Chapitre 4

SMALLTALK ET SES DÉRIVÉS

Nous allons présenter dans ce chapitre le langage Smalltalk et les langages qui en sont dérivés. Ils présentent la caractéristique commune d'être des langages non typés et interprétés ou semi-compilés.

La première version de Smalltalk date de 1972 et fut inspirée par les concepts de Simula et les idées d'Alan Kay, au laboratoire PARC de Xerox. Après une dizaine d'années d'efforts et plusieurs versions intermédiaires, notamment Smalltalk-72 et Smalltalk-76, Smalltalk-80 représente la version la plus répandue du langage et de la bibliothèque de classes qui l'accompagne. Smalltalk-80 inclut également un système d'exploitation et un environnement de programmation graphique. Xerox a aussi créé des machines spécialisées pour Smalltalk : le Star et le Dorado. Aujourd'hui, Smalltalk est disponible sur stations de travail Unix, sur Apple Macintosh, et sur compatibles IBM PC.

La syntaxe employée dans ce chapitre pour présenter les exemples est proche de celle de Smalltalk. Les commentaires sont introduits par deux tirets et se poursuivent jusqu'à la fin de la ligne. Toute instruction est un *envoi de message*, qui a l'une des formes suivantes :

```
receveur msg1
receveur msg2 argument
receveur clé1: arg1 clé2: arg2 … clén: argn
```

La première forme est un envoi de message unaire (message sans argument), par exemple :

```
3 factorielle              -- calculer 3!
Tableau Nouveau            -- créer un nouveau tableau
```

La deuxième forme est un envoi de message binaire (message avec un argument). Les expressions arithmétiques et relationnelles sont exprimées avec des messages binaires :

```
3 + 4      -- receveur = 3, msg = +, argument = 4
a < b      -- receveur = a, msg = <, argument = b
```

Enfin la troisième forme est utilisée pour des messages n-aires (à un ou plusieurs arguments) et est appelée message à mots clés. Chaque mot clé se termine par le symbole « : » et correspond à un argument :

```
tab en: 3 mettre: a
```

Ici le receveur est *tab*, le message est *en:mettre:* et les arguments sont *3* et *a*. Le nom du message est la concaténation des mots clés (y compris les deux-points). Les noms de message peuvent être préfixes les uns des autres. Dans ce cas, on prend la plus longue série de mots clés. Les parenthèses permettent de lever les ambiguïtés ou de forcer l'ordre d'évaluation. À titre d'exemple, nous utiliserons les messages *en:* et *en:mettre:*, pour l'accès aux éléments de tableau :

```
tab en: 3                          -- tab [3]
tab en: 4 mettre: a                -- tab [4] := a
tab en: (tab en: 5) mettre: a      -- tab [tab [5]] := a
```

La première expression retourne l'élément du tableau *tab* à l'indice *3*. La deuxième affecte *a* à l'élément d'indice *4*. La dernière expression affecte *a* à *tab [tab [5]]*. Dans la réalité, *en:* et *en:mettre:* ne sont pas dédiés ni réservés à l'accès aux tableaux, ceux-ci n'étant pas un type prédéfini de Smalltalk.

Les deux autres symboles utilisés dans notre langage sont le symbole d'affectation « ← » et le symbole de retour de valeur « ↑ ». Nous aurons également besoin de blocs qui seront décrits dans la section suivante.

Bien que Smalltalk permette de définir des classes et des méthodes par envoi de messages, nous utiliserons une forme plus lisible qui s'apparente à celle offerte par l'environnement graphique de programmation Smalltalk. La déclaration d'une classe aura la forme suivante :

```
classe                  idClasse
superclasse             idClasse
champs                  id1 id2 … idn
méthodes
    suite de déclarations de méthodes
```

L'ajout de méthodes dans une classe existante aura une forme identique, en omettant les lignes *superclasse* et *méthodes*.

Une déclaration de méthode se présente comme suit :

```
clé1: arg1 clé2: arg2 … clén: argn
    | idvar1 idvar2 … idvarn |
    corps de la méthode
```

La première ligne est le profil de la méthode. Ici, il s'agit d'une méthode à mots clés. La deuxième ligne est optionnelle et permet de déclarer des variables locales. Enfin le corps de la méthode est une suite d'expressions Smalltalk, séparées par des points.

4.1 TOUT EST OBJET

Les concepts de base de Smalltalk peuvent se décrire en quatre axiomes :

1. Toute entité est un objet.
2. Tout objet est l'instance d'une classe.
3. Toute classe est sous-classe d'une autre classe.
4. Tout objet est activé à la réception d'un message.

L'axiome 2 définit la notion d'*instanciation*. L'axiome 3 définit la notion d'*héritage*. Bien distinguer ces deux types de liens (lien d'instanciation *est-instance-de* et lien d'héritage *est-sous-classe-de*) est fondamental à la compréhension de Smalltalk. Il découle des axiomes 1 et 2 que toute classe est une entité du système, donc un objet. En tant qu'objet, toute classe est donc l'instance d'une classe, que l'on appelle sa *métaclasse*. Cette notion de métaclasse est fondamentale dans Smalltalk, et nous y reviendrons plus loin en détail.

Les seules entités prédéfinies dans le langage sont :

- les nombres entiers, définis dans la classe *Entier* ;
- la classe *Objet* qui est la seule à ne pas respecter le troisième axiome (*Objet* n'est la sous-classe d'aucune classe) ;
- la classe *Bloc* détaillée ci-dessous ;
- la métaclasse *Classe*.

La seule structure de contrôle est l'envoi de message aux objets. En particulier, le langage ne contient aucune structure de contrôle telles que conditionnelle, boucles, etc. Celles-ci sont définies grâce à la notion de *bloc*. Un bloc est une instance de la classe prédéfinie *Bloc*. Un bloc contient une liste optionnelle de paramètres et un ensemble d'expressions Smalltalk exécutables, séparées par des points. L'évaluation d'un bloc est obtenue en lui envoyant le message unaire *valeur*. Les blocs sont notés entre crochets :

```
incr ← [ n ← n + 1 ].
incr valeur.                    -- ajoute 1 à n
```

On peut comparer les blocs à des procédures anonymes que l'on peut exécuter par l'envoi du message *valeur*, ou à des lambda-expressions de Lisp. Les blocs peuvent avoir des paramètres. Dans ce cas, le bloc commence par la liste des noms des paramètres, préfixés d'un deux-points, et cette liste est séparée du corps du bloc par une barre verticale. Le message *valeur:* prend comme argument la valeur du paramètre réel. Nous utiliserons seulement des blocs avec un paramètre :

```
ajouter ← [ :x | n ← n + x ].          -- paramètre = x
ajouter valeur: 10.                    -- ajouter 10 à n
```

Le langage est donc réduit au strict minimum. De ce point de vue, on peut comparer Smalltalk au langage Lisp. Comme Lisp, Smalltalk est fourni avec un environnement qui évite au programmeur de tout reconstruire dans chaque programme. Cet environnement est un ensemble de classes d'intérêt général telles que booléens, tableaux, chaînes de caractères, etc. Il contient également les classes de l'environnement de programmation Smalltalk, qui permettent notamment de construire des applications graphiques interactives.

4.2 CLASSES, INSTANCES, MESSAGES

Le deuxième axiome indique que tout objet est l'instance d'une classe. Précisons ce que sont les objets et les classes : un objet contient un état, stocké dans un ensemble de champs (appelés en Smalltalk *variables d'instance*). Ces champs sont strictement privés et accessibles seulement par l'objet. Un objet ne peut être manipulé qu'à travers les messages qu'on lui envoie. Chaque objet répond à un message en activant une méthode (axiome 4). Les méthodes ne sont pas stockées dans l'objet lui-même, mais dans sa classe. Le lien d'instanciation qui unit l'objet à sa classe est donc crucial : il lui permet de retrouver la méthode à activer à la réception d'un message.

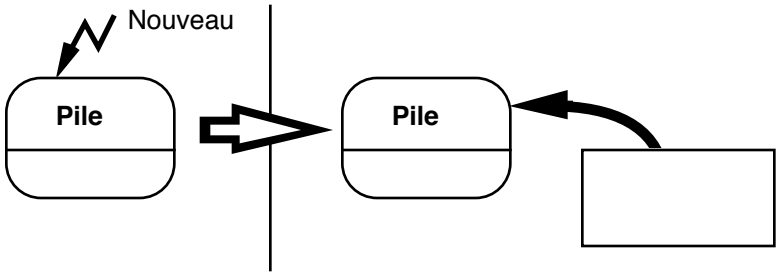

Figure 12 - Instanciation d'un objet Smalltalk

Les méthodes sont stockées dans la classe avec leurs corps, dans le *dictionnaire des méthodes*. Une classe contient également la liste des noms des champs de ses instances. Cela lui permet de créer de nouvelles instances : une classe est un objet générateur. La création d'un objet, ou *instanciation*, est réalisée en envoyant le message *Nouveau* à une classe. La classe, en tant qu'objet, répond au message *Nouveau* en créant un nouvel objet (figure 12). L'instanciation, si elle est une opération primitive du langage, est néanmoins réalisée par le seul moyen de contrôle qu'est l'envoi de message.

Pour résumer les points précédents :

- un objet contient un ensemble de champs ;
- une classe contient la description des champs de ses instances et le dictionnaire des méthodes que peuvent exécuter ses instances ;
- un objet est créé par l'envoi du message *Nouveau* à sa classe.

La correspondance entre un message reçu par un objet et la méthode à activer se fait simplement sur le nom du message : l'objet recherche dans le dictionnaire des méthodes de sa classe une méthode portant le même nom que le message. Si une telle méthode existe, elle est exécutée dans le contexte de l'objet receveur. Le corps de la méthode peut donc accéder aux champs de l'objet qui, rappelons-le, lui sont privés.

La réception d'un message qui n'a pas de méthode correspondante dans le dictionnaire des méthodes de sa classe provoque une erreur. L'erreur se manifeste par l'envoi d'un message *NeComprendsPas:* à l'objet qui a reçu le message incompris, avec pour argument le nom du message incompris. L'objet a donc l'opportunité de récupérer l'erreur : il suffit que sa classe détienne une méthode de nom *NeComprendsPas:*. Si ce n'est pas le cas, une erreur fatale d'exécution est déclenchée.

Le mécanisme de réponse à un message est dynamique et non typé : seule compte la classe de l'objet receveur, qui détermine le dictionnaire dans lequel la méthode est recherchée. Un même message peut donc donner lieu à l'exécution de méthodes différentes s'il est reçu par des objets de classes différentes. La classe des arguments n'intervient pas : si l'on veut imposer qu'un argument d'un message appartienne à une classe donnée, la méthode correspondante doit faire les tests nécessaires. Comme les classes sont des objets, on peut tester si la classe d'un objet est égale à une classe donnée.

L'aspect dynamique de l'envoi de message apparaît lorsque l'on modifie au cours de l'exécution le dictionnaire des méthodes d'une classe. Un message qui était précédemment incompris peut être défini en cours d'exécution. Comme la définition de méthode se fait par envoi de message, une méthode peut définir d'autres méthodes, de la même façon qu'une fonction Lisp peut définir d'autres fonctions.

Créer une classe

Il est temps de passer à un exemple, et de présenter l'implémentation en Smalltalk de la classe *Pile*. Pour cette classe, nous utilisons la classe *Tableau* de l'environnement Smalltalk. Nous utilisons deux messages de cette classe :

- le message *en:* qui permet d'accéder à un élément de tableau,
- le message *en:mettre:* qui permet de modifier la valeur d'un élément de tableau.

```
classe          Pile
superclasse     Objet
champs          pile sommet
méthodes
    Initialiser
        pile ← Tableau Nouveau.
        sommet ← 0.
    Empiler: unObjet
        sommet ← sommet + 1.
        pile en: sommet mettre: unObjet.
    Dépiler
        sommet ← sommet - 1.
    Sommet
        ↑ pile en: sommet.
```

La classe *Pile* hérite de la classe *Objet*, qui est une classe prédéfinie en Smalltalk. Une pile a deux champs : un tableau qui représente la pile, et l'indice du sommet de la pile dans le tableau. On peut référencer dans un corps de méthode les arguments du message, ainsi que les noms des champs de la classe. Ces noms de champs font référence aux champs de l'objet receveur du message.

Nous avons défini quatre méthodes dans la classe *Pile* :

- *Initialiser* initialise les champs de la pile ; le champ *pile* reçoit un objet de la classe *Tableau*, et le champ *sommet* est initialisé à *0*.
- *Empiler* est un message qui prend un objet en argument (l'objet à empiler). Notre pile est hétérogène : aucun contrôle n'est fait sur la classe des objets empilés. On peut donc empiler des objets de classes différentes.
- *Dépiler* enlève le sommet de pile.
- *Sommet* retourne le sommet courant.

Notre pile n'est pas très sûre : il n'y a pas de contrôle dans *Dépiler* ni dans *Sommet* pour s'assurer que la pile n'est pas vide. Nous pourrons ajouter ces tests lorsque l'on aura décrit la façon de réaliser des conditionnelles.

Pour utiliser la classe *Pile*, il suffit d'en créer une instance et de lui envoyer des messages :

```
mapile ← Pile Nouveau.  -- instanciation
mapile Initialiser.
mapile Empiler: 10.
mapile Empiler: 15.
mapile Dépiler.
s ← mapile Sommet.      -- s contient 10
o ← UneClasse Nouveau.
mapile Empiler: o.
mapile Empiler: mapile.  -- !!
```

La dernière instruction, pour surprenante qu'elle paraisse, est tout à fait correcte, puisque l'on peut empiler n'importe quel objet.

4.3 HÉRITAGE

L'axiome 3 indique que toute classe est sous-classe d'une autre classe. Cet axiome détermine l'*arbre d'héritage* qui lie les classes entre elles. Nous avons vu qu'une classe prédéfinie, *Objet*, faisait exception à cet axiome. *Objet* est la racine de l'arbre d'héritage, c'est-à-dire que, directement ou indirectement, toute classe est une sous-classe de *Objet*.

L'héritage permet de définir une classe à partir d'une autre, en conservant les propriétés de la classe dont on hérite. Une sous-classe est un enrichissement d'une classe existante : on peut ajouter de nouveaux champs et de nouvelles méthodes. On peut également modifier le comportement de la classe de base en redéfinissant des méthodes.

L'héritage modifie la recherche de méthode que nous avons décrite dans la section précédente : lorsqu'un message est reçu par un objet, celui recherche d'abord dans le dictionnaire des méthodes de sa classe. S'il ne trouve pas de méthode, il poursuit la recherche dans sa classe de base, et ainsi de suite jusqu'à trouver une méthode ou bien atteindre la racine de l'arbre d'héritage, à savoir la classe *Objet*.

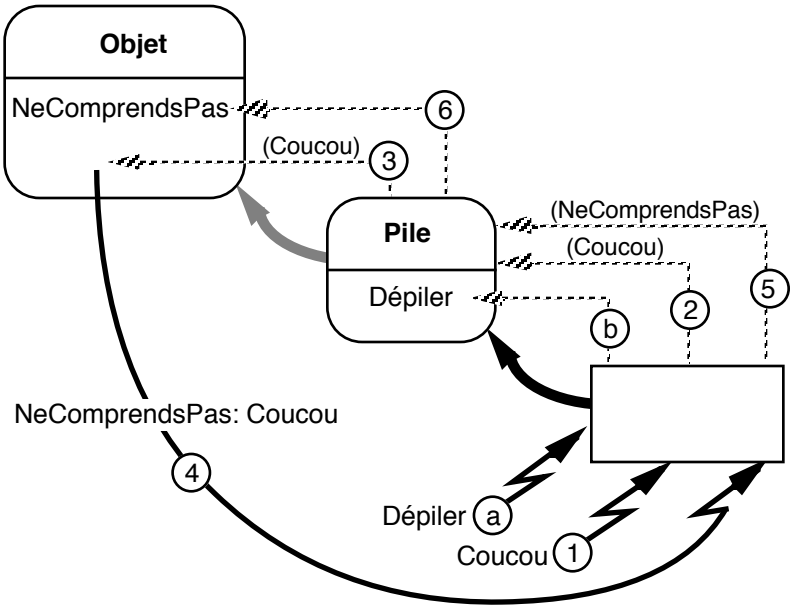

Figure 13 - Invocation de méthode.
Les flèches hachurées représentent la recherche de méthode.
L'envoi de *Dépiler* réussit, mais l'envoi de *Coucou* échoue

Dans le cas où le message n'a pas de méthode associée, le message *NeComprendsPas:* est envoyé au receveur du message, avec comme argument le nom du message incompris. La recherche d'une méthode de nom *NeComprendsPas:* suit le même mécanisme : recherche dans la classe de l'objet, puis ses superclasses successives. La classe prédéfinie *Objet* définit la méthode *NeComprendsPas*, de telle sorte que l'on est assuré de ne pas échouer cette fois-ci (figure 13). Cette technique de traitement des messages incompris permet à toute classe de redéfinir la méthode *NeComprendsPas:* et de réaliser une récupération d'erreur, sans introduire de mécanisme supplémentaire dans le langage tel que la notion d'exception.

Définir une sous-classe

Voyons comment définir une classe *HPile*, sous-classe de *Pile*, dans laquelle on force les éléments à appartenir à une même classe. Il s'agit d'ajouter un champ qui stocke la classe des objets que l'on empile, ainsi qu'une méthode qui permet d'affecter ce champ. Enfin, il faut redéfinir la méthode *Empiler* afin de réaliser le contrôle de la classe de l'objet empilé.

Pour décrire cette classe, il nous faut introduire la conditionnelle, qui a la forme suivante :

```
bool siVrai: [ blocSiVrai ] siFaux: [ blocSiFaux ]
```

Il s'agit de l'envoi du message *siVrai:siFaux:* à un objet de la classe *Booléen*. Les arguments du message sont deux blocs, correspondant aux actions à effectuer selon que l'objet receveur est l'objet *vrai* ou l'objet *faux*. Nous verrons plus loin comment sont définies la classe *Booléen* et les structures de contrôle telles que celle-ci.

La définition de la classe *HPile* est la suivante :

```
classe          HPile
superclasse     Pile
champs          classe
méthodes
    Classe: uneClasse
            classe ← uneClasse.
            self Initialiser.
    Empiler: unObjet
            unObjet Classe = classe
                siVrai: [ super Empiler: unObjet ]
                siFaux: [ "Empiler: erreur de classe" Écrire ]
```

La méthode *Classe:* permet de définir la classe des objets que l'on met dans la pile. Elle affecte le champ *classe* et exécute *self Initialiser*. La méthode *Empiler:* est redéfinie de manière à tester la classe de l'objet empilé. Pour cela, on utilise la méthode *Classe*, définie dans la classe *Objet*, qui retourne la classe de son receveur. Le receveur du message *siVrai:siFaux:* est le résultat

de l'expression *unObjet Classe = classe*. Cette expression se décompose en un envoi du message unaire *Classe* à *unObjet*. Le résultat est comparé au champ *classe* par le message binaire =. Le bloc à exécuter si l'expression est vraie, c'est-à-dire si l'objet est de la classe attendue, est *super Empiler*. Si l'expression est fausse, un message d'erreur est émis en envoyant le message *Écrire* à une chaîne de caractères. Il nous reste à décrire *self* (utilisé dans *Classe:*) et *super* (utilisé dans *Empiler:*).

Self et *Super*

Nous avons vu que le corps d'une méthode s'exécute dans le contexte de l'objet receveur du message. À l'intérieur du corps d'une méthode, on a directement accès aux champs de l'objet receveur. En revanche, si l'on veut envoyer un message au receveur lui-même, il faut un moyen de le nommer. On utilise alors la pseudo-variable *self*, qui désigne le receveur de la méthode en cours d'exécution. Ainsi, dans la méthode *Classe:* ci-dessus, le receveur du message s'envoie à lui-même (*self*) le message *Initialiser*.

Lorsque l'on veut redéfinir une méthode dans une sous-classe, on a en général besoin d'utiliser la méthode de même nom définie dans la classe de base. Pour cela, on utilise une autre pseudo-variable, *super*, qui dénote l'objet receveur en le considérant comme une instance de sa classe de base (ou superclasse, d'où le nom de *super*). Ceci est illustré par la méthode *Empiler:*. Il s'agit de tester une condition (l'objet est-il de la bonne classe ?), et si celle-ci est satisfaite, d'empiler effectivement l'élément. Pour cela, on envoie le message *Empiler:* à *super*. Si on l'envoyait à *self*, on aurait un appel récursif, ce qui n'est pas l'effet souhaité. L'envoi à *super* signifie que la recherche d'une méthode pour le message *Empiler* commence à la superclasse du receveur, et non pas à sa classe comme c'est le cas normalement (figure 14).

Dans la pratique, on utilise *super* seulement dans le corps d'une méthode redéfinie, comme nous venons de le faire. C'est en effet la seule situation dans laquelle il est justifié d'invoquer

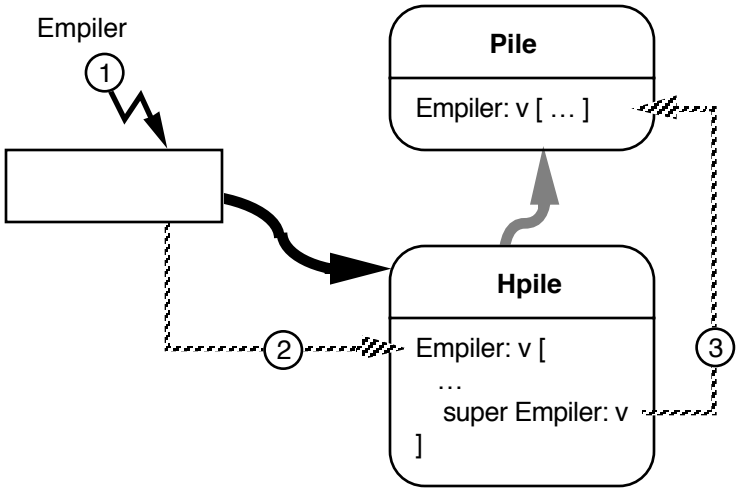

Figure 14 - Les pseudo-variables *self* et *super*

l'implémentation de la même méthode dans la classe de base. Utiliser *super* dans un autre contexte revient à transgresser la classe de l'objet receveur.

L'implémentation de l'envoi de message

L'héritage et la possibilité de modifier dynamiquement les dictionnaires des méthodes impliquent que, pour chaque envoi de message, on effectue à l'exécution une recherche dans la chaîne des superclasses de la méthode correspondant au message. Il en résulte que l'envoi de message est très coûteux.

Comme la hiérarchie d'héritage et les dictionnaires de méthodes changent peu par rapport au nombre de messages envoyés, une augmentation des performances importante est obtenue dans la plupart des implémentations en utilisant un *cache*. Les entrées du cache sont constituées de couples <nom de classe, sélecteur de message>. Pour chaque entrée, le cache contient le résultat de la recherche du message dans la classe et ses superclasses. Lors d'un envoi de message, on cherche dans

le cache une entrée correspondant à la classe de l'objet receveur du message et au sélecteur du message. Si l'entrée n'est pas dans le cache on effectue la recherche dans les dictionnaires, et on entre le résultat dans le cache. Avec cette technique, on obtient facilement des taux de présence dans le cache de plus de 98%, et une augmentation importante des performances.

Le cache doit être invalidé, c'est-à-dire vidé, chaque fois que l'arbre d'héritage ou un dictionnaire de méthode change. Chaque ajout de classe ou de méthode coûte donc cher. Aussi, diverses techniques permettent de ne pas invalider tout le cache afin de réduire le coût de ces modifications.

4.4 LES STRUCTURES DE CONTRÔLE

Un aspect original de Smalltalk est de ne pas contenir de structures de contrôle prédéfinies. Celles-ci sont définies par des classes et des méthodes, à l'aide de la classe prédéfinie des blocs, comme nous allons l'illustrer ici.

Les booléens et la conditionnelle

Revenons tout d'abord sur la conditionnelle, que nous avons déjà utilisée. Le receveur du message *siVrai:siFaux:* est un booléen ; selon sa valeur, c'est l'un des deux blocs arguments du message qui est exécuté. Pour produire cela, on définit trois classes et deux objets :

- la classe *Booléen*, qui n'a aucune instance ;
- la classe *Vrai*, sous-classe de *Booléen*, qui a une seule instance : l'objet *vrai* ;
- la classe *Faux*, sous-classe de *Booléen*, qui a une seule instance : l'objet *faux*.

Le receveur de *siVrai:siFaux:* ne peut être que l'objet *vrai* ou l'objet *faux*. Ainsi, l'évaluation de 3 < 4 retourne l'objet *vrai*, alors que 1 = 0 retourne l'objet *faux*. Il suffit donc de définir la méthode *siVrai:siFaux:* dans chacune des classes *Vrai* et *Faux* :

```
classe          Vrai
superclasse     Booléen
champs
méthodes
    siVrai: blocVrai siFaux: blocFaux
            ↑ blocVrai valeur.

classe          Faux
superclasse     Booléen
champs
méthodes
    siVrai: blocVrai siFaux: blocFaux
            ↑ blocFaux valeur.
```

Comme on le voit, si l'objet *vrai* reçoit un message *siVrai:siFaux:*, il retourne la valeur du premier argument ; si c'est l'objet *faux* qui reçoit le message, il retourne la valeur du deuxième argument. L'héritage nous a permis de reproduire un comportement conditionnel. Cette technique s'applique à d'autres messages, tels que les opérateurs logiques :

```
classe          Vrai
méthodes
    non
            ↑ faux.
    ou: unBool
            ↑ vrai.
    et: unBool
            ↑ unBool.
```

```
classe          Faux
méthodes
    non
            ↑ vrai.
ou: unBool
            ↑ unBool.
et: unBool
            ↑ faux.
```

La définition des méthodes *non*, *ou:* et *et:* dans les deux classes *Vrai* et *Faux* permet d'implémenter facilement les tables de vérité de ces opérateurs. La figure 15 illustre l'évaluation d'une expression booléenne qui utilise ces opérateurs.

Jusqu'ici la classe *Booléen* ne nous a pas servi : en fait *Vrai* et *Faux* auraient très bien pu hériter directement de *Objet*. Nous allons maintenant utiliser la classe *Booléen* pour factoriser des méthodes entre les classes *Vrai* et *Faux* :

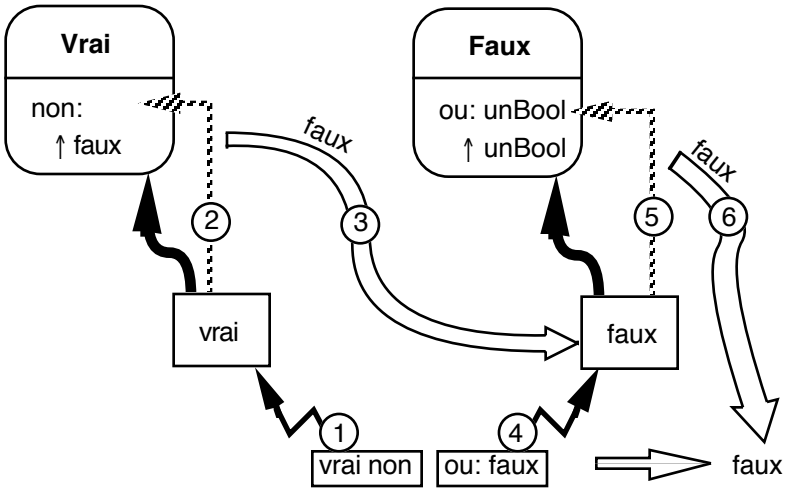

Figure 15 - Évaluation de l'expression *(vrai non) ou: faux*

```
classe          Booléen
superclasse     Objet
champs
méthodes
    siVrai: unBloc
            ↑ self siVrai: unBloc siFaux: [ ].
    siFaux: unBloc
            ↑ self siVrai: [ ] siFaux: unBloc.
    xor: unBool
            ↑ (self ou: unBool)
            et: ((self et: unBool) non).
```

On a défini dans la classe *Booléen* deux conditionnelles *siVrai:* et *siFaux:*, qui correspondent à la conditionnelle *siVrai:siFaux:* lorsque l'on omet l'un des blocs. Il est inutile de définir ces conditionnelles dans les deux classes *Vrai* et *Faux*, comme le montre la figure 16. De façon similaire, le *ou exclusif* (xor) est défini à partir des opérateurs élémentaires *et:*, *ou:* et *non*, selon l'expression : a xor b = (a ou b) et non (a et b).

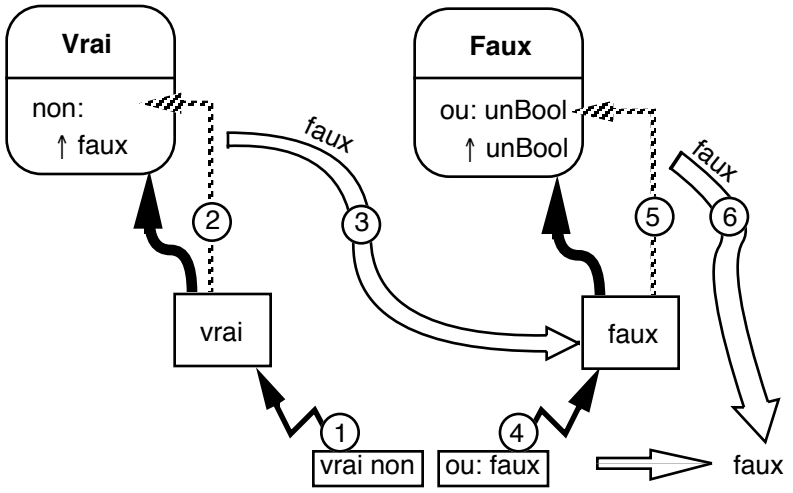

Figure 16 - Évaluation des conditionnelles

Les blocs et les boucles

En ce qui concerne les structures de boucles, ce n'est plus dans la classe *Booléen* que vont se faire les définitions, mais dans la classe des blocs. Décrivons tout d'abord la boucle *tant-que* :

```
classe          Bloc
méthodes
    tantQueVrai: corps
        (self valeur) siVrai: [ corps valeur.
                                self tantQueVrai: corps ].
```

Le message *tantQueVrai:* s'utilise de la façon suivante :

```
[ x < 10 ] tantQueVrai: [ s ← s + x ; x ← x -1 ]
```

Le receveur est un bloc, car celui-ci doit être évalué à chaque tour de boucle, comme le montre l'implémentation de la méthode *tantQueVrai:* : le bloc receveur s'évalue ; s'il est vrai, le corps de la boucle est évalué, et l'itération a lieu grâce à

l'envoi (récursif) du message *tantQueVrai:* au bloc receveur. Comme toujours avec Smalltalk, il n'y a aucun contrôle de type et rien n'empêche d'écrire :

```
[ 10 ] tantQueVrai: [ … ].
"coucou" tantQueVrai: [ … ].
```

Selon la valeur retournée par l'évaluation du bloc receveur, une erreur aura lieu ou l'exécution pourra continuer. Dans le premier cas, l'évaluation du bloc receveur retourne la valeur 10, de la classe *Entier*, qui ne sait répondre à *siVrai:*. Dans le deuxième cas, le receveur est une chaîne de caractères, qui ne sait répondre au message *valeur*. Mais si l'on définissait ces méthodes, l'exécution pourrait se poursuivre.

Le dernier type de structure de contrôle que nous allons examiner est l'itération. Il s'agit d'exécuter un corps de boucle (un bloc) un certain nombre de fois. Pour reproduire l'équivalent de la boucle itérative de Pascal, il faut définir une classe *Intervalle*, qui contient deux entiers représentant les bornes inférieure et supérieure de l'intervalle. La méthode *répéter:*, envoyée à un intervalle, réalise l'itération :

```
classe          Intervalle
superclasse     Objet
champs          inf sup
méthodes
    Inf: i Sup: s
            inf ← i. sup ← s.
    répéter: corps
            | i |
            i ← inf.
            [ i < sup ] tantQueVrai:
                [ corps valeur: i.
                  i ← i + 1 ].
```

La méthode *Inf:Sup:* permet d'initialiser les bornes de l'intervalle. La méthode *répéter:* introduit une variable locale *i*. Cette variable sert de compteur de boucle, et l'on utilise le message *tantQueVrai:* des blocs pour réaliser l'itération.

Le message *répéter:* s'utilise comme suit :

```
bornes ← Intervalle Nouveau.
bornes Inf: 10 Sup: 20.
s ← 0.
bornes répéter [ :x | s ← s + x ].
s Écrire.                                    -- s = 165
```

Pour faciliter l'écriture de l'itération, nous allons ajouter un message dans la classe prédéfinie *Entier* :

```
classe          Entier
méthodes
    à: val      ↑ (Intervalle Nouveau) Inf: self Sup: val.
```

Ce message permet de créer un intervalle en envoyant à un entier le message *à:* avec un entier en argument. Le receveur est la borne inférieure de l'intervalle, l'argument la borne supérieure. L'exemple précédent s'écrit alors :

```
s ← 0.
(10 à: 20) répéter [ :x | s ← s + x ].
s Écrire.
```

L'expression *10 à: 20* retourne un intervalle auquel on envoie le message d'itération *répéter*.

Cette technique d'itération s'applique à de nombreuses classes. Ainsi, Smalltalk fournit un grand nombre de classes conteneurs pour le stockage d'objets (tableaux, listes, ensembles, dictionnaires, etc.). La plupart de ces classes définissent une méthode qui permet l'itération de leurs éléments.

Nous pouvons appliquer cela à notre classe *Pile*, en lui ajoutant une méthode *répéter:* qui évalue un bloc pour les éléments successifs de la pile :

```
classe          Pile
méthodes
    répéter: unBloc
            (1 à: sommet) répéter:
                    [ :i | unBloc valeur: (pile en: i) ].
```

On utilise un intervalle créé par le message *à:* représentant l'ensemble des indices valides de la pile. L'intervalle est énuméré par la méthode *répéter:*. Pour chaque élément de l'intervalle, on évalue le bloc argument avec comme paramètre l'élément de pile.

Pour imprimer le contenu d'une pile, il suffit d'écrire :

```
pile ← Pile Nouveau.
pile Initialiser.
-- empiler des éléments …
pile répéter: [ :e | e Écrire ].
```

Nous avons défini la méthode *répéter:* dans la classe *Pile* à l'aide de la méthode *répéter:* de la classe *Intervalle*, c'est-à-dire que l'on utilise le polymorphisme ad hoc, intrinsèque aux langages objets. Ceci nous permet par exemple de définir dans la classe *Objet* elle-même une méthode qui imprime le contenu d'un objet de la façon suivante :

```
classe          Objet
méthodes
    ÉcrireContenu
        self répéter: [ :e | e Écrire ].
```

Tout objet qui sait répondre à *répéter:* pourra exécuter *ÉcrireContenu* :

```
(10 à: 20) ÉcrireContenu.
pile ← Pile Nouveau.
pile Initialiser.
-- empiler des éléments …
pile ÉcrireContenu.
```

Dans cette section, nous avons défini l'itération (message *répéter:*) de la classe *Pile* en fonction de l'itération des intervalles. Celle-ci est définie en fonction de la répétition des blocs (*tantQueVrai:*), elle-même définie récursivement à l'aide de la conditionnelle *siVrai:*. Enfin cette conditionnelle est définie en fonction de la conditionnelle générale *siVrai:siFaux:*, définie dans les deux classes *Vrai* et *Faux*.

Cet exemple illustre la souplesse et la puissance de Smalltalk, grâce à l'utilisation extensive de la liaison dynamique : il suffit de rajouter des méthodes à une classe (comme *répéter:* dans la classe *Pile*), pour que les instances de cette classe soient dotées de nouvelles capacités (comme *ÉcrireContenu*). Ceci fait de Smalltalk un environnement idéal pour le maquettage et le prototypage d'applications. En revanche, l'absence de typage, donc d'assurance a priori qu'un programme ne déclenchera pas d'exécution de messages indéfinis, est souvent un obstacle à la réalisation d'applications finales.

4.5 MÉTACLASSES

Nous avons vu dès la présentation des axiomes de base de Smalltalk que toute classe est un objet, et appartient donc à une classe, appelée *métaclasse*. Cette caractéristique est à la base de nombreuses possibilités intéressantes dans Smalltalk.

En premier lieu, la notion de métaclasse permet de réaliser l'instanciation par un envoi de message : l'envoi du message *Nouveau* à une classe retourne une instance de cette classe. Le message étant envoyé à une classe, c'est dans sa métaclasse qu'est cherchée la méthode correspondante (figure 17).

La métaclasse ne sert pas seulement à l'instanciation. En effet, une classe contient la définition de ses instances, c'est-à-dire la liste des noms de champs et le dictionnaire des méthodes. On peut donc interroger la classe pour savoir si une méthode particulière est définie, pour modifier le dictionnaire des méthodes, et pour définir de nouvelles sous-classes. Une métaclasse définit pour cela les méthodes *Connaît:*, *Superclasse* et *DériveDe:*. Le message *Connaît:* permet de savoir si une classe sait répondre au message dont le nom est passé en argument. Le message *Superclasse* retourne la classe de base de la classe receveur, et enfin le message *DériveDe:* permet de tester si la classe receveur est une classe dérivée de la classe dont le nom est passé en paramètre. Voici quelques exemples d'utilisation de ces messages :

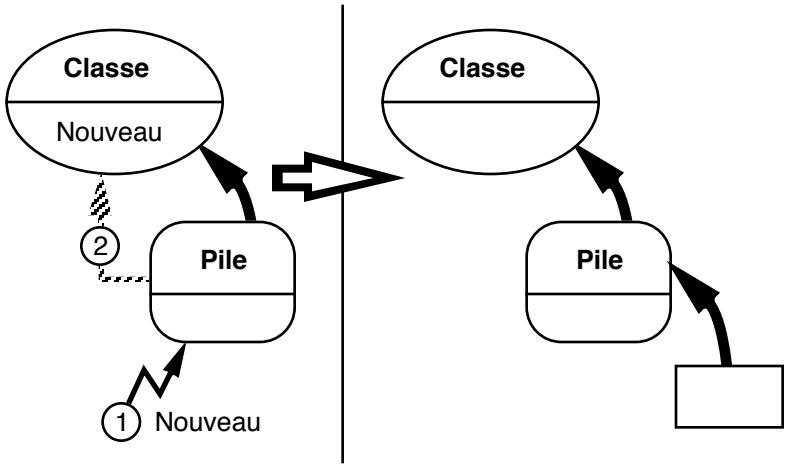

Figure 17 - Création d'un objet : utilisation de la métaclasse

```
Pile Connaît: "Empiler:".          -- vrai
Pile Connaît: "SiVrai:".           -- faux
Vrai Superclasse.                  -- Booléen
Vrai DériveDe: "Pile".             -- faux
HPile DériveDe: "Objet".           -- vrai
```

Le corps des méthodes correspondant à ces messages se trouve dans le dictionnaire des méthodes de la métaclasse de leur receveur, comme le prouvent les exemples suivants :

```
Pile Connaît: "Nouveau".                -- faux
(Pile Classe) Connaît: "Nouveau".       -- vrai
```

Dans le cas de Smalltalk, plusieurs modèles de métaclasses ont été expérimentés. Le plus simple comprenait une seule métaclasse dans le système, nommée *Classe*. Dans les versions suivantes de Smalltalk, cela s'est avéré être une limitation, car on ne pouvait différencier les métaclasses de classes distinctes. Le modèle de Smalltalk-80 définit une métaclasse par classe, et la hiérarchie d'héritage des métaclasses suit celle des classes. Chaque classe est l'unique instance de sa métaclasse. Pour

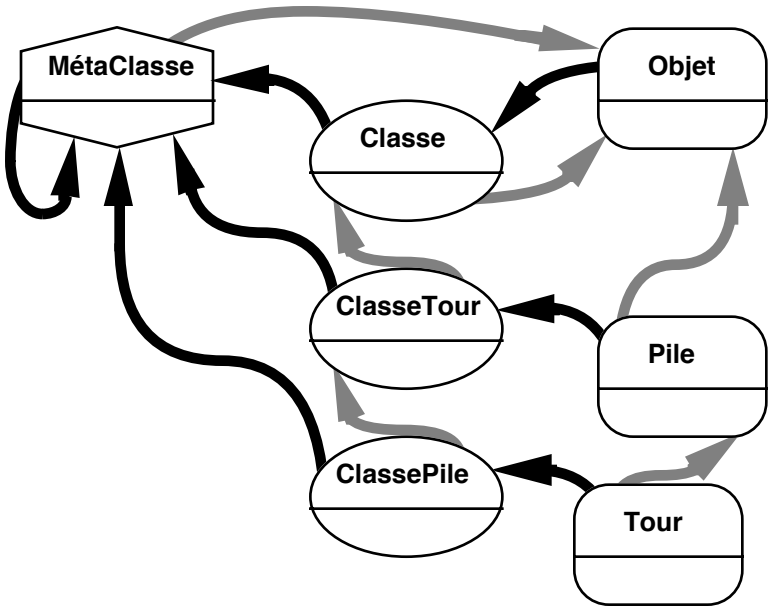

Figure 18 - Le modèle des métaclasses de Smalltalk-80

l'utilisateur, les métaclasses sont transparentes car elles sont créées automatiquement par la méthode de définition de classe.

Par convention, nous appellerons *ClasseX* la métaclasse de la classe *X*. La métaclasse de *Objet* est donc *ClasseObjet*, celle de *Pile* est *ClassePile*. Comme *Pile* hérite de *Objet*, *ClassePile* hérite de *ClasseObjet*. Les métaclasses héritent de *Classe*, qui elle-même hérite de *Objet*.

Le mécanisme des métaclasses induit une régression à l'infini. En effet, une métaclasse est aussi un objet, qui a donc une classe, une métaclasse, etc. Comme dans le cas de la hiérarchie d'héritage, cette régression est artificiellement interrompue par un bouclage dans le chaînage des métaclasses : dans Smalltalk-80, toutes les métaclasses héritent de la classe *Classe*, et sont des instances de *MétaClasse*, selon le schéma de la figure 18.

Du point de vue du programmeur, cet artifice est de peu d'importance. Il assure au langage la *méta-circularité*, c'est-à-dire la capacité à se décrire lui-même. Grâce aux métaclasses on peut écrire un interprète Smalltalk en Smalltalk.

L'intérêt des métaclasses pour le programmeur

Pour le programmeur, les métaclasses permettent d'une part de définir des méthodes de classe, et d'autre part de partager des champs entre toutes les instances d'une classe.

Les *méthodes de classe* sont des méthodes définies dans une métaclasse, et qui sont utilisées lorsque l'on envoie des messages à une classe. L'utilisation la plus répandue des méthodes de classe est la redéfinition de la méthode d'instanciation *Nouveau*, et la définition d'autres méthodes d'instanciation prenant des paramètres. On peut rapprocher cela des constructeurs de certains langages à objets typés (voir chapitre 3, section 3.6).

Reprenons le cas de la classe *Pile*. Nous avons défini dans cette classe la méthode *Initialiser* qui permet d'instancier et d'initialiser les champs de la pile. Lors de l'utilisation de la classe *Pile*, il faut s'assurer d'initialiser chaque pile après l'avoir instanciée avec *Nouveau* :

```
pile ← Pile Nouveau.
pile Initialiser.
```

Si l'on oublie l'initialisation, la pile ne pourra pas fonctionner comme prévu. Il serait plus sûr d'assurer l'initialisation lors de l'instanciation. Il suffit pour cela de redéfinir la méthode *Nouveau*, de la façon suivante :

```
classe          ClassePile
méthodes
    Nouveau
        | pile |
        pile ← super Nouveau.
        pile Initialiser.
        ↑ pile.
```

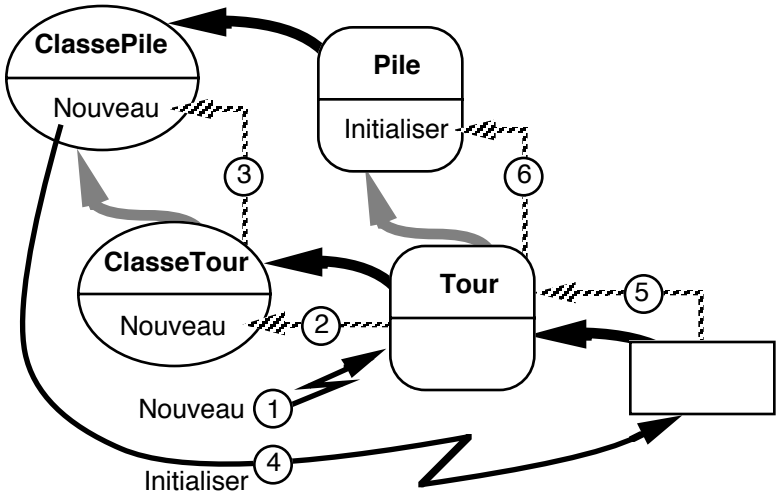

Figure 19 - Héritage des méthodes de classes

Cette méthode est définie dans la métaclasse de *Pile*, puisque c'est la classe *Pile* qui recevra le message *Nouveau*. Cette méthode commence par instancier la pile en s'envoyant le message *Nouveau*. *Nouveau* va donc être envoyé à la métaclasse *ClassePile*, considérée comme instance de sa classe de base *ClasseObjet*. Cela va invoquer la méthode *Nouveau* définie pour tous les objets dans la métaclasse *Classe*. La pile résultante est ensuite initialisée : le message *Initialiser* est envoyé à la pile, c'est donc la méthode *Initialiser* que nous avons écrite dans la classe *Pile* qui va être invoquée. Enfin la pile initialisée est retournée. On aurait pu condenser le corps de la méthode en :

```
↑ super Nouveau Initialiser.
```

Dans le corps de cette méthode, on n'a pas accès aux champs de l'objet *pile*. Il est donc impossible d'initialiser ces champs autrement que par l'envoi d'un message à la pile.

D'autre part, comme on a redéfini la méthode *Nouveau*, toute classe qui hérite de *Pile* utilisera également cette méthode redéfinie, grâce au parallélisme entre l'héritage des classes et

celui des métaclasses. Par exemple, l'instanciation de la classe *HPile* assurera son initialisation, comme le montre la figure 19.

Les métaclasses permettent également de définir des méthodes d'instanciation avec paramètres. Dans l'exemple suivant, on définit une méthode d'instanciation qui permet de donner la taille de la pile à sa création :

```
classe              Pile
méthodes
    Initialiser: taille
            pile ← Tableau Nouveau: taille.
            sommet ← 0.

classe              ClassePile
méthodes
    Nouveau: taille
            ↑ super Nouveau Initialiser: taille.
```

On ajoute à la classe *Pile* une méthode *Initialiser:* qui prend la taille de la pile ; cet argument est transmis à la méthode *Nouveau:* de la classe *Tableau*. On définit ensuite la méthode *Nouveau:* dans la métaclasse de *Pile*, qui instancie une pile et l'initialise avec la taille donnée. Cette méthode *Nouveau:* est un message binaire qu'il ne faut pas confondre avec le message unaire *Nouveau* que nous avons utilisé jusqu'à présent. Une fois définie cette méthode de classe, l'utilisation d'une pile devient :

```
pile ← Pile Nouveau: 100.
pile Empiler: 10.
```

Selon le même mécanisme, on pourrait définir une méthode d'instanciation pour la classe *HPile* qui prenne en paramètre la classe des objets de la pile homogène.

Variables de classe

L'autre utilisation des métaclasses concerne la possibilité de définir de nouveaux champs dans une classe. De fait, ces champs sont accessibles par toutes les instances d'une classe

(toute instance connaît sa classe) et jouent le rôle de variables globales d'une classe.

Pour illustrer cela, nous allons créer une classe d'objets dont les instances ont un numéro unique. La métaclasse a un champ qui est incrémenté à chaque instanciation :

```
classe          Démo
superclasse     Objet
champs          numéro
méthodes
    Initialiser: n
            numéro ← n.
    Numéro
            ↑ numéro.

classe          ClasseDémo
champs          nb
méthodes
    Nouveau
            nb ← nb + 1.
            ↑ (super Nouveau) Initialiser: nb.
```

La classe *Démo* contient un champ stockant le numéro unique de l'instance, une méthode d'initialisation, et une méthode qui retourne le numéro de l'objet. On ajoute à la métaclasse de *Démo* une variable de classe *nb*, et on redéfinit la méthode d'instanciation *Nouveau*. Cette méthode incrémente la variable de classe *nb*, puis instancie l'objet et l'initialise avec la valeur de *nb*.

4.6 LES DÉRIVÉS DE SMALLTALK

Smalltalk, influencé par Lisp, est à l'origine d'une famille de langages à objets implémentés au-dessus de Lisp. Il s'avère en effet que l'implémentation des mécanismes des langages à objets en Lisp est relativement aisée, et fournit un terrain d'expérimentation de nouveaux concepts.

Parmi les plus anciennes extensions de Lisp avec des objets, on trouve Flavors, qui a servi à implémenter le système d'exploitation des machines Symbolics au début des années 80. Plus récemment, CLOS (Common Lisp Object System) a repris et étendu le modèle des Flavors dans le but de définir une norme de Lisp à objets. Ceyx et ObjVLisp représentent l'école française : Ceyx a été développé vers 1985 au-dessus de Le_Lisp par Jean-Marie Hullot, et ObjVLisp, un peu plus ancien, est l'objet des travaux de Pierre Cointe à partir de VLisp, un dialecte de Lisp développé par Patrick Greussay à l'Université de Vincennes.

Tous ces langages procèdent de principes similaires : il s'agit d'*extensions* de Lisp, c'est-à-dire que le programmeur utilise des fonctions Lisp pour écrire ses programmes[2]. L'effet n'est pas toujours heureux car le style fonctionnel est assez radicalement différent du style de la programmation par objets.

Ces langages fournissent en général trois fonctions de base : la création d'une nouvelle classe, la création d'une instance, et l'envoi d'un message à un objet. La fonction de création d'une classe permet de spécifier son héritage (simple ou multiple), ses variables d'instances, ses méthodes, et éventuellement sa métaclasse. En général, le système est capable de générer automatiquement des fonctions d'accès aux variables d'instance. En effet, celles-ci sont stockées dans une liste associée à l'atome qui représente l'objet, et elles ne peuvent être accédées autrement que par une fonction. Cet artifice est néanmoins pratiquement transparent pour l'utilisateur, comme le montre l'exemple ci-dessous :

```
(def-classe Pile (Objet)    -- classe, superclasse
   (pile sommet))           -- variables d'instance
```

[2] La suite de cette section nécessite en conséquence quelques notions élémentaires de Lisp. Nous espérons cependant ne pas trop ennuyer le lecteur peu familier avec Lisp en limitant les exemples.

```
(def-méthode
    (Empiler Pile) (objet)   -- méthode, classe, arguments
    (setq sommet (+ sommet 1))
    (envoi 'mettre pile sommet objet))
```

Dans cet exemple, dont la syntaxe est inspirée de Flavors, *def-classe* définit une nouvelle classe, et *def-méthode* une nouvelle méthode dans une classe existante. Le corps de la méthode *Empiler* est constitué de deux expressions : l'affectation (*setq* en Lisp) du champ *sommet* et l'envoi du message *mettre* au champ *pile*. Ce message est supposé stocker l'objet passé en second argument à l'indice passé en premier argument. L'envoi de message est une fonction qui s'invoque de la manière suivante :

```
(envoi 'message receveur arg1 arg2 … argn)
```

On peut retrouver une syntaxe plus proche de celle de Smalltalk en transformant chaque objet en une fonction, ce que font certains langages. L'envoi de message s'écrit alors :

```
(receveur message arg1 arg2 … argn)
```

La création d'une pile et l'empilement d'un élément se font de la manière suivante :

```
(setq mapile (instancier 'Pile))
(envoi 'Initialiser mapile)
(envoi 'Empiler mapile 10)
```

La fonction *instancier* réalise l'instanciation d'une classe. La méthode *Initialiser* est définie comme suit :

```
(def-méthode
    (Initialiser Pile) ()        -- méthode, classe, arguments
    (setq pile (instancier 'Tableau))
    (setq sommet 0))
```

L'intérêt d'utiliser Lisp comme langage de base est de disposer d'un environnement déjà important qui simplifie l'implémentation des mécanismes d'objets. La souplesse de Lisp permet également d'expérimenter facilement de nouvelles fonctionnalités et des extensions au modèle général des objets.

Les démons des Flavors

Ainsi les Flavors fournissent des mécanismes sophistiqués pour l'héritage multiple. On a vu que l'héritage multiple provoquait des conflits de noms lorsque plusieurs classes de base définissent une méthode de même nom. Les Flavors permettent de déterminer, pour chaque classe, l'ordre de parcours du graphe des superclasses pour déterminer la bonne méthode à invoquer. Il est également possible de *combiner* l'ensemble des méthodes de même nom héritées, c'est-à-dire de les invoquer l'une après l'autre. Enfin, on peut définir des *démons*, qui sont des méthodes spéciales associées aux méthodes ordinaires. Lorsque la méthode ordinaire est invoquée, tous les démons qui lui sont associés dans la classe ou dans ses superclasses sont également automatiquement invoqués, selon un ordre que le programmeur peut contrôler. Par exemple, pour tracer les invocations du message *Empiler*, il suffit d'ajouter le démon suivant :

```
(def-démon
    (Empiler Pile) (objet)   -- méthode, classe, arguments
    (print "on Empile " objet))
```

Même si l'on redéfinit *Empiler* dans une sous-classe de *Pile*, le démon sera appelé. Dans la pratique, la combinaison de méthodes et les démons sont des mécanismes extrêmement puissants, qui sont par là même difficile à maîtriser : l'envoi d'un message à un objet peut déclencher une quantité d'effets dont l'origine risque d'être difficile à identifier. De la même façon, une modification locale du système, comme l'ajout ou le retrait d'un démon, peut provoquer des effets rapidement incontrôlables. Les programmeurs Lisp sont coutumiers de ce genre de phénomènes, et trouveront dans ces langages un terrain d'expérimentation encore plus vaste.

Les métaclasses d'ObjVLisp

Nous terminerons cette revue des dérivés de Lisp par ObjVLisp et son modèle de métaclasses. En effet, ce langage

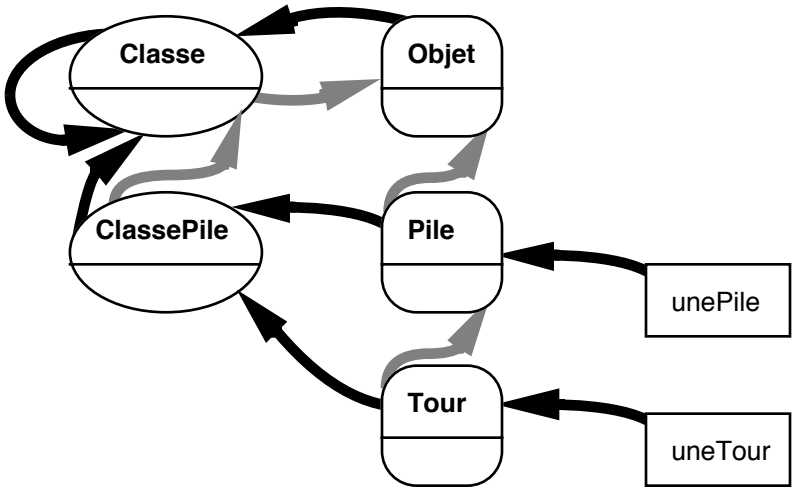

Figure 20 - Le modèle de métaclasses de ObjVLisp

offre ce que l'on peut considérer comme le modèle à la fois le plus simple et le plus ouvert de métaclasses (figure 20).

Les classes *Classe* et *Objet* sont les deux classes primitives du système. *Objet* est une instance de *Classe*, et *Classe* est une sous-classe de *Objet*. *Objet* n'a pas de superclasse, alors que *Classe* est sa propre instance. *Objet* est la racine de l'arbre d'héritage, tandis que *Classe* est la racine de l'arbre d'instanciation. Toute classe doit donc hériter indirectement de *Objet*, et être l'instance de *Classe* ou d'une classe dérivée de *Classe*. Ce dernier point correspond à la création de métaclasses, sans les contraintes imposées par Smalltalk-80. Dans la figure 20, *Pile* et *Tour* ont la même métaclasse alors qu'avec Smalltalk-80, chacune d'elles aurait sa propre métaclasse. Dans cet exemple, il est utile de n'avoir qu'une métaclasse car la même méthode d'instanciation convient aux deux classes. D'un autre côté, le modèle de Smalltalk-80 permet de rendre les classes transparentes à l'utilisateur grâce à la bijection entre classes et métaclasses, ce qui n'est pas le cas d'ObjVLisp.

4.7 CONCLUSION

Smalltalk est sans conteste le langage qui est à l'origine du succès du modèle des objets. Très rapidement, le langage a été validé par la réalisation d'applications importantes, en particulier l'environnement de programmation Smalltalk. De leur côté, les extensions de Lisp par les objets ont permis d'expérimenter et de mieux comprendre les mécanismes des langages à objets.

La puissance de Smalltalk est aussi sa principale faiblesse : avec la liaison dynamique et l'absence de typage statique, il est impossible de s'assurer de la correction d'un programme avant son exécution, ni même après. De plus, l'absence de mécanisme de protection des accès (toute méthode est accessible par tout le monde) n'ajoute pas à la sécurité de programmation. Des extensions de Smalltalk ont introduit avec succès la déclaration des classes des arguments et des variables locales. Dans CLOS, les types des arguments des méthodes doivent être déclarés. Dans les deux cas, la perte de fonctionnalité est minime, et, dans CLOS, le typage permet même d'introduire des méthodes génériques et d'augmenter ainsi la puissance du langage.

L'autre faiblesse de Smalltalk et des langages interprétés en général est le manque d'efficacité à l'exécution. Là encore, de nombreux travaux ont permis d'augmenter les performances de façon spectaculaire. Des mesures ont même montré que, pour certaines applications, la différence de performance entre Smalltalk et un langage à objets typé et compilé n'était pas significative. Dans d'autres cas, la différence est rédhibitoire, ce qui ne fait que confirmer qu'il n'existe pas de langage universel. Smalltalk en tout cas reste un langage privilégié pour découvrir les concepts des langages à objets, mais aussi pour développer des prototypes et, dans certains cas, des applications finales.