tour* définie comme suit :

```
TourG = classe Tour {
    champs
        f : Fenêtre;
        x, y : entier;
    méthodes
        procédure Placer (posX, posY : entier);
        procédure Dessiner ();
        procédure Empiler (valeur: entier);
        procédure Dépiler ();
}
```

Il s'agit ici d'un enrichissement : trois nouveaux champs indiquent la fenêtre de l'écran dans laquelle sera affichée la tour, et la position de la tour dans cette fenêtre. Chaque étage de la tour sera représenté par un rectangle de taille proportionnelle à la valeur entière qui le représente dans la tour. Deux nouvelles méthodes permettent d'affecter une position à la tour et de dessiner la tour. Enfin, les méthodes *Empiler* et *Dépiler* sont redéfinies afin d'assurer que la tour est redessinée à chaque modification. Le corps des méthodes est décrit ci-dessous.

Placer affecte la position de la tour et la redessine.

```
procédure TourG.Placer (posX, posY : entier) {
    x := posX;
    y := posY;
    Dessiner ();
}
```

Dessiner commence par effacer la fenêtre, puis redessine la tour étage par étage. *Dessiner* est similaire dans son principe à la méthode *Écrire* définie auparavant dans la classe *Pile*.

```
procédure TourG.Dessiner () {
    rect : Rectangle;
    f.Effacer ();
    pour i := 1 à sommet faire {
        rect.Centre (x, y - i);
        rect.Taille (pile [i], 1);
        rect.Dessiner (f);
    }
}
```

Empiler et *Dépiler* invoquent la méthode de même nom dans la classe de base *Tour* et redessinent la tour. On pourrait bien sûr optimiser l'affichage, mais ce n'est pas l'objet de l'exemple.

```
procédure TourG.Empiler (valeur: entier) {
    Tour.Empiler (valeur);
    Dessiner ();
}
```

```
procédure TourG.Dépiler () {
    Tour.Dépiler ();
    Dessiner ();
}
```

Il est à noter que *Tour.Dépiler* n'a pas été définie. En fait la classe *Tour* hérite *Dépiler* de la classe *Pile*, donc *Tour.Dépiler* est identique à *Pile.Dépiler*. Néanmoins, il serait imprudent d'utiliser *Pile.Dépiler* directement car on peut être amené à redéfinir *Dépiler* dans *Tour*.

3.3 HÉRITAGE MULTIPLE

L'héritage multiple permet d'utiliser plusieurs classes de base. La classe *TourG*, par exemple, représente une tour qui sait s'afficher dans une fenêtre. En changeant légèrement de point de vue, on peut considérer que la classe *TourG* est à la fois une tour et une fenêtre. On a alors un héritage multiple de *Tour* et de *Fenêtre* (figure 9). Voyons la définition de la classe *TourGM* ainsi obtenue :

```
TourGM = classe Tour, Fenêtre {
    champs
        x, y : entier;
    méthodes
        procédure Placer (posX, posY : entier);
        procédure Dessiner ();
        procédure Empiler (valeur: entier);
        procédure Dépiler ();
}
```

L'héritage multiple est mentionné en faisant apparaître la liste des classes de base dans l'en-tête de la déclaration. Le champ *f* n'apparaît plus : il est remplacé par l'héritage de *Fenêtre*.

La seule méthode qui change par rapport à la classe *TourG* est la méthode *Dessiner* :

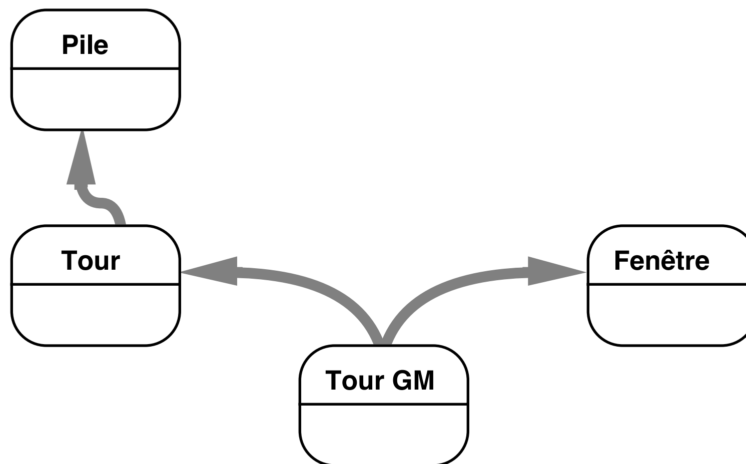

Figure 9 - Héritage multiple de la classe *TourGM*

```
procédure TourGM.Dessiner () {
    rect : Rectangle;
    Effacer ();
    pour i := 1 à sommet faire {
        rect.Centre (x, y - i);
        rect.Taille (pile [i], 1);
        rect.Dessiner (moi);
    }
}
```

On constate que l'invocation de la méthode *Effacer* n'est plus qualifiée par le champ *f*. En effet, cette méthode est maintenant héritée de la classe *Fenêtre*. Par ailleurs, l'invocation de la méthode *Dessiner* prend comme argument la pseudo-variable *moi*. En effet, *TourGM* hérite maintenant de *Fenêtre*, donc une *TourGM* est une fenêtre : on utilise ici le polymorphisme d'héritage sur le paramètre de la méthode *Dessiner*.

Bien que les différences entre les implémentations de *TourG* et *TourGM* soient minimes, l'effet de l'héritage multiple est plus important qu'il n'y paraît. En effet, alors que *TourG* n'hérite

que des méthodes de *Tour*, *TourGM* hérite de celles de *Tour* et de *Fenêtre*. Il est donc tout à fait possible d'écrire :

```
tgm : TourGM;
…
tgm.Placer (100, 100);
tgm.Empiler (20);
tgm.Empiler (10);
tgm.Effacer ();
```

L'invocation d'*Effacer* est correcte puisque *Effacer* est héritée de *Fenêtre*. Pour un objet de la classe *TourG*, cet envoi de message aurait été illicite. On voit donc que le choix d'implémenter une tour graphique par la classe *TourG* ou la classe *TourGM* dépend du contexte d'utilisation dans l'application. L'héritage, comme l'héritage multiple, ne doit pas être utilisé comme une facilité d'implémentation, mais comme une façon de spécifier des liens privilégiés entre classes.

L'héritage multiple crée cependant une ambiguïté : supposons que la classe *Fenêtre* définisse une méthode *Écrire*, qui imprime son état. La classe *Tour* de son côté hérite une méthode *Écrire* de la classe *Pile*. Que se passe-t-il si l'on écrit :

```
tgm.Écrire (fich);
```

Il y a un conflit car la classe *TourGM* hérite deux fois de la méthode *Écrire*. Les langages résolvent ce problème de différentes manières :

- l'ordre de l'héritage multiple détermine une priorité entre les classes ; dans notre cas *TourGM* hérite d'abord de *Tour*, puis de *Fenêtre*, donc *Tour.Écrire* masque *Fenêtre.Écrire*. Le résultat est donc d'imprimer la tour, c'est-à-dire la pile.
- il faut qualifier l'invocation de la méthode, par exemple *tgm.Fenêtre.Écrire (fich)*. Cela suppose donc que l'utilisateur connaisse le graphe d'héritage, ce qui ne favorise pas l'idée d'encapsulation et d'abstraction. C'est le mécanisme choisi par Modula3 et C++.
- il faut renommer, lors de l'héritage, les méthodes qui engendrent des conflits. On peut ainsi hériter de *Tour*, et de

Fenêtre en renommant la méthode *Écrire* héritée de fenêtre en *ÉcrireF*. *tgm.Écrire(fich)* imprime donc la tour, alors que *tgm.ÉcrireF(fich)* imprime la fenêtre. C'est le mécanisme imposé par Eiffel, et disponible en C++.

- il faut définir une méthode *Écrire* dans la classe *TourGM*, qui lèvera le conflit en masquant les deux méthodes *Écrire*.

La dernière méthode est toujours réalisable. Dans notre exemple, on pourrait définir cette méthode de la façon suivante :

```
procédure TourGM.Écrire (sortie : Fichier) {
    Tour.Écrire (sortie);
    Fenêtre.Écrire (sortie);
    sortie.Écrire (x);
    sortie.Écrire (y);
}
```

Cette méthode écrit la partie *Tour*, la partie *Fenêtre* et les champs propres de *TourGM*. Les invocations qualifiées *Tour.Écrire* et *Fenêtre.Écrire* lèvent l'ambiguïté en même temps qu'elles évitent l'appel récursif de *TourGM.Écrire*.

Lorsque des champs de plusieurs classes de base portent le même nom, les mêmes problèmes de conflits d'accès se posent. Ils sont résolus par un accès qualifié (en C++) ou par renommage (en Eiffel).

Nous avons évoqué au chapitre précédent d'autres problèmes concernant l'héritage multiple, et notamment l'héritage répété : que se passe-t-il si une classe hérite, directement ou indirectement, plusieurs fois de la même classe ? Faut-il dupliquer les champs de cette classe ou doivent-ils apparaître une seule fois ?

En C++, la notion de classe de base virtuelle permet de ne voir qu'une fois une classe de base qui est accessible par plusieurs chemins d'héritage. En Eiffel, le contrôle est plus fin car chaque champ d'une classe de base héritée plusieurs fois peut être dupliqué ou partagé, selon que le champ est renommé ou non.

3.4 LIAISON DYNAMIQUE

Tel que nous l'avons présenté, l'héritage des méthodes dans un langage à objets typé permet de déterminer de façon statique les invocations de méthodes : pour un objet *o* de la classe *A*, l'appel

```
o.m (paramètres)
```

est résolu en recherchant dans la classe *A* une méthode de nom *m*. Si elle n'est pas trouvée, la recherche se poursuit dans la classe de base de *A*, et ainsi de suite jusqu'à trouver la méthode ou atteindre la racine de l'arbre d'héritage. Si la méthode est trouvée, l'invocation de méthode est correcte, sinon elle est erronée.

Le compilateur peut profiter de cette recherche, destinée à vérifier la validité du programme, pour engendrer le code qui appelle directement la méthode trouvée. Cela évite une recherche similaire à l'exécution et rend donc le programme plus efficace. Cela s'appelle la *liaison statique*.

Malheureusement, le polymorphisme d'héritage rend cette optimisation invalide, comme le montre l'exemple suivant :

```
t : Tour ;
tg : TourG ;
...
t := tg ;
t.Empiler (10) ;  -- que se passe-t-il ?
```

L'affectation de la tour graphique *tg* à la tour simple *t* est correcte en vertu du polymorphisme d'héritage : une tour graphique est un cas particulier de tour, donc un objet de type tour peut référencer une tour graphique. En utilisant la liaison statique, le compilateur résout l'invocation d'*Empiler* par l'appel de *Tour.Empiler*, car *t* est déclaré de type *Tour*. Mais *t* contient en réalité, lors de l'exécution, un objet de classe *TourG*, et l'invocation de *Tour.Empiler* est invalide : il aurait fallu invoquer *TourG.Empiler*.

Dans cet exemple, le typage statique ne nous permet pas de savoir que *t* contient en réalité un objet d'une sous-classe de *Tour* et la liaison statique brise la sémantique du polymorphisme d'héritage.

Les méthodes virtuelles

Ce problème avait été noté dès Simula, et résolu en introduisant la notion de *méthode virtuelle* : en déclarant *Empiler* virtuelle, on indique d'utiliser une liaison dynamique et non plus une liaison statique : le contrôle de type a toujours lieu à la compilation, mais la détermination de la méthode à appeler a lieu à l'exécution, en fonction du type effectif de l'objet. On comprend aisément que la liaison dynamique est plus chère à l'exécution que la liaison statique, mais qu'elle est indispensable si l'on veut garder la sémantique du polymorphisme d'héritage.

L'exemple suivant montre une autre situation dans laquelle les méthodes virtuelles sont indispensables :

```
tg : TourG ;
tg.Initialiser (4);
```

On initialise une tour graphique avec quatre disques. Nous allons voir que là encore, il faut que *Empiler* ait été déclarée virtuelle. La procédure *Initialiser* est héritée de *Tour*. Voici comment nous l'avons définie :

```
procédure Tour.Initialiser (n : entier) {
    pour i := n à 1 faire Empiler (i);
}
```

Si *Empiler* n'est pas déclarée virtuelle, son invocation est résolue par liaison statique par l'appel de *Tour.Empiler*, puisque le receveur est considéré de classe *Tour*. Lorsque l'on invoque *tg.Initialiser(4)*, le receveur sera en réalité de classe *TourG*, et c'est la mauvaise version d'*Empiler* qui sera invoquée (voir figure 10). En déclarant *Empiler* virtuelle, ce problème disparaît. En l'occurrence, c'est *TourG.Empiler* qui sera invoquée, provoquant le dessin de la tour au fur et à mesure de son initialisation.

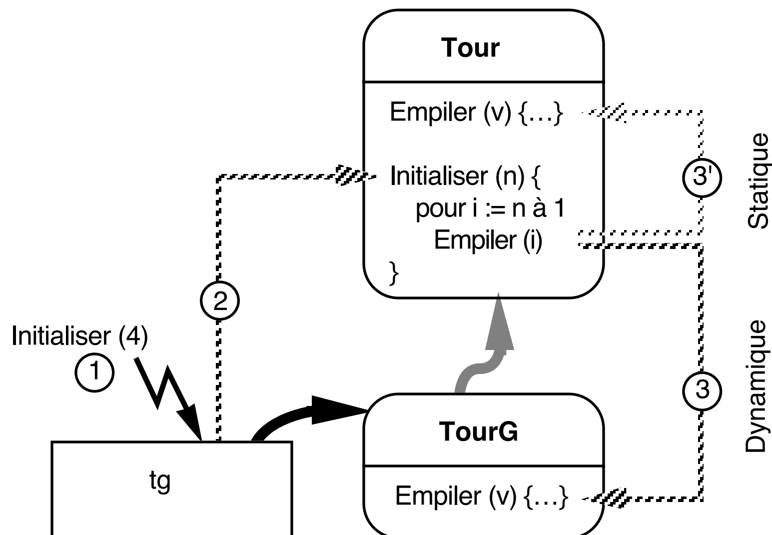

Figure 10 - Liaison statique contre liaison dynamique

L'utilisation extensive du polymorphisme dans les langages à objets pourrait laisser penser que toutes les méthodes doivent être virtuelles. C'est la solution choisie dans Eiffel et Modula3, qui assurent au programmeur que tout se passe comme si la liaison était toujours dynamique.

Par contre, C++ et Simula obligent à déclarer explicitement les méthodes virtuelles comme telles. Cela offre au programmeur la possibilité de choisir entre la sécurité de la liaison dynamique et l'efficacité de la liaison statique, à ses risques et périls. En pratique, on se rend compte qu'un nombre limité de méthodes ont effectivement besoin d'être virtuelles, mais qu'il est difficile de déterminer lesquelles, surtout lorsque les classes sont destinées à être réutilisées.

L'implémentation de la liaison dynamique

L'implémentation usuelle de la liaison dynamique consiste à attribuer à chaque méthode virtuelle un indice unique pour la

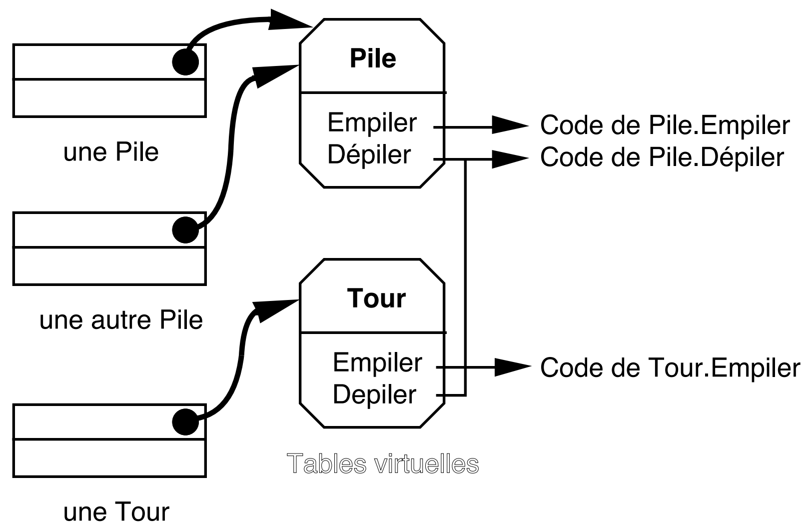

Figure 11 - Implémentation de la liaison dynamique

hiérarchie de classes à laquelle elle appartient. Lors de l'exécution, chaque classe est représentée par une table, qui contient pour un indice donné l'adresse de la méthode correspondante de cette classe. Chaque objet d'une classe contenant des méthodes virtuelles contient l'adresse de cette table (figure 11). L'invocation d'une méthode exige simplement une indirection dans cette table. Le coût de l'implémentation est donc le suivant :

- une table, dite *table virtuelle*, par classe ;
- un pointeur vers la table virtuelle par objet ;
- une indirection par invocation de méthode virtuelle.

On peut considérer ce coût comme acceptable, surtout si on le compare au coût d'invocation des méthodes dans les langages non typés (décrit à la fin de la section 4.3 du chapitre 4). On peut aussi considérer qu'il est inutile de supporter ce coût systématiquement, et c'est la raison pour laquelle Simula et C++ donnent le choix (et la responsabilité) au programmeur de

déclarer virtuelles les méthodes qu'il juge utile. Modula3, au contraire, tire parti de la table virtuelle nécessaire à chaque objet pour autoriser un objet à redéfinir des méthodes : il suffit de lui créer une table virtuelle propre. On quitte alors le modèle strict des langages de classes, puisque ce n'est plus la classe qui détient le comportement de toutes ses instances.

3.5 RÈGLES DE VISIBILITÉ

La déclaration explicite des types dans un langage assure la détection d'erreurs dès la compilation, et permet donc au programmeur de se protéger contre lui-même. Un autre aspect de cette protection concerne les règles de visibilité, c'est-à-dire les mécanismes d'encapsulation qui permettent de limiter l'accès à des données et méthodes. Le rôle principal de l'encapsulation est de masquer les détails d'implémentation d'une classe afin d'éviter que des clients extérieurs puissent la modifier impunément. On peut alors modifier a posteriori les parties cachées sans effet perceptible pour les clients.

Nous avons considéré jusqu'à présent que les règles de visibilité étaient les suivantes :

- Les champs d'une classe sont visibles seulement dans le corps des méthodes de cette classe et de ses classes dérivées.
- Les méthodes d'une classe sont visibles de l'extérieur par tout client.

L'unité de l'encapsulation : classe ou objet

La première règle ci-dessus est ambiguë : soit une classe *A* et une méthode *m* de cette classe ; on peut comprendre la règle de deux façons :

- dans le corps de *m*, on a accès aux champs de n'importe quel objet de classe *A* (en particulier le receveur de *m*) ;
- dans le corps de *m*, on n'a accès qu'aux champs de son receveur. Par exemple, si *m* a un paramètre de classe *A*, *m* n'a pas accès aux champs de ce paramètre.

Cette deuxième interprétation est plus restrictive. Elle correspond à un *domaine de visibilité* qui est l'objet : un objet n'est connu que des méthodes de sa classe, une méthode ne connaît que les champs de son receveur. La première interprétation correspond à un domaine de visibilité qui est la classe tout entière : une classe connaît ses instances, donc toute méthode de cette classe connaît toute instance de cette classe. Cette interprétation est celle utilisée dans les langages de la famille Simula. À l'inverse, les langages de la famille Smalltalk adoptent généralement la première interprétation, et considèrent donc l'objet comme l'unité de l'encapsulation.

Une fois définie cette notion de domaine de visibilité, il reste à montrer les différents mécanismes offerts par les langages. Modula3, C++ et Eiffel présentent de ce point de vue des approches différentes.

Modula3 : visible ou caché

Par défaut, tous les champs et méthodes d'une classe Modula3 sont visibles de n'importe quel client. Néanmoins, un mécanisme permet de créer un type opaque identique à une classe donnée, dont on ne rend visible que les méthodes souhaitées. Ce mécanisme, assez lourd, oblige à créer au moins deux types par classe : un type ouvert contenant les champs et méthodes, et un type fermé, utilisé par les clients, ne présentant que les méthodes utiles.

D'un autre côté, cette technique permet de définir plusieurs interfaces à une classe donnée en définissant plusieurs types limitant l'accès à cette classe de différentes manières. Ainsi on peut imaginer qu'une classe dérivée souhaite un accès plus large à sa classe de base qu'une classe quelconque.

C++ : privé, protégé ou public

En C++, les mécanismes de visibilité s'appliquent indifféremment aux champs et aux méthodes, que nous appellerons collectivement des *membres* dans cette section,

conformément à la terminologie C++. Le type de visibilité offert par C++ est intermédiaire entre ceux de Modula3 et Eiffel. Un membre d'une classe peut être déclaré avec l'un des trois niveaux de visibilité[1] suivants : privé, protégé, public.

- Un membre *privé* n'est visible que depuis les méthodes de la classe dans laquelle il est défini. Aucune classe extérieure n'y a accès, pas même une classe dérivée.
- Un membre *public* est au contraire accessible à tout client.
- Enfin, un membre *protégé* n'est accessible qu'aux méthodes de la classe et de ses classes dérivées. Cela entérine le fait qu'une classe dérivée est un client privilégié de sa classe de base.

Ces niveaux de visibilité sont transmis par héritage : un membre public d'une classe *A* est public dans toute classe dérivée de *A*. De même, un membre protégé dans *A* est protégé dans toute classe dérivée de *A*. En revanche, un membre privé n'est pas visible dans une classe héritée.

C++ offre trois mécanismes complémentaires en ce qui concerne la visibilité :

- l'*héritage privé* permet de construire une classe *B* dérivée d'une classe *A* sans que ce lien d'héritage ne soit visible de l'extérieur : les membres hérités de *A* sont privés dans *B*, et il n'y a pas de polymorphisme d'inclusion entre *B* et *A*.
- une classe peut réexporter un membre hérité avec un niveau de visibilité différent. Par exemple, la méthode protégée *m* d'une classe *A* peut être rendue publique dans une classe *B* dérivée de *A*. Dans le cas de l'héritage privé, on peut ainsi rendre visibles certains membres de la classe de base.
- une classe peut avoir des *classes amies* et des *méthodes amies* : une classe *B* amie de *A* ou une méthode *f* amie de *A* a accès à tous les membres de la classe *A*. Cela permet

[1] En C++, on parle d'*accessibilité* plutôt que de visibilité. Bien que la différence soit significative, nous n'entrerons pas dans les détails ici.

d'ouvrir une classe à des clients privilégiés, sans toutefois autoriser l'accès à une classe sans son consentement. En effet, c'est la classe *A* qui doit déclarer que *B* et *f* sont amies.

Eiffel : exportation explicite

Le mécanisme le plus raffiné pour ce qui concerne les règles de visibilité est celui d'Eiffel : chaque classe décrit la liste des membres qu'elle exporte. Chaque membre ainsi exporté peut être qualifié par une liste de classes clientes. Par défaut, un membre exporté est visible par n'importe quel client. Si l'on qualifie le membre avec une liste de classes, alors ce membre n'est visible que par ces classes et leurs classes dérivées.

Le contrôle de la visibilité des membres hérités est réalisé par le même mécanisme. On peut retransmettre les membres hérités avec la visibilité déclarée dans la classe de base, ou bien changer leur visibilité.

Des progrès à faire

Diverses raisons d'ordre syntaxique peuvent faire préférer tel ou tel mécanisme de visibilité. Dans tous les cas, il n'est pas facile de maîtriser la combinaison entre la visibilité, l'héritage et le polymorphisme.

La diversité syntaxique cache un réel problème sémantique : on ne maîtrise pas aujourd'hui les notions de visibilité de façon satisfaisante, et c'est la raison pour laquelle il n'existe pas de mécanisme simple qui réponde à des besoins par ailleurs mal définis : la visibilité doit être compatible avec l'héritage, mais des classes sans relation d'héritage doivent pouvoir se connaître de façon privilégiée, comme les amis de C++. Dès lors, un mécanisme global est nécessaire mais pas suffisant. Des travaux sur la notion de *vue*, assez proche du mécanisme de Modula3, sont prometteurs : ils permettent de définir plusieurs vues d'une classe donnée, c'est-à-dire plusieurs interfaces. Un client choisit alors la vue qui l'intéresse. Cet aspect des langages à objets n'est

donc pas figé, et l'on peut s'attendre à de nouvelles solutions à court terme.

3.6 MÉCANISMES SPÉCIFIQUES

Initialisation des objets

L'un des problèmes classiques dans les langages qui manipulent des variables (et, dans notre cas, des objets) est celui de l'initialisation. Ainsi, en Pascal, une variable non initialisée ne sera pas repérée par le compilateur et pourra provoquer un comportement aléatoire du programme. Même lorsque les variables non initialisées sont repérées par le compilateur, le programmeur doit les initialiser explicitement, ce qui alourdit le programme. La notion d'objet offre un terrain favorable pour assurer l'initialisation des objets. En effet, on peut imaginer qu'une méthode particulière prenne en charge automatiquement l'initialisation de tout nouvel objet. C++ et Eiffel offrent de tels mécanismes.

Une classe Eiffel peut déclarer une méthode spéciale, de nom prédéfini *Create*, qui assure l'initialisation des objets de cette classe. Le programmeur doit invoquer explicitement cette méthode, qui a un statut particulier. Ainsi, l'instruction *o.Create* invoque en réalité les méthodes *Create* de la classe de *o* et de chacune de ses classes de base. Cela assure que tous les composants de *o* sont initialisés correctement. Il s'ensuit que la méthode *Create* ne s'hérite pas ; le compilateur engendre au contraire une méthode *Create* pour les classes qui n'en définissent pas. Cette méthode initialise chacun des champs à une valeur par défaut dépendant de son type.

En C++, une classe peut déclarer des *constructeurs*, méthodes spéciales qui portent le nom de la classe elle-même. Divers constructeurs, avec des listes de paramètres différentes, permettent de définir plusieurs moyens d'initialisation. Un constructeur sans paramètre est un constructeur par défaut. L'appel du constructeur est réalisé automatiquement, par le

compilateur, à chaque déclaration d'objet. Comme en Eiffel, le constructeur d'une classe dérivée appelle automatiquement celui de sa classe de base. Par contre, à l'inverse d'Eiffel, l'appel du constructeur est implicite, ce qui évite les oublis malencontreux. De façon symétrique à l'initialisation, C++ permet la définition de méthodes spécifiques pour la destruction des objets : ces *destructeurs* sont, comme les constructeurs, invoqués automatiquement lorsqu'un objet devient inaccessible. Ainsi un objet déclaré comme variable locale d'une méthode voit son destructeur appelé lorsque la méthode termine son exécution.

La destruction assurée des objets est aussi importante que leur initialisation. Par exemple, si l'on considère un objet représentant un fichier, le destructeur peut assurer la fermeture du fichier lorsque celui n'est plus accessible. Le plus souvent, les destructeurs sont utilisés pour détruire des structures dynamiques contenues dans les objets. Ainsi, une classe *Liste*, contenant une liste chaînée d'objets alloués dynamiquement, peut assurer la destruction de ses éléments dans son destructeur.

Voici comment l'on pourrait définir et utiliser un constructeur et un destructeur pour la classe *Tour*. Nous avons pour cela étendu la syntaxe de notre langage par l'ajout des mots clés *constructeur* et *destructeur*.

```
Tour = classe Pile {
    …
    méthodes
        constructeur Tour (taille : entier);
        destructeur Tour ();
        …
}
constructeur Tour (taille : entier) {
    Initialiser (taille);
}
destructeur Tour () {
    tantque sommet > 0 faire Dépiler ();
}
```

```
{   -- exemple d'utilisation
    t : Tour (10);   -- constructeur Tour (entier)
    …
}   -- appel du destructeur de t
```

Généricité

Tout au long de ce chapitre, nous avons utilisé la classe *Pile* pour représenter une pile d'entiers. Si l'on voulait gérer une pile d'autres objets, il faudrait définir une nouvelle classe, probablement très proche dans sa définition et son implémentation de la classe *Pile*. La généricité, qui met en œuvre le polymorphisme paramétrique, est un mécanisme attrayant pour définir des types généraux. Par exemple, toute classe contenant une collection d'objets est un bon candidat pour une *classe générique* : tableau, liste, arbre, etc. En effet de telles classes conteneurs dépendent peu de la classe des objets contenus.

La généricité nous permet de définir une *classe générique GPile*, paramétrée par le type de ses éléments :

```
GPile = classe (T : classe) {
    champs
        pile : tableau [1..N] de T;
        sommet : entier;
    méthodes
        procédure Empiler (val : T);
        procédure Dépiler ();
        fonction Sommet () : T;
}
Pile = classe GPile (entier);          -- instanciation
p : Pile;
p.Empiler (10);
```

On ne peut utiliser la classe *GPile* telle quelle : il faut l'instancier en lui donnant le type de ses éléments. En revanche, on peut dériver *GPile* ; la classe dérivée est elle aussi générique.

La généricité et l'héritage sont deux mécanismes qui permettent de définir des familles potentiellement infinies de

classes. Aucun n'est réductible à l'autre, et un langage à objets qui offre la généricité est strictement plus puissant qu'un langage qui ne l'offre pas. Eiffel permet la définition de classes génériques. La généricité est également définie pour C++, mais elle n'est pas implémentée dans les compilateurs actuels.

Gestion dynamique des objets

Nous avons introduit la notion d'objet dans ce chapitre en généralisant la notion d'enregistrement présente dans des langages tels que Pascal. En Pascal comme dans d'autres langages, on peut déclarer des objets qui ont une durée de vie délimitée par leur portée (variables locales), mais on peut aussi gérer des variables dynamiques par l'intermédiaire des pointeurs. L'utilisation de variables dynamiques laisse au programmeur la charge de détruire les variables inutilisées, à moins que le module d'exécution du langage n'offre un ramasse-miettes qui détruise automatiquement les variables devenues inaccessibles.

En C++, les objets sont implémentés par des enregistrements, et le programmeur peut utiliser des objets automatiques ou des pointeurs vers des objets dynamiques. Dans ce cas, l'allocation dynamique et la libération de la mémoire pour les objets devenus inutiles ou inaccessibles est à sa charge. Les notions de constructeurs et de destructeurs aident cette gestion sans la rendre totalement transparente. À l'inverse, Modula3 et Eiffel assurent eux-mêmes la gestion dynamique de la mémoire. Un objet est en réalité un pointeur vers l'enregistrement de ses champs. Cette implémentation facilite le travail du programmeur, qui n'a pas à se soucier de la destruction des objets : un algorithme de ramasse-miettes s'en occupe pour lui à l'exécution. Le choix d'implémenter les objets par des pointeurs, et non par des enregistrements comme en C++, offre l'avantage de la simplicité. En revanche, l'accès à tout champ d'un objet nécessite une indirection lors de l'exécution, ce qui peut être coûteux. De plus, les algorithmes de ramasse-miettes aujourd'hui disponibles sont généralement peu efficaces, et coûtent cher en temps et en espace mémoire lors de l'exécution.

Bien entendu, ce problème n'est pas propre aux langages à objets. Néanmoins, on pourrait espérer que le choix du langage n'implique pas le choix de la gestion des objets à l'exécution. C'est le cas dans Modula3, où l'on peut indiquer pour chaque classe si l'on souhaite une gestion automatique par ramasse-miettes, ou bien une gestion à la charge du programmeur. C++ permet également au programmeur de redéfinir la gestion des objets dynamiques au niveau de chaque classe. On peut ainsi utiliser les propriétés spécifiques d'une classe pour gérer la mémoire plus efficacement qu'avec un ramasse-miettes général.

3.7 CONCLUSION

La richesse des langages à objets typés est encore loin d'être épuisée. Les langages actuels souffrent encore du lourd héritage des langages structurés. De nombreux travaux de recherche concernent la sémantique des systèmes de types mis en œuvre dans ces langages, et découvrent la complexité des problèmes mis en jeu dès lors que l'on veut combiner héritage multiple, généricité, surcharge, etc. Les langages actuels ont déjà fait la preuve de leurs qualités : sécurité pour le programmeur, facilité de maintenance, réutilisation des classes, efficacité du code exécutable. Ils sont de plus en plus facilement adoptés dans les entreprises pour le développement de logiciels complexes : systèmes d'exploitation, environnements de programmation, interfaces graphiques, simulation, etc.