OFL : les relations et descriptions d'*Eiffel* et de *Java*

Pierre Crescenzo
http://www.crescenzo.nom.fr/

Table des matières

1 Introduction

Ce rapport présente une étude des relations (telles l'héritage) et des descriptions (telles la classe) des langages de programmation *Eiffel* [Mey94] et *Java* [Fla99, AG00]. Cette étude a été réalisée dans le cadre de la conception du modèle *OFL* (*Open Flexible Languages*) [CCCL01, CCL01] qui décrit les langages à objets à classes en mettant l'accent sur la sémantique opérationnelle des descriptions et relations entre descriptions.

Ce document peut être utilisé comme un modeste manuel de référence pour découvrir les capacités, limitations, ressemblances et différences d'*Eiffel* et de *Java* du point de vue de leurs relations et descriptions.

2 Les relations

Dans la suite, un *type de relation* est nommé *concept-relation*.

2.1 Présentation

Pour chaque relation, nous avons défini les notions de :

description-source Il s'agit de la description qui déclare la relation.

description-cible C'est une description qui définit et fournit le service (au sens très général du terme) demandé, par l'intermédiaire de la relation, par la description-source.

Par exemple, dans la relation implements de *Java*, la description-source est la classe qui implante un ensemble de descriptions-cibles constitué par les interfaces implémentées. Pour l'agrégation, la description-source est la classe qui déclare un attribut et la description-cible celle qui décrit le type de celui-ci. De manière générale, une description cliente d'un service est la source de la description (cible) qui fournit celui-ci.

Pour chacun des concepts-relations nous donnerons, sauf dans les cas trop atypiques, les propriétés suivantes :

- La relation constitue-t-elle une importation ou une utilisation de la description-cible par la description-source ?
- La relation est-elle simple ou multiple (dans ce cas, elle peut admettre plusieurs descriptions-cibles) ?
- Le relation peut-elle être répétée (dans ce cas, elle peut admettre que la même description apparaisse plusieurs fois comme description-cible) ? Si elle est simple, elle est forcément non répétée.
- La relation est-elle obligatoirement linéaire ou admet-elle des cycles ?
- Les primitives de la description-cible sont-elles accessibles directement (comme si elles étaient locales) ou indirectement (par nommage explicite de la description-cible ou de l'une de ses instances) par la description-source ? Cette question est illustrée sur la figure 1 page 3.
- Le polymorphisme est-il permis pour cette relation ? Cette question n'a de sens que dans le cas d'une relation d'importation.
- Toujours exclusivement pour les relations d'importation, quel type de variance (covariance, contravariance, ...) est acceptée ?
- Pour les relations d'utilisation seulement, l'objet utilisé est-il dépendant ou indépendant de celui qui l'utilise. On parle d'objet dépendant quand la *vie* de l'utilisé est conditionnée par celle de l'utilisant. C'est par exemple la cas quand l'objet utilisé, plutôt qu'une référence, est une simple valeur.
- On se pose ensuite la question de savoir si l'objet de la description-cible est utilisé systématiquement, au travers de la relation, par tous les objets de la description-source (notion d'*attribut de description*) ou par un ou plusieurs objets de la description-source (*attribut d'instance*) ou encore par un unique objet de la description-source (comme les attributs expansés d'*Eiffel*). Cette question ne prend de sens, comme la précédente, que dans le cas d'une relation d'utilisation.
- Les deux questions suivantes, spécifiques aux relations d'utilisation elles-aussi, concernent la possibilité d'accéder directement aux attributs ou le fait de devoir obligatoirement passer par un accesseur. La question

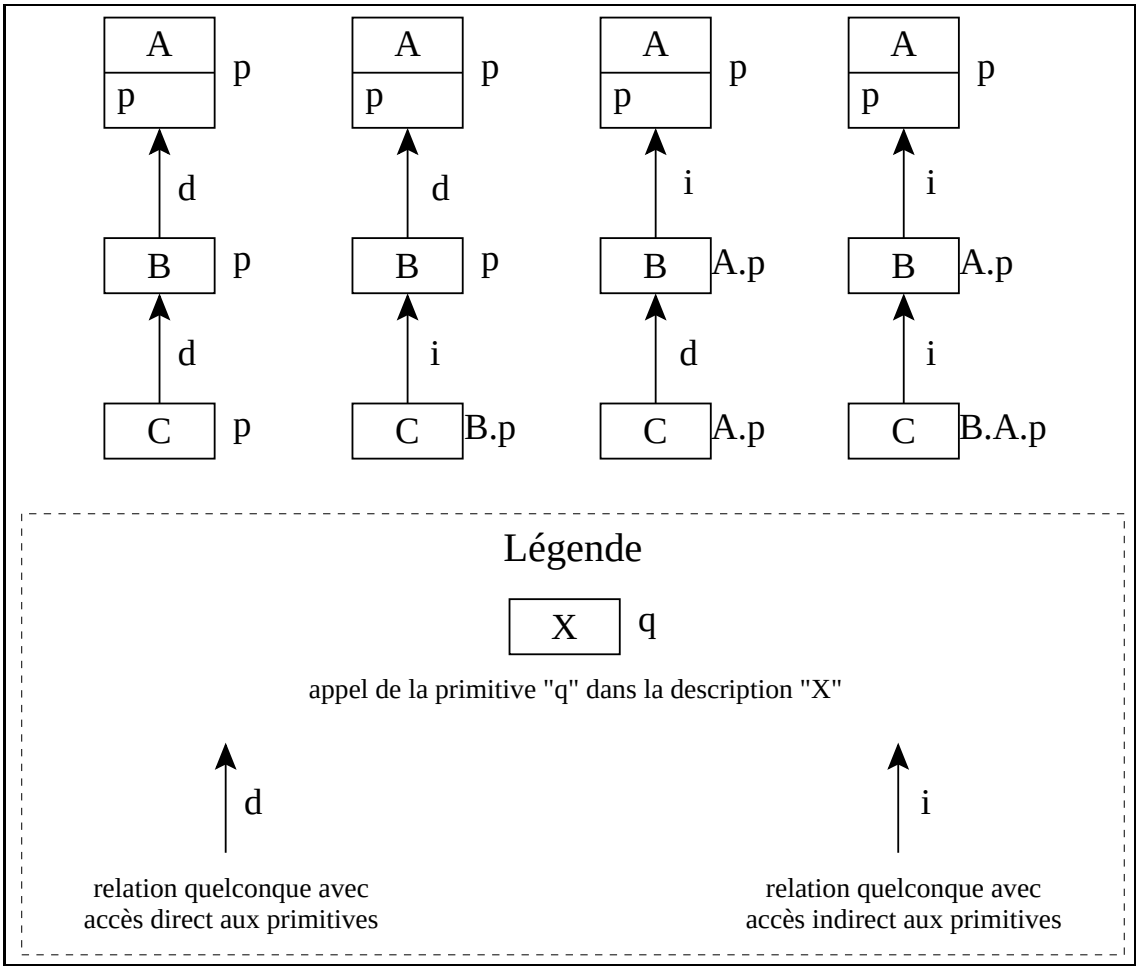

FIG. 1 – Accès direct ou indirect

se pose en lecture puis en écriture (modification).
- Il est ensuite question de symétrie. La relation est-elle ou non symétrique ? C'est-à-dire fournit elle un lien identique de la source vers la cible et de la cible vers la source ?
- Nous cherchons ensuite à reconnaître une éventuelle relation opposée.
- Nous voulons ensuite déterminer quelles sont les opérations permises par le concept-relation parmi :
  - supprimer une primitive,
  - renommer une primitive,
  - redéfinir une primitive,
  - masquer une primitive,
  - montrer (*dé*masquer) une primitive,
  - rendre une primitive abstraite et
  - rendre une primitive concrète.
- Enfin, nous donnons la liste des concepts-descriptions (que nous décrivons section 3.2 page 13) qui peuvent servir de source à ce concept-relation, puis celle de ceux qui peuvent être utilisés comme cible.

2.2 Les concepts-relations d'*Eiffel*

Le langage *Eiffel* définit plusieurs concepts-relations inter-descriptions : un concept-relation d'importation : l'héritage et quatre concepts-relations d'utilisation : la clientèle, la clientèle expansée, la généricité et la généricité expansée.

héritage (mot-clé inherit) Prenons un exemple dans lequel une classe S (pour *Source*) hérite d'une classe C (pour *Cible*), S et C pouvant être concrètes ou abstraites (avec, dans ce cas, les restrictions d'usage sur la gestion des instances propres). Si nous essayons de résumer les caractéristiques de l'*héritage à la Eiffel*, nous pouvons dire que ce concept-relation est :
- importation,
- multiple (S peut hériter d'autres classes),
- répété (S peut hériter plusieurs fois, directement ou indirectement, de C),
- linéaire (C ne peut hériter de S, ni directement ni indirectement),
- accès directs obligatoires (conséquence de l'importation des primitives) et indirects interdits (il n'existe aucun moyen d'appeler une primitive de la classe-cible par l'intermédiaire de la relation d'héritage),
- polymorphique (chaque instance de S est aussi une instance de C et peut-être utilisée comme telle),
- covariant (dans S, tous les attributs, résultats de fonction et paramètres de méthode[1] doivent être d'un type conforme [et de même nombre, pour les paramètres] à leur homologue dans C),
- asymétrique (« S hérite de C » n'implique pas « C hérite de S »),

[1]Désormais, pour simplifier la lecture, sans que cela nuise cependant à la généralité, nous parlerons souvent des attributs seulement et négligerons les résultats de fonction, paramètres de méthode, ...

- sans opposé (l'héritage permet de spécifier, comme description-cible, une classe ascendante mais aucune relation n'existe pour faire l'inverse : déclarer une descendante),
- le renommage, la redéfinition, l'abstraction et la concrétisation sont autorisés (sous certaines conditions, par exemple pour la redéfinition : même nombre de paramètres, conformité des types de paramètres, ...), la suppression, le masquage et le démasquage sont interdits,
- concepts-descriptions applicables en tant que :

  source directe : classe (cf. page 13), classe abstraite (cf. page 14), classe générique (cf. page 14), classe abstraite générique (cf. page 15), classe expansée (cf. page 15) et classe expansée générique (cf. page 15) ;

  cible directe : classe, classe abstraite, classe générique, classe abstraite générique, classe expansée, classe expansée générique et type fondamental (cf. page 16).

clientèle Imaginons une classe S qui possède un attribut (référencé) de classe C. Cette relation possède les caractéristiques suivantes (le polymorphisme et la variance n'ont aucun sens pour les concepts-relations d'utilisation, au contraire de la dépendance et du partage qui leur sont spécifiques). Il est :
- utilisation,
- multiple (S peut avoir d'autres attributs)[2],
- répété (S peut avoir d'autres attributs de classe C),
- circulaire (directement : S peut posséder un attribut de classe S, ou indirectement : C peut posséder un attribut de classe S),
- accès directs interdits (il est indispensable de passer par l'attribut pour appeler une de ses primitives) et indirects obligatoires (pour toutes les primitives non masquées, l'accès se fait par l'intermédiaire de l'attribut),
- indépendant (l'objet de classe C attaché à l'attribut a une durée de vie indépendante de l'objet de classe S qui l'utilise),
- attribut d'instance (plusieurs instances de S peuvent référencer la même instance C)[3],
- attributs directement accessibles en lecture (des limitations peuvent être introduites par les clauses d'exportation),
- accesseurs obligatoires pour la modification des attributs (d'autres limitations peuvent être introduites par les clauses d'exportation),
- asymétrique (« S est cliente de C » n'implique pas « C est cliente de S »),
- sans opposé (aucune relation ne permet à une classe de choisir d'avoir un attribut dans une autre),

[2]On peut considérer soit que la relation de clientèle est multiple, soit qu'elle peut être composée avec elle-même, ce qui revient au même.

[3]En utilisant une fonction once, il est possible de créer l'équivalent d'un attribut de description.

- le masquage et le démasquage sont autorisés, la suppression, le renommage, la redéfinition, l'abstraction et la concrétisation sont interdits,
- concepts-descriptions applicables en tant que :

  source directe : classe (cf. page 13), classe abstraite (cf. page 14), classe générique (cf. page 14), classe abstraite générique (cf. page 15), classe expansée (cf. page 15) et classe expansée générique (cf. page 15) ;

  cible directe : classe, classe abstraite, classe générique, classe abstraite générique et néant (cf. page 17).

clientèle expansée Prenons maintenant une classe S qui possède un attribut expansé (mot-clé expanded) de classe C. Cette nouvelle forme de clientèle est définie par les propriétés suivantes :
- utilisation,
- multiple,
- répété,
- linéaire (C ne peut posséder un attribut expansé de classe S),
- accès directs interdits et indirects obligatoires (sauf masquage),
- dépendant (l'objet de classe C attaché à l'attribut expansé a une durée de vie dépendante de l'objet de classe S qui l'utilise),
- attribut d'instance unique (plusieurs instances de S ne peuvent pas référencer la même instance de C)[4],
- asymétrique,
- attributs directement accessibles en lecture,
- accesseurs obligatoires pour la modification des attributs,
- sans opposé,
- le masquage et le démasquage sont autorisés, la suppression, le renommage, la redéfinition, l'abstraction et la concrétisation sont interdits,
- concepts-descriptions applicables en tant que :

  source directe : classe (cf. page 13), classe abstraite (cf. page 14), classe générique (cf. page 14), classe abstraite générique (cf. page 15), classe expansée (cf. page 15) et classe expansée générique (cf. page 15) ;

  cible directe : classe, classe générique, classe expansée, classe expansée générique, type fondamental (cf. page 16) et type binaire (cf. page 16).

généricité Voyons comment peut être décrit la relation entre une classe S générique dont un paramètre de généricité est de classe C : C n'est connue qu'à l'exécution. Ce concept-relation est :
- utilisation,
- multiple (S peut avoir plusieurs paramètres de généricité),
- répété (S peut avoir d'autres paramètres de généricité de classe C),
- circulaire,

[4]Une fonction once permet de gérer un attribut de description unique.

- accès directs interdits et indirects obligatoires (sauf masquage),
- indépendant,
- attribut d'instance[5],
- asymétrique,
- attributs directement accessibles en lecture,
- accesseurs obligatoires pour la modification des attributs,
- sans opposé,
- le masquage et le démasquage sont autorisés, la suppression, le renommage, la redéfinition, l'abstraction et la concrétisation sont interdits,
- concepts-descriptions applicables en tant que :

  source directe : classe générique (cf. page 14), classe abstraite générique (cf. page 15) et classe expansée générique (cf. page 15) ;

  cible directe : classe (cf. page 13), classe abstraite (cf. page 14), classe générique et classe abstraite générique.

généricité expansée C'est une simple composition de la clientèle expansée (cf. page 6) et de la généricité (cf. page 6). Nous n'en décrivons donc que les parties non communes à ces deux sortes de description :

- linéaire,
- dépendant,
- attribut d'instance unique,
- concepts-descriptions applicables en tant que :

  source directe : classe générique (cf. page 14), classe abstraite générique (cf. page 15) et classe expansée générique (cf. page 15) ;

  cible directe : classe expansée (cf. page 15), classe expansée générique et type fondamental (cf. page 16).

2.3 Les concepts-relations de *Java*

Nous allons maintenant essayer de réaliser le même genre d'étude pour le langage *Java*. Nous trouvons quatre concepts-relations d'importation (héritage inter-classes, héritage inter-interfaces, concrétisation et implémentation) et quatre concepts-relations d'utilisation (agrégation, agrégation de classe, composition et composition de classe).

héritage inter-classes La classe S hérite (mot-clé extends) de la classe C, S et C pouvant être concrètes ou abstraites. Nous considérons ici le cas où les classes sont toutes deux abstraites ou toutes deux concrètes. Lorsqu'une classe concrète hérite d'une classe abstraite, nous appelons cette relation *concrétisation* (cf. page 9). Ce concept-relation est :

- importation,
- simple (S ne peut hériter d'une autre classe),
- non répété (S ne peut hériter plusieurs fois de C),
- linéaire (C ne peut hériter de S, directement ou indirectement),

[5] Le mot *attribut* peut être considéré ici comme un abus de langage.

- accès directs autorisés (sauf pour les constructeurs et les primitive déclarées private) et indirects autorisés (grâce au cas particulier du mot-clé super),
- polymorphique (chaque instance de S est aussi une instance de C est peut-être utilisée comme telle),
- invariant pour la redéfinition (dans S, tous les attributs, résultats de fonction et paramètres de méthode doivent être de même type [et même nombre, pour les paramètres] que leur homologue dans C) et libre pour le remplacement (dans S, tous les attributs et paramètres de méthode doivent être de type [ou de nombre, pour les paramètres] différent de leur homologue dans C),
- asymétrique,
- sans opposé,
- la redéfinition, le masquage (par exemple en déclarant un attribut de la description-source comme private) et la concrétisation sont autorisés (sous certaines conditions, par exemple pour la redéfinition : même nombre de paramètres, même type pour les paramètres, ...), la suppression, le renommage, le démasquage et l'abstraction sont interdits,
- concepts-descriptions applicables en tant que :

  source directe : classe (cf. page 17), classe abstraite (cf. page 18), classe membre statique (cf. page 19), classe membre statique abstraite (cf. page 19), classe membre (cf. page 20), classe membre abstraite (cf. page 20), classe locale (cf. page 20), classe locale abstraite (cf. page 21) et classe anonyme (cf. page 21) ;

  cible directe : classe, classe abstraite, classe membre statique, classe membre statique abstraite, classe membre, classe membre abstraite, classe locale et classe locale abstraite.

héritage inter-interfaces L'interface S hérite (mot-clé extends) de l'interface C. Les caractéristiques de ce concept-relation sont :

- importation,
- multiple,
- répété,
- linéaire,
- accès directs et indirects interdits (il n'y a pas d'instruction, pas de moyen donc d'accéder à quoi que ce soit dans la description-cible),
- polymorphique (chaque instance, forcément indirecte, de S est aussi une instance de C),
- invariant,
- asymétrique,
- sans opposé,
- la redéfinition est autorisée, la suppression, le renommage, le masquage, le démasquage, l'abstraction et la concrétisation sont interdits,
- concepts-descriptions applicables en tant que :

  source directe : interface (cf. page 18) et interface membre statique (cf. page 22) ;

cible directe : interface et interface membre statique.

concrétisation La classe S concrétise[6] (mot-clé extends) la classe abstraite C. Nous avons donc un concept-relation avec les caractéristiques :
- importation,
- simple,
- non répété,
- linéaire,
- accès directs autorisés (sauf pour les constructeurs et les primitive déclarées private) et indirects autorisés (grâce au cas particulier du mot-clé super),
- polymorphique,
- invariant pour la redéfinition et libre pour le remplacement,
- asymétrique,
- sans opposé,
- la redéfinition et le masquage sont autorisés (sous certains conditions), la concrétisation est obligatoire (il faut donner un corps à toutes les méthodes abstraites) et la suppression, le renommage, le démasquage et l'abstraction sont interdits,
- concepts-descriptions applicables en tant que :

  source directe : classe (cf. page 17), classe membre statique (cf. page 19), classe membre (cf. page 20), classe locale (cf. page 20) et classe anonyme (cf. page 21) ;

  cible directe : classe abstraite (cf. page 18), classe membre statique abstraite (cf. page 19), classe membre abstraite (cf. page 20) et classe locale abstraite (cf. page 21).

implémentation La classe S implémente (mot-clé implements) l'interface C ce qui donne les caractéristiques suivantes :
- importation,
- multiple,
- répété (S ne peut implémenter directement qu'une fois C, mais peut avoir, indirectement, plusieurs fois C comme super-interface par l'intermédiaire d'une relation d'héritage),
- linéaire,
- accès directs et indirects interdits (pas de moyen d'accéder à quoi que ce soit dans la description-cible puisqu'il s'agit d'une interface ne contenant que des signatures),
- polymorphique (chaque instance, forcément indirecte, de S est aussi une instance de C),
- invariant,
- asymétrique,
- sans opposé,
- la redéfinition et la concrétisation sont autorisées, la suppression, le

[6] Nous utilisons indifféremment le mot *concrétisation* pour nommer ce concept-relation de *Java* et pour indiquer le fait de donner un corps à une méthode. Les deux idées sont totalement distinctes (bien que liées) mais attention cependant à ne pas confondre.

renommage, le masquage, le démasquage et l'abstraction sont interdits,
- concepts-descriptions applicables en tant que :

  source directe : classe (cf. page 17), classe abstraite (cf. page 18), classe membre statique (cf. page 19), classe membre statique abstraite (cf. page 19), classe membre (cf. page 20), classe membre abstraite (cf. page 20), classe locale (cf. page 20), classe locale abstraite (cf. page 21) et classe anonyme (cf. page 21) ;

  cible directe : interface (cf. page 18) et interface membre statique (cf. page 22).

agrégation Il s'agit ici d'un concept-relation d'utilisation et non plus d'importation, ce concept-relation est :
- utilisation,
- multiple,
- répété,
- circulaire,
- accès directs interdits et indirects autorisés (pour les primitives déclarées public),
- indépendant,
- attribut d'instance,
- attributs directement accessibles en lecture (des limitations d'espace de nommage peuvent être introduites par les qualifieurs d'accès),
- attributs directement accessibles en écriture (des limitations d'espace de nommage peuvent être introduites par les qualifieurs d'accès),
- asymétrique,
- sans opposé,
- la suppression (en déclarant une primitive private par exemple) est autorisée, le renommage, la redéfinition, le masquage, le démasquage, l'abstraction et la concrétisation sont interdits,
- concepts-descriptions applicables en tant que :

  source directe : classe (cf. page 17), classe abstraite (cf. page 18), interface (cf. page 18), classe membre statique (cf. page 19), classe membre statique abstraite (cf. page 19), classe membre (cf. page 20), classe membre abstraite (cf. page 20), classe locale (cf. page 20), classe locale abstraite (cf. page 21), classe anonyme (cf. page 21) et interface membre statique (cf. page 22)[7] ;

  cible directe : classe, classe abstraite, interface, classe membre statique, classe membre statique abstraite, classe membre, classe membre abstraite, classe locale, classe locale abstraite[8], classe anonyme[9], interface membre statique et tableau (cf. page 22).

[7] Les interfaces et interfaces membres statiques ne peuvent pas déclarer d'attribut d'instance ni de variable locale. Cependant, l'agrégation est possible par les paramètres de méthodes et résultat de fonctions.

[8] La déclaration d'une variable locale de type d'une classe locale ou locale abstraite s'apparente en effet à une agrégation.

[9] L'usage, même éphémère, d'une instance de classe anonyme est aussi une sorte d'agréga-

agrégation de classe Dans ce cas, le qualifieur d'accès static est précisé. Cela signifie que l'attribut de classe C est associé à la classe S plutôt qu'à ses instances. Ce nouveau concept-relation est :
- utilisation,
- multiple,
- répété,
- circulaire,
- accès directs interdits et indirects autorisés (pour les primitives déclarées public),
- indépendant,
- attribut de description[10],
- attributs directement accessibles en lecture,
- attributs directement accessibles en écriture,
- asymétrique,
- sans opposé,
- la suppression (en déclarant un attribut private par exemple) est autorisée, le renommage, la redéfinition, le masquage, le démasquage, l'abstraction et la concrétisation sont interdits,
- concepts-descriptions applicables en tant que :

  source directe : classe (cf. page 17), classe abstraite (cf. page 18), interface (cf. page 18), classe membre statique (cf. page 19), classe membre statique abstraite (cf. page 19), classe membre (cf. page 20), classe membre abstraite (cf. page 20), classe locale (cf. page 20), classe locale abstraite (cf. page 21), classe anonyme (cf. page 21) et interface membre statique (cf. page 22)[11] ;

  cible directe : classe, classe abstraite, interface, classe membre statique, classe membre statique abstraite, classe membre, classe membre abstraite, classe locale, classe locale abstraite[12], classe anonyme[13], interface membre statique et tableau (cf. page 22).

composition Ce concept-relation décrit l'utilisation d'attribut de type primitif (exemples : entiers, booléens) dans la classe source. Les caractéristiques d'un tel concept-relation peuvent être décrites par :
- utilisation,
- multiple,
- répété,
- linéaire,
- accès directs et indirects interdits (le type primitif qui sert de cible n'a aucune primitive, ce n'est pas une classe),

tion.

[10] Cette capacité à être partagé se rapproche un peu des fonctions once d'*Eiffel* mais diffère d'elles par le fait que la valeur partagée est variable.

[11] Les interfaces, classes membres, classes membres abstraites, classes locales, classes locales abstraites, classes anonymes et interfaces membres statiques peuvent déclarer une agrégation de classe pour une constante.

[12] Il est possible de faire une agrégation de classe pour une constante d'une classe locale ou locale abstraite dans une autre classe locale ou locale abstraite.

[13] Une instance de classe anonyme peut être utilisée pour initialiser une variable de classe.

- dépendant,
- attribut d'instance,
- pas d'attribut, donc pas de problème d'accessibilité,
- asymétrique,
- sans opposé,
- la suppression est autorisée, le renommage, la redéfinition, le masquage, le démasquage, l'abstraction et la concrétisation sont interdits,
- concepts-descriptions applicables en tant que :

  source directe : classe (cf. page 17), classe abstraite (cf. page 18), interface (cf. page 18), classe membre statique (cf. page 19), classe membre statique abstraite (cf. page 19), classe membre (cf. page 20), classe membre abstraite (cf. page 20), classe locale (cf. page 20), classe locale abstraite (cf. page 21), classe anonyme (cf. page 21) et interface membre statique (cf. page 22)[14] ;

  cible directe : type primitif (cf. page 23).

composition de classe Elle peut être vue comme un mélange d'agrégation de classe (cf. page 11) et de composition (cf. page 11). En voici les caractéristiques marquantes :

- linéaire,
- accès directs et indirects interdits,
- dépendant,
- attribut de description,
- pas d'attribut, donc pas de problème d'accessibilité,
- concepts-descriptions applicables en tant que :

  source directe : classe (cf. page 17), classe abstraite (cf. page 18), interface (cf. page 18), classe membre statique (cf. page 19), classe membre statique abstraite (cf. page 19), classe membre (cf. page 20), classe membre abstraite (cf. page 20), classe locale (cf. page 20), classe locale abstraite (cf. page 21), classe anonyme (cf. page 21) et interface membre statique (cf. page 22)[15] ;

  cible directe : type primitif (cf. page 23).

3 Les descriptions

Nous nommerons désormais *concepts-descriptions* les *types de descriptions*.

[14] Les interfaces et interfaces membres statiques ne peuvent pas déclarer d'attribut d'instance ni de variable locale. Cependant, la composition est possible par les paramètres de méthodes et résultat de fonctions.

[15] Les interfaces, classes membres, classes membres abstraites, classes locales, classes locales abstraites, classes anonymes et interfaces membres statiques peuvent déclarer une agrégation de classe pour une constante.

3.1 Présentation

Nous venons de présenter plusieurs exemples de concepts-relations tirés des langages *Eiffel* et *Java*. Nous avons, à cette occasion, vu que ces concepts-relations s'appliquent souvent entre des *classes*, mais pas toujours. Le concept-relation d'implémentation de *Java* (cf. section 2.3 page 9), s'il a pour source une classe, gère un ensemble d'*interfaces* comme cibles. Nous appelons ces différentes sortes d'entité des *concepts-descriptions*.

Comme pour les concepts-relations, nous donnons ci-dessous un ensemble des concepts-descriptions d'*Eiffel* puis de *Java*. Voici les points que nous avons considérés comme essentiels dans la présentation de ces concepts-descriptions :

- S'agit-il d'un concept-description simple c'est-à-dire associé à un seul type, ou multiple, associé à plusieurs types[16]. ?
- Ce concept-description a-t-il la capacité de créer des instances propres ?
- A-t-il la possibilité des les détruire ?
- Les primitives sont-elles encapsulées dans les descriptions ? Rappelons que tous les concepts de classe ne sont pas encapsulant. Par exemple, *CLOS* n'encapsule pas les méthodes.
- Les attributs et/ou les méthodes sont-ils permis ? d'instance unique, d'instance ou de description ou plusieurs de ces possibilités ?
- La description permet-elle la surcharge, c'est-à-dire la capacité de posséder plusieurs méthodes de même nom différent par le nom et/ou le type de leurs paramètres ?
- Quelle est la visibilité de cette description ? C'est-à-dire à partir d'où peut-on y accéder ?
- Enfin, pour chaque concept-description, nous associons une liste des concepts-relations qui peuvent être utilisés en prenant le concept-description comme source. Une seconde liste réalise la même chose pour les cibles.

3.2 Les concepts-descriptions d'*Eiffel*

Nous décrivons dans cette partie neuf concepts-descriptions pour *Eiffel*. Nous nous attacherons tout d'abord à préciser les caractéristiques des classes puis nous établirons des distinctions selon qu'elles sont abstraites, génériques ou expansées. Nous nous intéresserons ensuite à des cas particuliers : les types fondamentaux, les types binaires et néant.

classe C'est le concept-description essentiel d'*Eiffel*. Il est possible de le voir comme une description de la structure (les attributs) et du comportement (les méthodes) de ses instances. Il peut également être vu comme un conteneur de celles-ci, un peu particulier il est vrai, puisqu'il ne peut fournir son contenu. Imaginons donc, pour ce premier exemple, une classe C (non générique, non abstraite et non expansée). Nous pouvons décrire le concept-description correspondant de la manière suivante :

[16] Notion de type générique ou de *template*.

- simple (C est associé à un seul type),
- générateur (C se charge de la création de ses instances propres),
- non destructeur (C ne peut détruire ses instances. *Eiffel*, comme d'autres langages à objets possède un ramasse-miettes qui se charge de libérer la mémoire utilisée par les objets qui ne sont plus référencés. Il est donc inutile, et ce serait d'ailleurs dangereux, de permettre au programmeur de détruire des instances. D'autres langages, comme *C++* ne dispose pas de ramasse-miettes généraux et la destruction doit donc parfois être faite explicitement par le programmeur, à ses risques et périls.),
- encapsulant (tout attribut et toute méthode associée aux instances de C est décrite dans C),
- attributs et méthodes autorisés,
- autorise les attributs d'instance unique (expansé) ou d'instance,
- surcharge interdite,
- global (la classe est directement accessible par toute autre classe),
- concepts-relations valides en tant que :

  source directe : héritage (cf. page 4), clientèle (cf. page 5) et clientèle expansée (cf. page 6) ;

  cible directe : héritage, clientèle, clientèle expansée et généricité (cf. page 6).

classe abstraite Les classes abstraites (mot-clé deferred) constituent un premier cas particulier de classe. Elles ne sont pas complètement spécifiées et sont dites *différées* ou *retardées* car leur spécification doit être précisée (concrétisée) par une autre classe. Cette incomplétude ne leur permet pas de créer des instances ; elles possèdent cependant les instances indirectes de leurs héritières non abstraites. Soit un classe abstraite C (non générique et non expansée), le concept-description classe abstraite est :

- simple,
- non générateur (C ne peut créer d'instance propre),
- non destructeur,
- encapsulant,
- attributs et méthodes (il faut au moins une méthode abstraite dans C ou dans une des ses classes héritées) autorisés,
- autorise les attributs d'instance unique ou d'instance,
- surcharge interdite,
- global,
- concepts-relations valides en tant que :

  source directe : héritage (cf. page 4), clientèle (cf. page 5) et clientèle expansée (cf. page 6) ;

  cible directe : héritage, clientèle et généricité (cf. page 6).

classe générique Une des capacités très utiles d'*Eiffel* est la possibilité de déclarer des classes génériques. Voyons donc maintenant comment nous pouvons décrire une classe générique C (non abstraite et non expansée).

- multiple (C est associé à autant de types qu'il y a de combinaisons différentes des types des paramètres génériques effectifs lors de l'exécution de l'application),
- générateur,
- non destructeur,
- encapsulant,
- attributs et méthodes autorisés,
- autorise les attributs d'instance unique ou d'instance,
- surcharge interdite,
- global,
- concepts-relations valides en tant que :

  source directe : tous.

  cible directe : héritage (cf. page 4), clientèle (cf. page 5), clientèle expansée (cf. page 6) et généricité (cf. page 6).

classe abstraite générique Il s'agit d'une composition de classe abstraite (cf. page 14) et de classe générique (cf. page 14). Voici les caractéristiques particulières à cette sorte de description, nous oublierons tout ce qui est commun à classe abstraite et à classe générique :

- multiple,
- non générateur,
- attributs et méthodes (au moins une méthode abstraite) autorisés,
- concepts-relations valides en tant que :

  source directe : tous.

  cible directe : héritage (cf. page 4), clientèle (cf. page 5) et généricité (cf. page 6).

classe expansée Le dernier cas particulier de classe est constitué par les classes expansées (mot-clé expanded). Elles permettent notamment de décrire élégamment les types de base d'une application. Décrivons donc une classe expansée C.

- simple,
- générateur,
- non destructeur,
- encapsulant,
- attributs et méthodes autorisés,
- n'autorise que les attributs d'instance unique,
- surcharge interdite,
- global,
- concepts-relations valides en tant que :

  source directe : héritage (cf. page 4), clientèle (cf. page 5) et clientèle expansée (cf. page 6) ;

  cible directe : héritage, clientèle expansée et généricité expansée (cf. page 7).

classe expansée générique Elles possèdent les caractéristiques à la fois des classes expansée (cf. page 15) et des classes génériques (cf. page 14) :

- simple,

- n'autorise que les attributs d'instance unique,
- concepts-relations valides en tant que :

  source directe : tous.

  cible directe : héritage (cf. page 4), clientèle expansée (cf. page 6) et généricité expansée (cf. page 7).

type fondamental En *Eiffel*, certains types sont considérés comme fondamentaux. Ils représentent les entités de base de l'application. Les types fondamentaux (également appelés classes fondamentales) permettent de manipuler les valeurs les plus couramment utilisées : nous y trouvons ainsi les booléens, les caractères, les flottants[17] (simples et doubles) et les pointeurs. Ces classes peuvent être considérées comme normales à ceci près :

- Elles disposent d'une interface syntaxique particulière : des mots-clés leur sont associés (exemples : `true`) et surtout, il est possible d'utiliser directement des constantes de leur type (exemples : `12` est reconnu comme un entier).
- Chacune est expansée et hérite d'une classe non expansée sans en modifier autrement l'interface. Il est donc possible d'utiliser plutôt les classes héritées si nous voulons manipuler une valeur fondamentale comme un objet normal (*i. e.* référencé).
- Bien qu'il n'y ait aucune relation d'héritage entre les classes fondamentales, les entiers sont considérés comme[18] des flottants simples et les flottants simples comme des flottants doubles. Il ne s'agit cependant que de respecter une convention humaine bien pratique qui décrit l'ensemble des réels comme incluant celui des entiers.
- concepts-relations valides en tant que :

  source directe : non applicable (il n'est pas possible de créer un nouveau type fondamental) ;

  cible directe : héritage (cf. page 4), clientèle expansée (cf. page 6) et généricité expansée (cf. page 7).

type binaire Les types binaires sont d'un usage très spécialisé et forment un cas très particulier. Ils permettent de décrire et manipuler des objets de type *suite de n bits*[19]. Leurs particularités sont :

- Aucune classe n'existe pour décrire un objet de type *suite de n bits* avant que celui-ci ne soit déclaré. Après sa déclaration, tout se passe cependant comme si une telle classe existait et un certain nombre de primitives est donc disponible.
- Bien qu'il soit possible de s'en servir comme si elles existaient, ces classes ne sont que virtuelles et ne font donc pas partie de l'arbre d'héritage. Il est néanmoins considéré qu'elles descendent directement de la classe racine.

[17] Les flottants sont des approximations de réels.
[18] « considéré comme » se dit « conforme à » dans la terminologie *Eiffel*.
[19] n ne peut varier au cours du temps pour un objet donné.

- Bien entendu, ces classes, qui n'existent pas, n'héritent de rien sauf de la classe racine ANY. Cependant, dans un souci de les rendre pratiques à utiliser, il est possible de considérer tout objet de type *suite de n bits* comme un objet quelconque (de la classe racine donc) ou comme un objet de type *suite de m bits*, à la condition *sine qua none* que $n \leq m$.
- concepts-relations valides en tant que :

  source directe : non applicable (il n'est pas possible de créer un nouveau type binaire) ;

  cible directe : clientèle expansée (cf. page 6).

néant Il n'est pas possible de terminer cette liste sans citer le cas très particulier de la classe NONE qui est une classe fictive utile pour le graphe de type. Elle hérite automatiquement (et en ignorant tous les conflits) de toutes les classes de l'application (donc, à cause de la non-circularité, aucune classe ne peut en hériter) et n'exporte aucune primitive. De plus, elle ne possède qu'une seule instance, nommée Void, qui sert de valeur par défaut à tout objet non initialisé.
- concepts-relations valides en tant que :

  source directe : non applicable (il n'est pas possible de créer un nouveau néant) ;

  cible directe : clientèle (cf. page 5).

Enfin, précisons que les classes représentant les tableaux et les chaînes de caractères constituent des cas particuliers en *Eiffel*, mais cela n'est vrai que du point de vue syntaxique : des facilités sont offertes au programmeur pour exprimer des constantes de ces types. Du point de vue sémantique, ces classes sont standards et ne sont pas traitées de manière particulière par le compilateur.

3.3 Les concepts-descriptions de *Java*

Nous présentons maintenant dans cette partie treize concepts-descriptions pour *Java* comme nous l'avons fait dans la section précédente pour *Eiffel*.

classe C'est un concept-description proche de celui d'*Eiffel*. Prenons comme premier exemple une classe C. Nous pouvons décrire le concept-description correspondant de la manière suivante :
- simple,
- générateur,
- non destructeur (présence d'un ramasse-miettes),
- encapsulant,
- attributs et méthodes autorisés,
- autorise les attributs d'instance ou de description,
- surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
- local au paquetage[20] (C peut être accessible, si elle est public, depuis

[20] Un paquetage est un ensemble de descriptions réunies par un thème commun.

un autre paquetage grâce à la spécification du nom de son paquetage ou d'une clause d'importation non ambiguë),
- concepts-relations valides en tant que :

  source directe : héritage inter-classes (cf. page 7), concrétisation (cf. page 9), implémentation (cf. page 9), agrégation (cf. page 10), agrégation de classe (cf. page 11), composition (cf. page 11) et composition de classe (cf. page 12) ;

  cible directe : héritage inter-classes, agrégation et agrégation de classe.

classe abstraite Il s'agit de classe qui peuvent contenir en même temps des méthodes abstraites et des méthodes concrètes. Une classe abstraite *Java* (mot-clé *abstract*) peut se décrire ainsi :
- simple,
- non générateur (mais peut posséder des constructeurs que l'on peut activer indirectement par un super(. . .) dans une classe héritière directe concrète),
- non destructeur (présence d'un ramasse-miettes),
- encapsulant,
- attributs et méthodes autorisés,
- autorise les attributs d'instance ou de description,
- surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
- local au paquetage (cf. la remarque correspondante pour les classes *Java* page 17),
- concepts-relations valides en tant que :

  source directe : héritage inter-classes (cf. page 7), implémentation (cf. page 9), agrégation (cf. page 10), agrégation de classe (cf. page 11), composition (cf. page 11) et composition de classe (cf. page 12) ;

  cible directe : héritage inter-classes, concrétisation (cf. page 9), agrégation et agrégation de classe.

interface Les interfaces de *Java* offrent un moyen simple de spécifier un comportement pour un objet sans être, à l'inverse des classes, des conteneurs : une interface ne possède pas d'instance. Le concept-description interface est :
- simple,
- non générateur,
- non destructeur,
- encapsulant,
- attributs interdits mais méthodes (abstraites seulement) autorisées,
- autorise les attributs d'instance ou de description,
- surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
- local au paquetage,
- concepts-relations valides en tant que :

source directe : héritage inter-interfaces (cf. page 8), agrégation (cf. page 10), agrégation de classe (cf. page 11), composition (cf. page 11)[21] et composition de classe (cf. page 12)[22] ;

cible directe : héritage inter-interfaces, implémentation (cf. page 9), agrégation et agrégation de classe.

classe membre statique Les classes membres statiques ont pour principal intérêt d'apparaître dans l'espace de nommage d'une autre description. Voyons donc maintenant comment nous pouvons décrire une classe membre statique C. Notons que, de par son statut, C peut accéder aux attributs et méthodes de classe (static) de sa classe encapsulante.

– simple,
– générateur,
– non destructeur,
– encapsulant,
– attributs et méthodes autorisés,
– autorise les attributs d'instance ou de description,
– surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
– local à la description encapsulante (mais C peut aussi être accessible, si elle n'est pas private, depuis une autre description grâce à la spécification du nom de sa description encapsulante),
– concepts-relations valides en tant que :

source directe : héritage inter-classes (cf. page 7), concrétisation (cf. page 9), implémentation (cf. page 9), agrégation (cf. page 10), agrégation de classe (cf. page 11), composition (cf. page 11) et composition de classe (cf. page 12) ;

cible directe : héritage inter-classes, agrégation et agrégation de classe.

classe membre statique abstraite Les classes membres statiques peuvent elles-aussi être abstraites. Une classe membre statique abstraite est donc, en quelque sorte, une composition d'une classe abstraite (cf. page 18) et d'une classe membre statique (cf. page 19). Pour les caractéristiques qui diffèrent dans ces concepts-relations, voici celles que nous pouvons retenir.

– non générateur,
– local à la description encapsulante,
– concepts-relations valides en tant que :

source directe : héritage inter-classes (cf. page 7), implémentation (cf. page 9), agrégation (cf. page 10), agrégation de classe (cf. page 11), composition (cf. page 11) et composition de classe (cf. page 12) ;

[21] Agrégation et composition sont présentes ici en raison des paramètres de méthode et résultats de fonction.

[22] Les interfaces peuvent déclarer des constantes de classes, d'où la présence ici de l'agrégation de classe et de la composition de classe.

cible directe : héritage inter-classes, concrétisation (cf. page 9), agrégation et agrégation de classe.

classe membre Il est possible de définir en *Java* des classes membres qui sont des sortes de sous-description d'une autre description. Il est à remarquer que chaque instance de C est automatiquement associée à une instance de sa classe encapsulante et a donc accès à tous les attributs et méthodes de celle-ci. Une classe membre C peut être décrite comme suit.
- simple,
- générateur,
- non destructeur,
- encapsulant,
- attributs et méthodes autorisés,
- autorise les attributs d'instance ou de description,
- surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
- local à la description encapsulante,
- concepts-relations valides en tant que :

  source directe : héritage inter-classes (cf. page 7), concrétisation (cf. page 9), implémentation (cf. page 9), agrégation (cf. page 10), agrégation de classe (cf. page 11), composition (cf. page 11) et composition de classe (cf. page 12)[23] ;

  cible directe : héritage inter-classes, agrégation et agrégation de classe.

classe membre abstraite Ce sont en même temps des classes abstraites (cf. page 18) et des classes membres (cf. page 20). Voici la valeur adéquate pour les caractéristiques qui diffèrent dans ces deuxconcepts-relations :
- non générateur,
- local à la description encapsulante,
- concepts-relations valides en tant que :

  source directe : héritage inter-classes (cf. page 7), implémentation (cf. page 9), agrégation (cf. page 10), agrégation de classe (cf. page 11), composition (cf. page 11) et composition de classe (cf. page 12)[24] ;

  cible directe : héritage inter-classes, concrétisation (cf. page 9), agrégation et agrégation de classe.

classe locale Les classes locales sont internes à une méthode (ou d'un initialiseur statique ou non) d'une classe : elles sont ainsi similaires à des variables locales. Nous pouvons présenter ci-dessous les caractéristiques d'une classe locale C.
- simple,
- générateur,

[23] L'agrégation de classe et la composition de classe sont permises pour la définition des constantes.

[24] L'agrégation de classe et la composition de classe sont permises pour la définition des constantes.

- non destructeur,
- encapsulant,
- attributs et méthodes autorisés,
- n'autorise que les attributs d'instance[25],
- surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
- local à la méthode encapsulante,
- concepts-relations valides en tant que :

  source directe : héritage inter-classes (cf. page 7), concrétisation (cf. page 9), implémentation (cf. page 9), agrégation (cf. page 10), agrégation de classe (cf. page 11), composition (cf. page 11) et composition de classe (cf. page 12)[26] ;

  cible directe : héritage inter-classes, agrégation et agrégation de classe[27].

classe locale abstraite Les classes locales abstraites sont, bien évidemment, des classes abstraites (cf. page 18) et des classes locales (cf. page 20). Les points divergents de ces deux concepts-relations trouvent ici les caractéristiques suivante.

- non générateur,
- n'autorise que les attributs d'instance,
- local à la méthode encapsulante,
- concepts-relations valides en tant que :

  source directe : héritage inter-classes (cf. page 7), implémentation (cf. page 9), agrégation (cf. page 10), agrégation de classe (cf. page 11), composition (cf. page 11) et composition de classe (cf. page 12)[28] ;

  cible directe : héritage inter-classes, concrétisation (cf. page 9), agrégation et agrégation de classe[29].

classe anonyme Les classes internes anonymes sont comme des classes locales auxquelles le nom a été retiré[30]. Elles sont donc utilisées directement dans des expressions. Une classe anonyme C peut être décrite par les capacités suivantes.

- simple,

[25] *Attribut d'instance* est un abus de langage car il s'agit en fait de variable locale et non d'attribut.

[26] L'agrégation de classe et la composition de classe sont permises pour la définition des constantes.

[27] L'agrégation de classe est possible par exemple dans le cas d'une classe locale qui définit une constante de classe du type d'une autre classe locale.

[28] L'agrégation de classe et la composition de classe sont permises pour la définition des constantes.

[29] L'agrégation de classe est possible par exemple dans le cas d'une classe locale qui définit une constante de classe du type d'une autre classe locale.

[30] On peut ainsi les rapprocher des lambda-expressions de *Lisp* et elles sont très utiles quand il s'agit, par exemple, d'implémenter une interface pour n'exécuter qu'en un seul endroit le code en question.

- générateur (une seule instance est générée pour C et elle l'est automatiquement),
- non destructeur,
- encapsulant,
- attributs et méthodes autorisés,
- attribut d'instance unique (l'instance de C est anonyme et créée au sein d'une expression, elle n'est donc pas partageable),
- surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
- local à l'objet issu de l'expression encapsulante,
- concepts-relations valides en tant que :

  source directe : héritage inter-classes (cf. page 7), concrétisation (cf. page 9), implémentation (cf. page 9), agrégation (cf. page 10), agrégation de classe (cf. page 11), composition (cf. page 11) et composition de classe (cf. page 12)[31] ;

  cible directe : agrégation et agrégation de classe[32].

interface membre statique Elles sont l'équivalent, du point de vue des interfaces, des classes membres statiques. Notez que les interfaces membres statiques sont les seules interfaces servant de sous-description en *Java* : il n'y a pas d'interface membre (non statique), d'interface locale ou d'interface anonyme. Imaginons donc une interface membre statique C et voyons comment nous pouvons la décrire.

- simple,
- non générateur,
- non destructeur,
- encapsulant,
- attributs interdits mais méthodes (abstraites) autorisées,
- autorise les attributs d'instance ou de description,
- surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
- local à la description encapsulante,
- concepts-relations valides en tant que :

  source directe : héritage inter-interfaces (cf. page 8), agrégation (cf. page 10), agrégation de classe (cf. page 11), composition (cf. page 11)[33] et composition de classe (cf. page 12)[34] ;

  cible directe : héritage inter-interfaces, implémentation (cf. page 9), agrégation et agrégation de classe.

[31] L'agrégation de classe et la composition de classe sont permises pour la définition des constantes.

[32] L'agrégation de classe est possible en programmant une variable de classe initialisée par un objet issu d'une classe anonyme.

[33] Agrégation et composition sont présentes ici en raison des paramètres de méthode et résultats de fonction.

[34] Les interfaces peuvent déclarer des constantes de classes, d'où la présence ici de l'agrégation de classe et de la composition de classe.

tableau Il s'agit d'un cas particulier de *Java*. Les tableaux sont manipulables par la classe java.lang.reflect.Array et sont modélisées par des pseudos-classes créées automatiquement lors de la création de leur première instance[35]. Une description tableau peut être caractérisée comme suit.
- simple (une description est créée pour chaque type d'élément : un tableau d'entiers n'est pas du même type qu'un tableau de booléens)[36],
- générateur,
- non destructeur,
- encapsulant,
- attributs (par exemple le pseudo-attribut length) et méthodes (en fait des opérateurs) autorisés,
- partageur,
- rien n'est surchargé dans cette pseudo-classe,
- global,
- concepts-relations valides en tant que :

  source directe : non applicable (il n'est pas possible de créer un nouveau concept-description tableau) ;

  cible directe : agrégation (cf. page 10) et agrégation de classe (cf. page 11).

type primitif Ils constituent un second cas particulier pour *Java* et sont au nombre de huit : booléen, caractère, octet, entier court, entier, entier long, flottant et flottant double. Contrairement à tous les autres, les types primitifs ne sont pas des objets. Voici leurs caractéristiques.
- Ces types disposent d'une interface syntaxique particulière : des mots-clés leur sont associés et il est possible d'utiliser directement des constantes de leur type (exemples : 16, true, ...).
- Chaque type primitif décrit une valeur et non une référence. Invoquer une primitive pour lui n'a donc pas de sens.
- Chaque type primitif est également décrit par un véritable classe (exemple : Integer pour int) ce qui permet d'utiliser des valeurs primitives en tant qu'objet (descendant d'Object).
- concepts-relations valides en tant que :

  source directe : non applicable (il n'est pas possible de créer un nouveau type primitif) ;

  cible directe : composition (cf. page 11) et composition de classe (cf. page 12).

Comme pour *Eiffel*, les classes représentant les chaînes (de caractères) ne constituent un cas particulier que du point de vue syntaxique.

[35] Les tableaux *Java* sont donc des objets.

[36] On peut donc voir la pseudo-classe tableau de *Java* comme générique, le paramètre de généricité étant le type des éléments.

4 Conclusion

Une telle étude permet de déceler un nombre réduit de paramètres qui définissent le comportement des relations entre descriptions et des descriptions elles-mêmes dans les langages à objets. Cette réflexion est à l'origine du modèle *OFL* qui réifie ces deux concepts et leur associe un ensemble de paramètres. Par une nouvelle valuation de ceux-ci, l'utilisateur (méta-programmeur) d'*OFL* a la possibilité de créer de nouveaux concepts-relations et concepts-descriptions adaptés aux besoins du programmeur. Mais là n'est pas le propos de ce rapport...

5 Remerciements

Je tiens à remercier **Robert Chignoli**, **Philippe Lahire** et **Adeline Capouillez** qui sont les trois autres co-concepteurs du modèle *OFL* et sans lesquels ce document, prémisse de ma thèse, n'aurait pu voir le jour.

Références

[AG00] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series... from the Source. Sun Microsystems, 3 edition, 2000.

[CCCL01] A. Capouillez, R. Chignoli, P. Crescenzo, and P. Lahire. Hyper-généricité pour les langages à objets : le modèle OFL. In *Conférence LMO 2001 (Langages et Modèles à objets)*. Hermes Science Publications, janvier 2001.

[CCL01] A. Capouillez, P. Crescenzo, and P. Lahire. OFL : l'hyper-généricité au service du méta-programmeur. Technical Report I3S/RR–2001-04–FR, Laboratory *Informatique, Signaux et Systèmes de Sophia Antipolis*, mars 2001.

[Fla99] D. Flanagan. *Java in a Nutshell : a Desktop Quick Reference*. O'Reilly, 3 edition, December 1999.

[Mey94] B. Meyer. *Eiffel, le langage*. InterEditions, 1994.