C++, Java, Smalltalk

Les concepts objets à l'épreuve

mars 99

Philippe PRADOS

pprados@club-internet.fr

Table des matières

Introduction 1

1 Les concepts 1

Classe et encapsulation 7
Héritages et polymorphisme 14
Classe abstraite 26
Classe contextuelle 29
Langage typé 32
Sémantique 34
Référence constante 38
Pureté syntaxique 41
Structure de choix 43
Structures de boucles 45
Fonctions 47
Retour de méthode 49
Surcharge 51
Redéfinition des opérateurs 53
Conversion d'objets 56
Conversion de références 59
Mutation 62
Gestion de la mémoire 64
Référence « faible » 70
Tableaux 72
Réflexivité 75
Constructeur 78
Destructeur 80
Référence sur une méthode 82
Traitements objets 84
Exception 86

Généricité ... 91
Multitraitement ... 94
Paquetage ... 97
Extensibilité ... 99
Persistance ... 101

2 Autres caractéristiques 103

Syntaxe ... 105
Robustesse ... 107
Compilation ... 109
Préprocesseur ... 113
Interface avec les autres langages ... 115
Portabilité ... 117
Normalisation ... 118
Maîtrise ... 119
Développement en équipe ... 121

3 Conclusion 123

Smalltalk ... 124
Java ... 126
C++ ... 128
Récapitulatif ... 130
Lexique ... 132
Bibliographie ... 134

Introduction

Pour faciliter la réalisation des programmes, plusieurs paradigmes ont été proposés. L'historien des sciences Thomas Kuhn [KU70] utilise le terme de paradigme pour désigner un ensemble de théories et de méthodes qui, associées, représentent une manière d'organiser la connaissance, de voir le monde. Parmi les différents paradigmes présents dans les langages de développement (impératif, objet, logique,...), le paradigme objet prend une part de plus en plus importante dans les développements.

L'origine de la programmation objet est liée au besoin de regrouper des éléments informatiques ayant des caractéristiques communes afin de faciliter la programmation et l'exploitation de gros logiciels. La programmation structurée a privilégié les traitements par rapport aux données. La programmation objet s'intéresse aux données et permet de définir les traitements associés.

L'émergence de langages offrant automatiquement le paradigme objet a facilité son implantation et son utilisation. Un concept facile à utiliser permet d'améliorer la qualité des programmes.

L'approche procédurale propose un regroupement par traitement (ou sous-programme). Le modèle de données a un impact important sur l'ensemble des traitements de l'application. Une modification de celui-ci a une incidence sur la plupart des traitements. En effet, il est difficile de réutiliser un sous-programme complexe.

La programmation objet peut être comme un style de programmation permettant d'améliorer la qualité des programmes en répartissant la complexité sur chaque objet. Un programme n'est alors qu'une collection de traitements associés à des données, appelés objets et regroupés pour fonctionner ensemble. Les données n'étant modifiées que par un ensemble restreint et identifié de traitements, les erreurs sont plus facilement localisables. Un autre apport de la programmation objet est le polymorphisme. Ce concept permet de sélectionner un traitement suivant l'objet manipulé et d'enrichir ou de réutiliser un programme existant.

Le modèle objet est utilisable lors de l'analyse, de la conception et du codage. Cela permet d'obtenir une cohérence totale lors du développement. Il est facile de retourner en conception alors que la phase de codage est déjà fortement amorcée. Les langages ne possèdent pas

toujours une traduction immédiate des concepts utilisés lors de l'analyse objet (tel est le cas de la relation « a un » par exemple), mais tout programme objet possède une traduction immédiate dans le modèle de conception. On ne retourne pas à la conception initiale mais on reste dans le paradigme objet.

Il n'est pas nécessaire d'utiliser un langage objet pour utiliser ce paradigme. Celui-ci peut s'exprimer avec un langage procédural traditionnel, mais cette procédure est très complexe, donc difficile à réaliser et à maintenir. Dans tous les cas, si un concept n'est pas présent dans un langage, il peut être traduit à la main. Il ne faut pas oublier qu'en final, cela se traduit par des ordres primitifs destinés au microprocesseur du poste de travail.

Plus un concept est apparent, plus le code est facile à maintenir et à comprendre. Par exemple, la syntaxe du langage peut exprimer clairement que l'on utilise une boucle de type « tant que ». Il est facile de rédiger une boucle équivalente sans utiliser la syntaxe proposée par le langage. Un commentaire peut alors indiquer que le code traduit une boucle de ce type.

Pour des concepts plus complexes, la démarche est identique. Un code peut être la traduction d'un concept absent du langage. Les commentaires peuvent indiquer l'existence d'un concept. Le plus souvent, le programmeur ne sait même pas qu'il les utilise. Les règles de traduction de ces concepts ne seront pas maîtrisées, et il y aura des erreurs qui auraient pu être évitées si le sens avait été parfaitement compris. C'est pour cela que des langages plus évolués ont été inventés. Après avoir identifié les concepts les plus courants en développement, les langages ont proposé une syntaxe permettant de les exprimer.

L'identification d'un concept facilite également la **communication** entre développeurs. Un vocabulaire commun permet de négliger les détails d'implémentation. L'expression d'une idée sera éclairée par l'utilisation d'un concept déjà connu des interlocuteurs.

Un concept est l'identification d'un traitement répétitif dans les développements. Il possède une traduction syntaxique. Un langage est équipé d'outils syntaxiques facilitant l'utilisation des concepts. Il est préférable d'avoir des informations de sens dans le langage plutôt que dans les commentaires. Les commentaires n'étant pas nécessaires à l'exécution du programme, ils sont souvent négligés. De plus, rien ne garantit leur adéquation au programme. C'est pourquoi il est préférable d'utiliser les éléments syntaxiques du langage plutôt qu'un codage à la main. Un concept est vérifié par le compilateur. Attention : l'existence d'un concept n'entraîne pas forcément la justesse de son utilisation. Il faut bien comprendre son objectif pour utiliser correctement les éléments syntaxiques correspondants.

La qualité d'un langage de programmation s'évalue selon sa richesse, sa cohérence, sa facilité d'utilisation et son intégration des concepts mis en place. De plus, l'environnement de développement doit apporter une aide importante pour faciliter la réalisation des logiciels, mais il est très difficile de comparer les environnements de développement, car ceux-ci sont spécifiques à chaque fournisseur.

Les langages, au contraire, sont normalisés. Ils se concurrencent sur la qualité. Les fournisseurs s'affrontent alors sur les environnements de développement et cherchent à imposer leurs bibliothèques pour fidéliser leurs clients. Smalltalk est un cas à part, car il est très fortement lié à ses environnements de développement. Choisir Smalltalk, c'est également choisir un fournisseur d'environnement et de bibliothèques.

Il existe pour tous les langages orientés objet, des bibliothèques permettant de créer des applications interactives avec fenêtres. Les approches sont différentes, mais les fonctionnalités

très similaires. Les environnements de développement proposent des outils d'aide à la manipulation du langage (éditeur, navigateur de classe, références croisées, débogueurs, etc.), et des outils de génération de code spécifiques à leurs bibliothèques.

La famille **Visual Age™** d'IBM est un exemple d'environnement adaptable à différents langages. Cela permet d'ajouter la notion de composants logiciels (`Parts`) absents du langage, mais supportés par l'environnement. Visual Age propose aussi la programmation visuelle.

Dans ce document, les concepts présents dans les environnements ne sont pas décrits.

But du livre

Cet ouvrage reprend les concepts essentiels présents dans les langages orientés objet. Il permet de les comparer, non par leurs syntaxes ou leurs approches techniques, mais par la richesse et la facilité de mise en œuvre de leurs idées.

Ce document se propose donc de comparer les trois langages orientés objet les plus en vogue : Smalltalk, Java™ et C++. Cet ordre n'est pas chronologique. En effet, Java est le plus jeune des trois. Intercalé entre Smalltalk et C++, sa place correspond à sa position technique, même s'il est proche des deux autres. Cet ordre facilite la description séquentielle des approches.

Smalltalk

Smalltalk est le pionnier dans le domaine de la programmation objet avec héritage fondé sur une hiérarchie (il s'inspire de Simula). Il a été créé en 1972 par Alan Kay, de l'entreprise Rank Xerox Parc. C'est un langage interprété à base de machine virtuelle (par l'intermédiaire d'un byte-code), ayant subi de nombreuses évolutions au cours de son existence. Il propose un nombre très restreint de concepts qu'il utilise judicieusement afin d'offrir un environnement de développement uniforme. Il s'apparente plus à un L4G (langage de 4^{e} génération) qu'à un langage traditionnel. Son environnement de développement est écrit avec lui-même. Il est pratiquement impossible de séparer l'environnement du langage. Il propose une exécution portable sous différents OS (Windows, Unix, Mac OS, OS/2…).

C++

C++ est une extension objet du langage C. Développé aux Bell Laboratory (ATT) par Bjarne Stroustrup (1983), il s'inspire de Simula, le premier langage orienté objet, conçu quant à lui en 1967 par Dahl et Nygaard, de l'université d'Oslo. Grand frère du C, il a rapidement conquis le marché des langages orientés objet. Comme extension du C, il a hérité d'un passé pas toujours joyeux. Sa syntaxe est particulièrement difficile. Néanmoins, il présente une très grande performance.

Java

Java est un concurrent récent ayant bénéficié de la vague Internet pour s'imposer. Initialement, il a été inventé par Sun pour permettre la création de programmes embarqués sur des périphériques grand public (télévision, machine à laver...). Ses concepteurs ont judicieusement adapté leurs langages à Internet et l'ont proposé gracieusement sur le réseau. Java a acquis en

moins d'un an une notoriété telle que tous les acteurs majeurs de l'informatique s'y sont attelés. Il sera bientôt livré en standard avec tous les systèmes d'exploitation. C'est un langage portable, indépendant de la plate-forme. Il utilise une « machine virtuelle » qui est supportée par les différents environnements. Ce langage répond à une demande forte des développeurs qui sont confrontés régulièrement aux problèmes de portabilité. Un programme écrit en Java est directement utilisable sous Windows, Unix ou Mac OS par exemple.

Structure de l'ouvrage

Les idées essentielles du modèle objet sont présentées dans ce document. Les réponses proposées par ces trois langages sont étudiées. Le concept est décrit schématiquement, sans faire intervenir de langage particulier. Ensuite, vous découvrez comment chacun y répond ou comment utiliser le langage pour répondre au concept. En effet, si certaines idées ne sont pas syntaxiquement présentes dans les langages, le développeur peut alors traduire à la main les concepts présents dans les autres langages. Grâce à cet ouvrage, vous apprendrez ce qu'est la programmation objet et ce qui différencie ces trois langages.

Le paradigme objet utilise un vocabulaire qui lui est propre. Il vous sera présenté au fur et à mesure. Vous trouverez en annexe un lexique qui vous permettra de vérifier la définition d'un terme.

Le livre est découpé en trois parties : la première, *Les concepts*, fait place aux *Autres caractéristiques*, dans la deuxième partie, qu'achève la troisième partie en guise de *Conclusion*.

Je remercie Stéphane Ducasse et Dan Dimcea, Philippe Duret, Paul-Henri Lampe, Marc Legru et Le Xiaohua pour leurs nombreuses remarques.

Pour me contacter : `pprados@club-internet.fr`

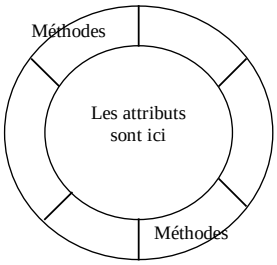
Partie 1

Les concepts

Cette partie évoque les principaux concepts mis en place par les langages orientés objets. La classification des concepts a été choisie pour faciliter leurs introductions.

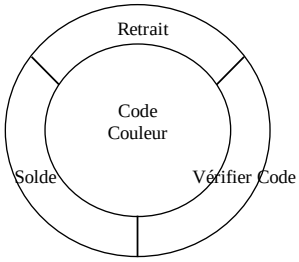
Classe et encapsulation

Le premier principe du modèle objet est l'**encapsulation** des données. Les données d'un objet sont appelées **attributs**. Ces derniers sont cachés dans l'objet et ne peuvent être manipulés qu'au travers d'une interface publique. Les fonctions de l'interface sont appelées **méthodes**. Cette approche permet de modifier profondément les données sans intervenir sur l'interface. Les évolutions d'un objet n'ont ainsi pas d'impact sur les développements en cours. Il y a séparation entre le « quoi » et le « comment », c'est-à-dire la spécification et l'implantation. Le « quoi » est public, le « comment » est caché. Le concept d'encapsulation peut être défini comme une protection d'un objet ou des parties d'un objet contre des accès non autorisés ou mal intentionnés.

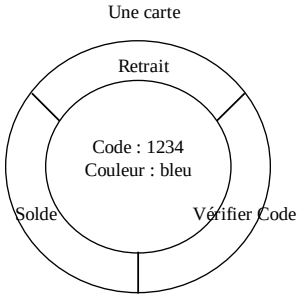


Les langages utilisent différentes approches pour contrôler l'accès à ces données. Tous les langages orientés objet possèdent une traduction syntaxique de ce concept. Ainsi, la lecture du code source d'un de ces langages fait immédiatement apparaître la notion de **classe**. Une classe est une construction syntaxique permettant de définir les attributs d'un objet et les méthodes associées. C'est une abstraction permettant de factoriser la structuration des données et les comportements associés aux objets. Par conséquent, c'est la classe qui permet d'organiser et de structurer les programmes.

Une classe est un moule permettant de générer des **instances**. Les instances mémorisent les états des objets et partagent les comportements avec les autres instances de la même classe. Un objet est une instance d'une classe. Les attributs sont dans l'objet. Ils ne sont accessibles, par un client, que par l'intermédiaire des méthodes. Dans le vocabulaire objet, on parle d'instance d'une classe pour chaque objet. Lorsqu'on crée un objet, on construit une instance de sa classe. La classe d'une carte de crédit peut être représentée comme ceci :



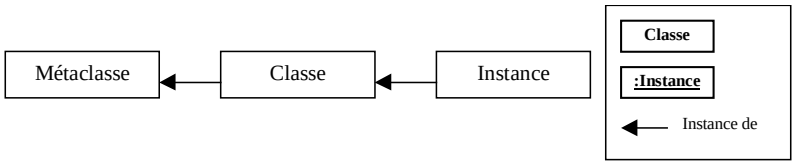
Toutes les instances de cette classe possèdent deux attributs : « Code » et « Couleur ». Leur valeur dépend de chaque instance. Par exemple, une instance carte de crédit peut être représentée comme ceci :



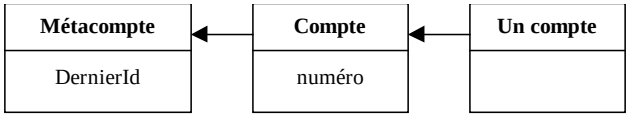
Un objet particulier « Une carte » de type « carte de crédit » possède un attribut appelé « Code » dont la valeur est « 1234 » et un attribut appelé « Couleur » de valeur « bleu ». Il existe trois méthodes pour manipuler l'objet : « Solde », « Retrait » et « Vérifier code ». Le comportement des méthodes est mémorisé dans la classe. En effet, toutes les cartes de crédit utiliseront les mêmes traitements, mais les appliqueront à des instances différentes. Cette approche permet de factoriser les traitements. Un paramètre ou une variable particulière de la méthode (`this` ou `self`) pointera sur l'instance devant être manipulée.

Chaque instance garde un lien (physique ou logique) avec la classe qu'elle représente. Cela permet de retrouver la méthode à appliquer à partir d'une instance.

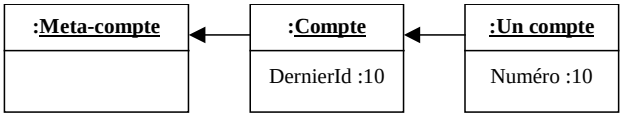
La classe peut aussi être une instance d'une autre classe (alors appelée métaclasse).



Une méthode permet de manipuler les attributs de l'instance courante et éventuellement ceux de sa classe. Un attribut de classe peut être partagé par l'ensemble des instances de la classe. Par exemple, si l'on crée une instance de `compte`, elle doit posséder un `numéro` qui lui est propre. Une classe `compte` déclare un attribut de classe `dernierId`. Une méthode `initialise` applicable à un `compte` peut demander l'incrémentation de `dernierId` à la classe pour valoriser le numéro du nouveau compte. Le modèle objet est celui-ci :



Le modèle décrit dans les classes les attributs présents dans les objets. Les attributs sont physiquement localisés comme ceci :



La méthode `initialise` sera traduite ainsi :

```
this.Numero=class.DernierId()
class.incrementeDernierId()
```

Les langages objet proposent un accès direct aux attributs et aux variables de la classe. `initialise` peut être simplifiée comme ceci :

```
Numero=DernierId()
incrementeDernierId()
```

Les langages objet offrent plusieurs sortes de variables.

- **Les variables locales (ou temporaires)** sont déclarées dans le corps d'une méthode. Elles disparaissent lorsque la méthode est terminée.

- **Les paramètres** sont valorisés lors de l'appel d'une méthode et disparaissent avec la méthode.
- **Les attributs** sont les variables d'une instance. Ils ne sont généralement accessibles que par les méthodes définies dans la classe. Les attributs appartiennent à l'instance. Ils disparaissent lorsque l'instance disparaît.
- **Les variables de classe** appartiennent à la classe. Elles sont partagées par l'ensemble des instances de la classe et de ses sous-classes (voir « Héritages », page 14). Ce sont des variables accessibles par les instances de la classe. Elles disparaissent lorsque la classe disparaît.
- **Les variables globales** sont accessibles par toutes les méthodes de toutes les classes. Elles n'appartiennent pas à une classe seule. Elles disparaissent à la fin du programme.

Lorsque le langage rencontre un nom de variable, il le recherche dans différents dictionnaires successifs. La recherche commence dans le dictionnaire des variables locales et se termine dans celui des variables globales. Si la recherche n'aboutit pas dans le niveau courant, elle continue dans le niveau inférieur. Si la variable n'est toujours pas trouvée, une erreur est signalée.

De même, les méthodes définissent des traitements de classes ou d'instances.

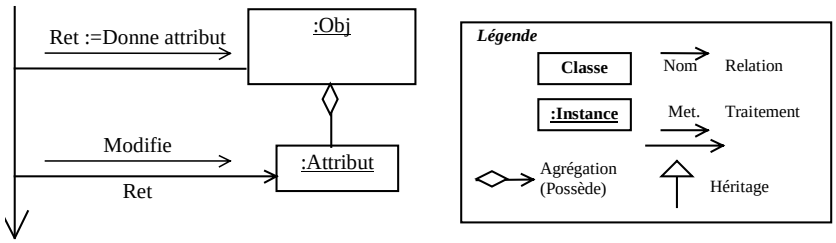
- Une méthode d'instance peut ne s'appliquer qu'à une instance. Il faut posséder une instance pour lui appliquer la méthode.
- Une méthode de classe peut être exécutée sans posséder d'instance de la classe (mais une instance de la métaclasse). Elle est partagée par l'ensemble des instances de la classe mais peut également être appelée par d'autres méthodes n'appartenant pas à la classe.

Dans les paragraphes suivants, nous allons étudier les approches proposées par les trois langages Smalltalk, Java et C++ en matière de classe et d'encapsulation.

Smalltalk

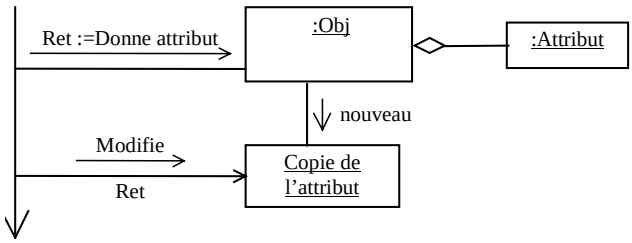
Avec Smalltalk, les attributs d'une instance sont uniquement accessibles par les méthodes de celle-ci. Une méthode ne peut pas accéder aux attributs d'une autre instance, même si elle appartient à la même classe. L'accès est autorisé pour l'instance et non pour la classe.

Le seul moyen, pour un client, d'avoir accès aux attributs d'un objet est de passer par un **accesseur**, c'est-à-dire une méthode particulière rédigée à cette fin. Cet accesseur permet d'autoriser ou non la modification d'un attribut et d'en contrôler les effets de bords. S'il n'existe pas de méthode pour manipuler un attribut, celui-ci est inaccessible à l'extérieur de l'objet. En revanche, comme ces méthodes d'accès retournent une référence sur les attributs de l'objet (un pointeur sur l'attribut), rien n'interdit à l'utilisateur de modifier directement un attribut en utilisant cette référence.



La méthode « donne attribut » retourne une référence sur son attribut (`Ret`). L'appelant peut alors utiliser directement cette référence pour manipuler l'attribut à l'insu de l'objet. La référence pointant sur l'attribut donne un accès à celui-ci à l'insu de l'objet propriétaire.

Les règles d'usage proscrivent cette démarche, mais ce n'est pas une contrainte syntaxique. Pour interdire rigoureusement l'accès à un attribut d'un objet, il faut retourner une copie de celui-ci dans l'accesseur. Cette copie permet de retourner la *valeur* de l'attribut et non l'attribut lui-même.



Ainsi, la méthode « donne attribut » retourne une référence sur une copie de l'attribut. La copie peut être manipulée sans risque pour l'objet propriétaire.

Pour des raisons de performance, on ne le fait généralement pas. On prend ainsi le risque d'avoir des erreurs dissimulées. L'encapsulation existe en Smalltalk, mais elle est perméable.

Smalltalk propose des paramètres, des variables locales, d'instance, de classe et des variables globales. Les paramètres de Smalltalk ne sont pas modifiables dans une méthode.

Smalltalk propose également des variables de pool. Ces variables sont partagées par un ensemble de classes non reliées par héritage. Elles sont dans un dictionnaire partagé par les instances. Chaque classe possède un ensemble de dictionnaire de variables partagées. Ce sont des variables globales regroupées dans un dictionnaire pour en limiter l'utilisation.

Smalltalk propose les métaclasses. Une métaclasse est générée automatiquement lors de la création d'une classe. Une méthode de classe particulière (`initialize`) permet d'initialiser les variables de classe et/ou les variables partagées.

Java

Java propose les paramètres, les variables d'instance et de classe, mais n'offre pas de variables globales. Une variable appartient soit à une instance, soit à une classe.

Une méthode peut accéder à tous les attributs de toutes les instances de la même classe. L'accès est autorisé pour la classe (Smalltalk l'autorise pour l'instance).

Il existe quatre niveaux de protection d'accès aux attributs et aux méthodes. Chaque attribut et chaque méthode peuvent être déclarés avec une des protections `private`, `protected`, `public` ou sans aucune d'entre elles.

- Un accès `private` limite l'utilisation au corps de la classe. Les classes dérivées ou les utilisateurs externes à la classe ne peuvent pas accéder à ces éléments. Une classe est dite dérivée si elle hérite de la classe courante. Le concept d'héritage est décrit un peu plus loin.
- Un accès `protected` limite l'utilisation des attributs ou des méthodes à la classe ou à ses dérivés. L'extérieur de la classe n'a pas accès à ces éléments.
- Un accès `public` autorise tous les accès.
- L'absence de qualificatif offre un accès aux classes appartenant au même groupe de classe (voir « Paquetage », page 97). Un groupe de classes est un regroupement logique permettant d'offrir des accès privilégiés aux classes du même groupe. Les classes du même groupe peuvent utiliser les attributs ou les méthodes de ce type. C'est une approche comparable (mais différente) à celle des variables de pool de Smalltalk, le dictionnaire étant le « paquetage ».

Accessible	Private	Protected	Public	Paquetage
De l'intérieur	Oui	Oui	Oui	Oui
D'une classe dérivée	Non	Oui	Oui	Non*
D'un paquetage	Non	Oui	Oui	Oui
De l'extérieur	Non	Non	Oui	Non

* Sauf si la classe dérivée appartient au même paquetage.

Les mécanismes de protection de Java interdisent rigoureusement les accès depuis un client. Il n'est pas possible de contourner les protections syntaxiques pour consulter un attribut privé. C'est un des éléments de protection du langage. Il n'est pas possible de consulter ou de modifier un attribut si le rédacteur de la classe ne l'a pas autorisé.

Les classes Java appartiennent à un groupe de classes. C'est un peu comme dans une entreprise. Il peut y avoir plusieurs classes dans l'entreprise : les classes `Commerciaux`, `Produits`, etc. Certaines classes ne sont connues qu'à l'intérieur de l'entreprise (la classe `FicheDePaie` par exemple). Seules les méthodes de l'entreprise peuvent les manipuler. Au sein d'un groupe, les restrictions d'accès peuvent être allégées. Des classes peuvent être cachées pour l'extérieur du groupe. Seules les méthodes du groupe peuvent manipuler les instances de ces classes cachées. Elles ne sont pas connues à l'extérieur du groupe.

Les méthodes d'accès aux attributs en Java renvoient, comme Smalltalk, une référence sur ces attributs. Il n'y a aucune garantie qu'un attribut ne sera pas modifié par mégarde. Comme pour

Smalltalk, il faut dupliquer l'attribut pour que sa valeur soit renvoyée, et garantir ainsi l'encapsulation. Celle-ci est perméable.

Depuis la version 1.1, il est possible de déclarer une classe dans une classe (*inner class*). Cela permet de construire des classes techniques ne pouvant être vues que par une seule classe. On évite de polluer la table des symboles et les conflits de noms entre classes techniques. C'est un enrichissement de l'encapsulation (voir « Classe contextuelle », page 29). Par exemple, une liste chaînée possédera une classe `Noeud`. Ce nom de classe pourra être utilisé par une autre, pour un arbre binaire par exemple. Le `Noeud` de la liste chaînée sera différent du `Noeud` de l'arbre binaire car ils auront été déclarés dans des classes différentes. Les noms des deux classes ne sont pas en conflit.

C++

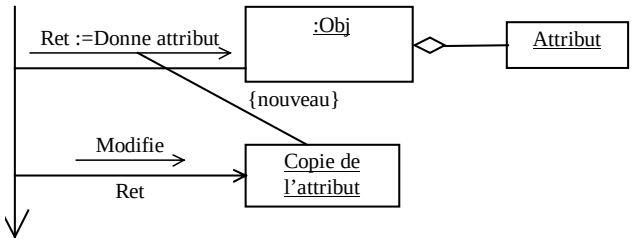
Le C++ propose les paramètres, les variables locales, d'instance, de classe et les variables globales.

Il utilise le même mécanisme de protection des données que Java, mis à part l'accès limité à un ensemble de classes, qui n'existe pas. L'absence de qualificatif n'équivaut à un accès `private`.

Il est parfois nécessaire d'accéder aux attributs et aux méthodes protégées. Pour cela, il est possible de déclarer une classe comme amie d'une autre (`friend`). Une classe amie a le droit d'accéder aux attributs et aux méthodes protégées.

Contrairement à Java, il est possible pour les *hackers* — moyennant une gymnastique complexe — de violer ces droits. Les protections sont uniquement syntaxiques, et non techniques.

Le C++ retourne des objets et non des références sur les objets.



Il y a toujours copie lors d'un appel ou du retour d'une méthode. Il n'est pas possible de modifier un attribut sans l'autorisation de l'objet. L'encapsulation est respectée.

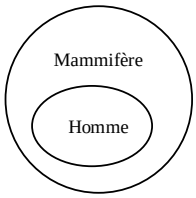
Comme pour Java, le C++ autorise la déclaration de classe dans une classe. On réduit ainsi les risques de conflits de noms entre classes.

Héritages et polymorphisme

Outre l'encapsulation, l'héritage et le polymorphisme constituent deux concepts important du modèle objet. Ces notions sont intimement liées ; l'héritage perd une grande partie de son intérêt sans le polymorphisme.

Héritages

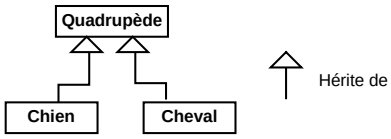
Le deuxième concept de base des langages orientés objet est la capacité à créer une classe par un **héritage** d'une autre classe. Un héritage permet de concevoir une spécialisation d'une classe. C'est l'identification d'un sous-ensemble. Par exemple, si une classe `Homme` hérite de la classe `Mammifère`, cela signifie que tout `Homme` est une sorte de `Mammifère`. La classe des `Hommes` est un sous-ensemble de la classe des `Mammifères`.



Tous les attributs et tous les comportements de la classe héritée sont présents dans la classe **dérivée**. On dit qu'une classe dérive d'une autre si elle en hérite. La classe `Homme` dérive de la classe `Mammifère`. Une classe peut hériter d'une autre, mais ne peut pas modifier dynamiquement l'héritage pour une instance donnée. Une école, par exemple, ne peut pas hériter momentanément de « bureau de vote ». L'héritage est figé une fois pour toutes lors de la déclaration de la classe.

Construire une classe dérivée, c'est traduire l'existence d'une spécialisation. Les `Hommes` sont des sortes de `Mammifères`. Ils répondent à toutes leurs caractéristiques et en ajoutent de nouvelles. Au contraire, regrouper les caractéristiques de plusieurs objets s'effectue en déclarant une classe de base. Par exemple, pour regrouper les caractéristiques des `Chiens` et des `Chevaux`, il faut construire une classe `Quadrupèdes`. `Chiens` et `Chevaux` héritent de `Quadrupèdes`. Il s'agit alors d'une *généralisation*. La création de la classe `Quadrupèdes` permet de généraliser les caractéristiques des `Chiens` et des `Chevaux`.

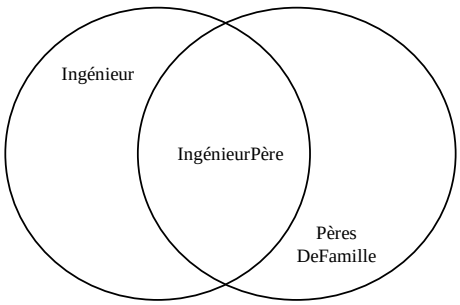
L'héritage peut être représenté à l'aide d'un arbre.



Descendre un arbre d'héritage permet d'identifier les spécialisations ; le remonter identifie les généralisations.

Suivant les langages, on peut hériter simultanément d'une ou plusieurs classes. Une classe appartenant à l'intersection de plusieurs ensembles peut être traduite à l'aide de l'**héritage multiple**.

Par exemple : soit l'ensemble des `Ingénieurs` et l'ensemble des `PèresDeFamille`. Un individu `IngénieurPère` appartient à la fois aux deux ensembles.



L'héritage multiple est un enrichissement pouvant entraîner des confusions. Si une classe hérite de deux classes distinctes ayant des attributs ou des méthodes de même nom, la classe dérivée doit régler ces ambiguïtés.

Par exemple, il peut exister un attribut `nom` dans la classe `Ingénieurs` et un attribut équivalent dans la classe `PèresDeFamille`. Lors de la déclaration de la classe `IngénieurPère`, ces deux attributs entrent en conflit. Le langage doit offrir un moyen de le lever.

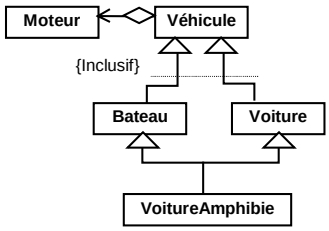
Avec un héritage multiple, il existe une subtilité supplémentaire : l'**héritage virtuel**. Prenons l'exemple d'un véhicule à moteur. La classe `Véhicule` possède un attribut `Moteur`. Une classe `Voiture` hérite de `Véhicule`. Une autre classe `Bateau` hérite également de `Véhicule`. Une classe `VoitureAmphibie` hérite de `Voiture` et de `Bateau`. Dans cette optique, il existe deux possibilités :

- Soit il existe deux moteurs, un pour la propulsion du véhicule en tant que `Voiture` et un pour la propulsion en tant que `Bateau`.
- Soit le moteur est partagé par les deux propulsions.

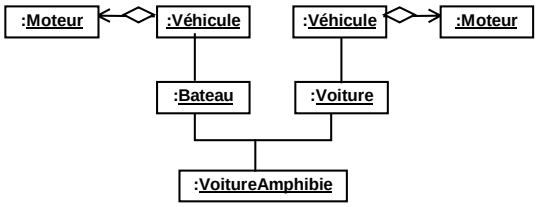
Dans le premier cas, l'arbre d'instances se traduit comme ceci :

Héritage normal

Modèle de classes



Modèle d'instances

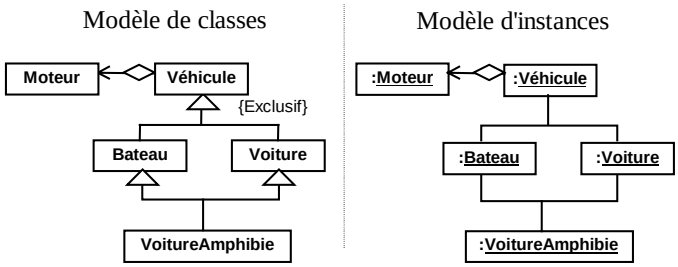


La flèche commencée par un losange représente une **agrégation**. Cela veut dire que l'objet `Moteur` est possédé par l'objet `Véhicule`. Un véhicule agrège différents objets, dont un moteur.

La `VoitureAmphibie` possède deux moteurs : celui du bateau et celui de la voiture.

Dans le deuxième cas, l'arbre d'instances se traduit comme ceci :

Héritage virtuel



La `VoitureAmphibie` ne possède qu'un seul moteur, commun au bateau et à la voiture.

L'héritage virtuel permet de déclarer que la classe de base sera partagée ou non en cas d'héritage multiple.

Des objets peuvent posséder des similitudes sans entraîner la création d'une classe particulière pour les factoriser. Si l'on ne désire pas, ou que l'on ne peut pas utiliser d'héritage multiple, il existe d'autres techniques permettant d'employer indifféremment plusieurs classes n'appartenant pas à la même branche de la hiérarchie. Ces classes doivent posséder les mêmes méthodes avec les mêmes **signatures** (mêmes noms et les mêmes paramètres). Par exemple, une méthode `ajoute` est très courante dans les objets mais ne doit pas nécessairement entraîner la création d'une classe spécifique pour regrouper toutes les classes offrant ce service. Comment bénéficier des similitudes sans pour autant utiliser la notion d'héritage ? Les différents langages utilisent des approches techniquement très différentes pour résoudre cette difficulté.

Polymorphisme

Le **polymorphisme** est un mécanisme permettant d'appeler une même méthode sur différents objets et de provoquer des comportements qui dépendent de la nature du receveur. Il permet de spécialiser une classe héritée en lui ajoutant les modifications de traitements spécifiques à la classe.

Par exemple, un pointeur référence un compte en banque. Le programme lui demande d'afficher son solde. Il invoque la méthode correspondante en désignant le compte référencé.

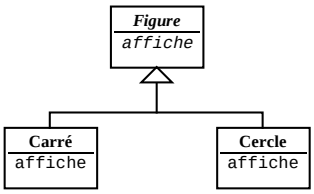
Plus tard, le même pointeur est utilisé pour lui faire référencer un plan d'épargne logement. Le programme demande l'affichage du solde du compte référencé. Il invoque, comme précédemment, la méthode correspondante, en désignant l'objet manipulé à l'aide de la référence.

L'invocation de la méthode est identique pour le compte en banque et pour le plan d'épargne logement. Le type de l'objet référencé permet de sélectionner le traitement correspondant à l'objet manipulé.

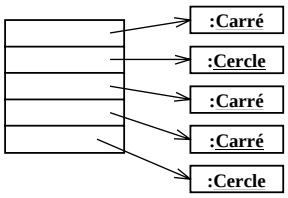
```
Déclare X comme référence de type CompteEnBanque
X=nouveau CompteEnBanque
appel affiche pour la référence X   calcul et affiche le solde du
CompteEnBanque
X=nouveau PEL
appel affiche pour la référence X   calcul et affiche le solde du PEL
```

L'application n'a pas à être modifiée si elle pointe sur un objet de type `CompteEnBanque` ou sur un objet dérivé de `CompteEnBanque` (hérité de `CompteEnBanque`).

Avec le modèle objet, il est possible d'enrichir un logiciel sans modifier les développements en cours. Imaginons maintenant l'arbre d'héritage suivant :



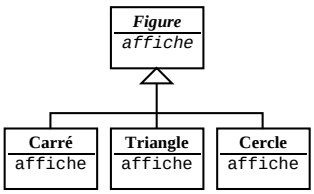
La classe `Figure` possède une méthode `affiche` qui est redéfinie dans les classes `Cercle` et `Carré`. Maintenant, utilisons un tableau de figure et ajoutons dans le désordre des instances de `Cercle` et de `Carré`.



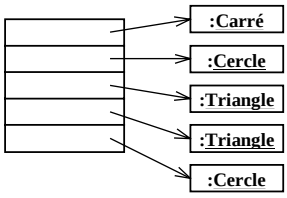
Le polymorphisme va permettre de rédiger facilement une routine demandant à chaque objet du tableau de s'afficher. La méthode `affiche` sera demandée pour chaque objet. Si l'élément est un `Cercle`, celui-ci sera automatiquement dessiné. S'il s'agit d'un `Carré`, la méthode correspondante sera exécutée. La routine n'utilise pas d'instruction permettant de faire un choix en râteau (type `switch` ou `case`). Elle demande à chacun des objets de s'afficher, sans avoir à connaître précisément l'objet manipulé.

```
Pour chaque élément du tableau
   appeller la méthode 'affiche'
Suivant
```

Maintenant, imaginons que l'on ajoute à l'arbre d'héritage une nouvelle figure, un triangle par exemple.



Ajoutons des triangles dans le tableau.



Sans le modifier, l'algorithme va être capable de dessiner des triangles alors qu'au moment de sa rédaction, il ne soupçonnait pas leur existence. La même méthode est appelée, mais le choix du traitement à exécuter dépend de l'objet receveur. C'est cette capacité que l'on appelle le polymorphisme.

Avec cette approche, il n'est plus nécessaire de rédiger des aiguillages compliqués pour tenir compte de tous les cas particuliers. Une règle en programmation objet indique d'ailleurs qu'il faut éviter d'utiliser les aiguillages, car c'est un signe d'une mauvaise conception. Il est préférable d'utiliser le polymorphisme.

Smalltalk

Smalltalk autorise l'héritage simple. Une classe ne peut hériter que d'une seule autre classe. La classe `Object` représente le sommet de l'arbre d'héritage.

Il est également possible de créer une autre branche d'héritage. Dans ce cas, Smalltalk ajoute automatiquement une nouvelle version de la méthode `doesNotUnderstand:` (traitement d'erreur si une méthode n'est pas trouvée dans un objet), dont le but est de recopier automatiquement et de façon paresseuse (lorsqu'un appel échoue) dans cette nouvelle classe

racine les méthodes de la classe `Object` nécessaires. On aboutit à une copie incrémentale de la classe `Object`.

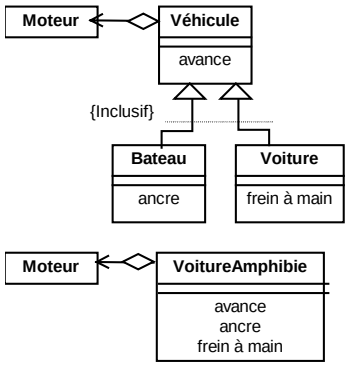
Smalltalk autorise un polymorphisme sans contrainte. Une méthode peut être appelée pour tout objet pouvant y répondre, que celui-ci ait une relation d'héritage ou non. Le mécanisme de résolution ne s'effectue qu'à l'exécution. L'appel d'une méthode sur une instance ne sachant pas y répondre provoquera l'arrêt du programme (s'il n'y a pas de récupération de l'erreur). Lors de la réception d'un message (au moment de l'appel d'une méthode), le moteur Smalltalk le recherche dans le dictionnaire de la classe de l'objet. S'il n'est pas trouvé, la recherche continue dans la hiérarchie des classes. Si la méthode n'est toujours pas trouvée, un traitement d'erreur est exécuté.

Il est possible de capturer ces erreurs (méthode `doesNotUnderstand:`). Il est alors permis de renvoyer le message à un autre objet. Cette technique est très utilisée pour implanter des objets intermédiaires s'occupant de transmettre le message par réseau. La récupération de l'erreur transmet le message par le réseau à un serveur qui se chargera d'exécuter le traitement. Ce n'est pas la meilleure technique pour faire cela, mais c'est la plus utilisée [DUC97].

Pour demander l'utilisation d'un service de la classe héritée, il faut utiliser la pseudo-variable `super`. Cela permet d'appeler une méthode déclarée dans une des classes héritées, sans tenir compte de la redéclaration présente au niveau de l'appel. Par exemple, la classe `A` déclare la méthode `f`. La classe `B` hérite de `A` et déclare également la méthode `f`. L'appel de `f` par l'intermédiaire de la pseudo-variable `super` permet d'appeler la version de `f` déclarée par la classe `A` à la place de la version de `f` déclarée par la classe `B`.

Pour pallier l'absence d'héritage multiple, on peut définir des méthodes de signatures identiques (même nom et mêmes paramètres) dans des objets appartenant à des branches différentes de l'arbre d'héritage. La résolution des accès aux méthodes s'effectuant à l'exécution, deux objets de types différents peuvent répondre aux mêmes services si ceux-ci possèdent les mêmes noms.

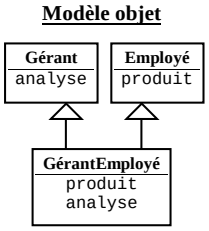
Par exemple, pour résoudre l'exemple de l'héritage virtuel de l'exemple précédent, il faut construire une classe `VoitureAmphibie` n'héritant d'aucune autre, mais simulant le comportement des `Bateaux` et des `Voitures`.



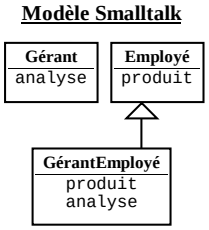
Toutes les méthodes des `Bateaux` et des `Voitures` doivent être redéfinies dans la classe `VoitureAmphibie`. Ainsi, partout où un `Bateau` ou une `Voiture` est utilisable, une `VoitureAmphibie` l'est aussi.

Le concept de l'héritage multiple n'apparaît pas dans la syntaxe. Il faut alors ajouter les commentaires nécessaires pour indiquer que le programme est écrit dans l'optique d'un héritage multiple, pour *simuler* l'héritage multiple.

Un autre exemple : il existe une classe `Employé` et une classe `Gérant`. Pour modéliser un `Gérant` salarié, une classe `GérantEmployé` est déclarée comme héritant de ces deux classes. Le modèle objet est celui-ci :

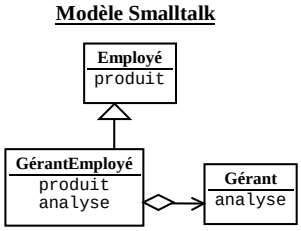


Smalltalk ne possédant pas d'héritage multiple, il faut adapter le modèle comme ceci :



La classe `GérantEmployé` n'hérite que de `Employé`. Les méthodes déclarées dans l'interface de la classe `Gérant` sont déclarées à l'identique dans `GérantEmployé`.

`GérantEmployé` peut également avoir un objet `Gérant` en attribut et lui déléguer des services (lui sous-traiter des services).



Java

Java n'autorise que l'héritage simple. Comme Smalltalk, toutes les classes dérivent de la classe `Object`. Celle-ci offre les services minimaux nécessaires à tous les objets. La pseudo-variable `super` permet d'appeler les méthodes héritées d'un objet.

Java permet également de déclarer une classe comme finale, ce qui interdit toutes les dérivations. Une classe finale ne pourra pas être héritée.

Java ajoute une sorte d'héritage multiple appauvri : les « interfaces ». Ce sont des classes déclarant des méthodes sans jamais définir leurs implantations. Java possède l'héritage simple d'implantation et l'héritage multiple d'interfaces.

L'interface permet de formaliser une similitude de comportement. Ce n'est pas une classification de type, contrairement à l'héritage. Des objets appartenant à des branches différentes d'un arbre d'héritage peuvent partager la même interface.

Java n'autorise le polymorphisme qu'avec les classes partageant une classe de base ou une même interface. Le compilateur refusera un appel de méthode sur une instance n'étant pas capable d'y répondre.

Par exemple, une interface est présente dans la bibliothèque pour décrire les objets étant capables d'être dupliqués (`Cloneable`). Cette interface définit la méthode de duplication (`clone`) qui doit être rédigée dans chaque classe l'implémentant. Des objets de tout type peuvent se dupliquer sans pour autant représenter des concepts similaires. Un `CompteEnBanque` ou un `Paragraphe` peuvent se dupliquer ; ils ne doivent pas pour autant hériter de la même classe.

Tous les objets désirant partager une interface doivent explicitement se déclarer comme l'implémentant. Une interface ne peut hériter que d'une autre interface (ou de plusieurs).

Deux objets se ressemblant mais ne déclarant pas partager la même interface ne pourront pas utiliser les mêmes services.

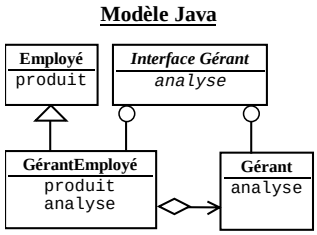
Par exemple, si l'objet `CompteEnBanque` et l'objet `Paragraphe` déclarent une méthode `clone`, mais n'implantent pas l'interface correspondante, il ne sera pas possible de bénéficier

du **polymorphisme** entre ces deux classes. Imaginez un tableau d'objets. Si vous utilisez ce tableau et désirez y regrouper des `CompteEnBanque` et/ou des `Paragraphes`, vous ne pourrez pas utiliser la méthode `clone` de ces objets. En revanche, s'ils héritent de la même interface, vous pourrez parcourir le tableau et demander la duplication de tous les objets, indépendamment de leur type. Chaque objet répondra à sa manière à l'invocation du service.

```
Pour tous les éléments du tableau
  appeler la méthode 'clone' de l'interface
éléments suivant
```

La boucle appelle la méthode `clone` de `CompteEnBanque` ou de `Paragraphe` suivant l'objet présent dans le tableau.

Pour le modèle objet décrit précédemment, les modifications à apporter pour Java sont les suivantes :



Il faut créer une interface *`InterfaceGérant`*, et déclarer `Gérant` et `GérantEmployé` comme l'implantant.

Avec l'adjectif `final`, une méthode peut indiquer qu'elle refuse d'être redéfinie dans les classes dérivées. Le compilateur peut optimiser le byte-code généré. Définir des contraintes pour les dérivées ne facilite pas la réutilisation d'une classe. En revanche, cela permet de garantir l'encapsulation en interdisant une classe dérivée d'intercepter un service protégé. C'est un des éléments de sécurité de Java.

C++

Le C++ autorise l'héritage multiple et l'héritage virtuel (attribut `virtual` de l'héritage). La résolution de l'ambiguïté des attributs ou des méthodes s'effectue en préfixant le nom de l'élément par celui de la classe. Cela permet d'appeler la ou les super-classes. Il n'y a pas de mot clé `super` car celui-ci serait ambigu. Il existe en effet plusieurs classes `super`. Cela a pour conséquence de lier fortement les classes entre elles. Il existe des techniques pour simuler `super` [PP96].

Comme Java, le C++ n'autorise le polymorphisme qu'avec les méthodes autorisées (adjectif `virtual`). Cela permet au compilateur d'optimiser les appels. Seules les instances partageant

les mêmes classes permettent le polymorphisme. La **généricité** permet de réduire cette limitation (voir page 91).

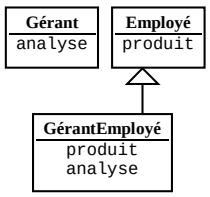
Il est possible de qualifier l'héritage. Une classe peut hériter en `public`, `protected` et `private`. Cela permet d'influencer les droits d'accès aux méthodes et aux attributs hérités. Un attribut public devient protégé s'il est hérité en `protected`. Il devient privé s'il est hérité en `private`.

	Public	Protected	Private
Héritage public	`Public`	`protected`	`private`
Héritage protected	`Protected`	`protected`	`private`
Héritage private	`Protected`	`private`	`private`

Pour interdire toute dérivation, il faut déclarer le **destructeur** de la classe en `private`. Comme son nom l'indique, le destructeur est une méthode appelée lors de la destruction de l'objet. En interdisant à une classe différente de détruire cet objet, toute dérivation est interdite. La classe ne peut plus être héritée.

Il est possible d'éviter l'héritage multiple par l'utilisation des méthodes génériques (voir page 91). Celles-ci sont des *modèles* de méthode dont certains types de paramètres sont inconnus (`template`). Lors de l'appel de la méthode avec un objet particulier, le code nécessaire à son utilisation est généré à la demande lors de la compilation. Par exemple, bien qu'il soit possible de décrire l'arbre d'héritage en utilisant l'héritage multiple, le développeur peut choisir une traduction équivalant à celle de Smalltalk.

Modèle C++ pour généricité



Le C++ étant un langage typé, il n'est pas possible d'utiliser la classe `GérantEmployé` comme une sorte de `Gérant`. Il n'existe pas de lien d'héritage entre eux. Normalement, le polymorphisme ne peut pas s'appliquer. On ne peut pas utiliser indifféremment un objet `Gérant` ou un objet `GérantEmployé` sans problème lors de la compilation.

Un service peut être rédigé pour recevoir en paramètre un objet `GérantEmployé` et appelant la méthode `analyse`. Ce même service peut être dupliqué pour recevoir un objet `Gérant`.

```
Déclare ServicePourGérantEmployé(paramètre X de type GérantEmployé)
Début
  appel de la méthode analyse du paramètre X
Fin
```

```
Déclare ServicePourGérant(paramètre X de type Gérant)
Début
  appel de la méthode analyse du paramètre X
Fin

Déclare A comme un objet de type GérantEmployé
Déclare B comme un objet de type Gérant
appel ServicePourGérantEmployé(A)
appel ServicePourGérant(B)
```

Les sources de ces deux services sont rigoureusement identiques, seul le type du paramètre diffère. Les méthodes génériques permettent de décrire ce service en omettant de préciser le type du paramètre.

```
Déclare Service(paramètre X de type inconnu)
{ appel de la méthode analyse du paramètre X
}

Déclare A comme un objet de type GérantEmployé
Déclare B comme un objet de type Gérant
appel Service(A)  génération d'une version
      pour GérantEmployé
appel Service(B)  génération d'une version
      pour Gérant
```

Lors de l'appel, une version spécifique est construite par le compilateur.

Lors de l'utilisation d'une méthode générique avec une instance `GérantEmployé` en paramètre, le code de la méthode sera généré et compilé. Une utilisation de la méthode avec une instance `Gérant` générera une nouvelle version.

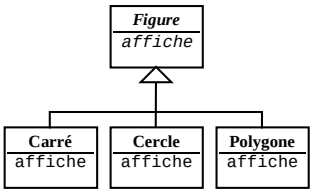
Déclaration C++	Compilation
`f<T>(T para)`	`f<GérantEmployé>(GérantEmployé para)` `f<Gérant>(Gérant para)`

Malgré l'absence d'héritage entre `GérantEmployé` et `Gérant`, des services peuvent utiliser indifféremment ces deux types d'objets. Ils permettent d'avoir un code extrêmement efficace car spécialisé à l'ensemble des types manipulés par le programme. Ils peuvent s'apparenter à des macros. L'utilisation de cette technologie ne permet pas de faire apparaître clairement l'interface que doit posséder un objet paramètre pour pouvoir être utilisé par une méthode générique. Il faut indiquer dans les commentaires l'interface attendue. Par exemple, il faut indiquer dans les commentaires de `Service` que le paramètre doit être capable de répondre à la méthode `analyse`.

Classe abstraite

Les **classes abstraites** sont des classes qui ne peuvent être instanciées. Elles représentent des ensembles abstraits d'objets.

Par exemple, une classe `Figure` n'est pas instanciable. On ne peut pas construire une figure sans informations supplémentaires. Elle peut être dérivée par les classes `Carré`, `Cercle` ou `Polygone`, qui représentent des classes concrètes.



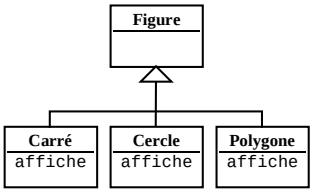
Il n'est pas possible d'instancier une `Figure`. En revanche, les attributs et les services partagés par les `Figure`s sont déclarés dans cette classe. Les méthodes spécifiques à chaque figure sont déclarées dans chacune d'elles. La méthode `affiche` de `Figure` n'est pas implantée dans cette classe, mais dans toutes les classes dérivées.

Une classe est abstraite à partir du moment où elle possède dans son protocole une méthode dont la définition est différée. Toutes les classes concrètes dérivées doivent spécifier cette méthode. Une classe dérivée ne déclarant pas cette méthode est encore une classe abstraite.

Smalltalk

La notion de classe abstraite existe en Smalltalk mais le langage ne permet pas de qualifier une classe d'abstraite.

Les classes dérivées doivent déclarer les services partagés dans l'ensemble des branches. La classe `Figure` ne possédera pas de méthode `affiche`. Les classes `Carré`, `Cercle` et `Polygone` déclareront cette méthode avec la même signature exactement.



Souvent, bien qu'étant inutile à l'exécution du programme, la classe `Figure` déclare la méthode `affiche` en affichant un message d'erreur lors de son exécution (`subclassResponsability`) ou en appelant le débogueur. C'est un bon style de programmation. La méthode ne devrait jamais être appelée. Elle permet aux utilisateurs de la classe `Figure` de connaître l'interface complète de celle-ci. Le mécanisme de résolution des appels de Smalltalk n'impose absolument pas cette approche. Si une classe dérivée oublie de déclarer la méthode `affiche`, l'erreur apparaîtra lors de l'exécution, signalant l'absence de la méthode.

Il faut normalement interdire la construction d'un objet abstrait. Il faut alors modifier la méthode de création d'instance de la classe pour qu'elle génère une erreur si on l'utilise.

Ce concept n'apparaît pas dans la syntaxe de Smalltalk. C'est une information qu'il faut indiquer dans les commentaires. Il n'est pas évident d'identifier ce concept lors de la lecture du code. Des tests unitaires sérieux doivent être rédigés pour trouver les erreurs classiques des classes abstraites, qui consistent à :

- oublier de redéfinir une méthode dans une classe dérivée,
- permettre la création d'un objet abstrait.

Smalltalk propose de déclarer, dans le corps d'une méthode abstraite, l'appel à `subclassResponsibility`. Cela permet à des analyseurs intelligents (`RefactoringBrowser`) de détecter les mauvaises dérivations des classes abstraites. Si une classe hérite d'une autre classe dont des méthodes utilisent `subclassResponsibility`, il faut vérifier que ces méthodes sont redéfinies dans les classes dérivées.

Java

Avec Java, tous les services doivent être déclarés dans une classe abstraite, mais pas nécessairement avec leurs implantations. La syntaxe impose d'ajouter l'adjectif `abstract` lors de la déclaration pour identifier cette classe comme abstraite. Certaines méthodes seront implantées, d'autres ne seront que déclarées.

Il n'est pas possible d'instancier une classe abstraite. Une classe concrète héritant d'une classe abstraite mais ne déclarant pas l'ensemble des méthodes abstraites sera rejetée par le compilateur.

C++

Le C++ permet également de déclarer une classe abstraite. À cet effet, il faut déclarer au moins une méthode comme abstraite. Une syntaxe particulière permet de distinguer la déclaration normale d'une méthode d'une méthode abstraite. Par exemple, pour déclarer la méthode `affiche` comme abstraite, il faut indiquer « `=0` » à la place du corps de la méthode, pour indiquer que celui-ci n'existe pas.

Toutes les classes dérivées d'une classe abstraite et n'implémentant pas toutes les méthodes abstraites sont considérées comme abstraites. Le compilateur interdira alors toute instanciation de ces classes. Ce type d'erreur apparaît à la compilation et non à l'exécution.

Ce concept n'apparaît pas aussi clairement que dans Java. L'existence d'une méthode déclarée comme abstraite permet de détecter une classe abstraite. Il est plus difficile de savoir si cette méthode est déclarée dans une super-classe. S'il existe une seule méthode héritée déclarée comme abstraite, la classe est abstraite. Il faut remonter la hiérarchie pour vérifier qu'il n'existe pas de classe abstraite dans les super-classes.

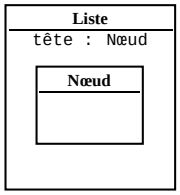
Classe contextuelle

De nouveaux types de classes ont fait leur apparition dernièrement. Suivant le contexte de déclaration, la classe a accès à des attributs supplémentaires qui ne lui appartiennent pas.

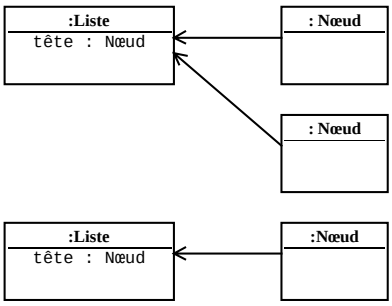
Classe « interne »

Une classe peut être déclarée au sein d'une autre classe. On l'appelle alors classe « interne » par opposition aux classes « globales ». La classe ainsi déclarée n'est accessible qu'à la classe possédante.

La création d'une instance d'une classe interne doit se faire dans une méthode de la classe globale externe. L'instance ainsi créée garde un lien avec l'instance génératrice, ce qui lui permet d'accéder à ses attributs. Par exemple, imaginons une classe `Liste` déclarant une classe interne `Nœud`. La classe `Liste` possède un attribut `tête`.



Les méthodes de la classe `Nœud` peuvent directement accéder à l'attribut `tête` de la `Liste` comme s'il s'agissait d'un attribut propre à l'instance `Nœud`. L'instance `Nœud` garde un lien avec l'instance `Liste` qui l'a générée.

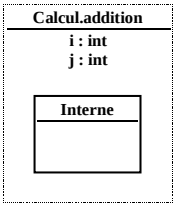


Chaque instance `Nœud` est en liaison directe avec l'instance `Liste` correspondante. La syntaxe du langage initialise automatiquement la liaison et offre un accès direct à tous les attributs de la classe externe. Une classe interne garde le contexte de sa création.

Classe de méthode

Il est également possible de déclarer une classe dans une méthode. De même que pour les classes « internes », une instance d'une classe de méthode a accès aux informations de la méthode l'ayant créée. Les variables locales sont accessibles en lecture dans les instances de la classe.

Par exemple, la classe `Interne` est déclarée dans la méthode `addition` de la classe `calcul`. Deux variables locales `i` et `j` sont également déclarées dans la méthode.



Les méthodes de `Interne` peuvent lire les variables locales `i` et `j` de la méthode `addition`. L'instance garde un lien avec son contexte de déclaration. Attention, seuls les accès en lecture sont possibles. Une classe interne ne peut pas modifier une variable locale.

Smalltalk

Smalltalk ne possède pas ces notions.

Java

Java a introduit ce concept dans la version 1.1. Il est également possible de déclarer une classe « interne » n'ayant pas de lien avec la classe contenante. Il s'agit alors d'une classe « externe », dont le nom n'est accessible que par la classe contenante. Ce concept permet de réduire les ambiguïtés de noms pour les classes techniques. Deux classes `Nœud` peuvent être déclarées dans deux classes distinctes, `Arbres`, et `Liste`, sans qu'il y ait d'ambiguïté. La classe `Nœud` de `Arbre` est différente de la classe `Nœud` de `Liste`.

Les classes de méthode peuvent être anonymes, ce qui réduit le vocabulaire et facilite la compréhension des sources. La déclaration de la classe de méthode s'effectue alors au cours de la valorisation d'un paramètre d'appel.

C++

Le C++ ne possède pas ce concept. Il permet la déclaration d'une classe dans une classe. Les instances des classes internes ne gardent pas de lien avec l'instance l'ayant générée. Ce concept permet de gérer les ambiguïtés de noms entre classes techniques. C'est un enrichissement de l'encapsulation et non un nouveau concept de classe.

Langage typé

Un **langage non typé** permet de manipuler un objet comme une entité. Lors de la demande d'un service, celui-ci sera recherché parmi l'ensemble des services de l'objet manipulé. S'il est présent, le code correspondant sera traité. Sinon, une erreur d'exécution sera générée. Cette catégorie de langage n'assure qu'un contrôle dynamique qui ne permet pas de détecter les inconsistances des types.

Le plus souvent, une méthode recevant un objet s'attend à ce que celui-ci soit d'un certain type afin de pouvoir utiliser tous les services attendus. Ce n'est pas une obligation. Un objet simulant tous les services d'un autre objet peut être utilisé à la place. Avec ce type de langage, il est possible de déclarer un *imposteur* et de l'utiliser partout où l'original est attendu. Tant que tous les chemins possibles du programme n'ont pas été parcourus, il n'est pas possible d'identifier une erreur aussi banale que l'utilisation d'une chaîne de caractères là où l'on attend un entier.

Les **langages typés**, au contraire, imposent de déclarer explicitement le type des objets manipulés. Il n'est pas possible d'appeler un service sans lui fournir un objet du type voulu. Même s'il existe une classe cherchant à en simuler une autre, il n'est pas possible de l'utiliser si elle ne dérive pas du type attendu. Ces langages permettent de détecter les erreurs lors de la phase de codage. L'interface prévue pour chaque service est plus rigide mais permet d'éviter des erreurs avant même la phase de tests. L'inconvénient de cette approche est qu'il faut parfois avoir recours à une conversion explicite de type (voir « Conversion de références », page 59).

Un type peut être une classe mais aussi un ensemble de symboles ; il s'appelle alors un **type symbolique**. Par exemple, les couleurs primaires peuvent être définies par les symboles Rouge, Vert, Bleu, et uniquement eux. Un type symbolique permet de déclarer les symboles valides pour un type donné et interdit toute rédaction invalide.

Smalltalk

Smalltalk est un langage non typé. Pour déclarer une référence sur un objet, on ne doit donner que son nom. Pour communiquer au développeur les informations de types attendues par les services, il faut ajouter cette information dans les commentaires. Le sens n'est pas apparent dans la syntaxe. La documentation d'un service indique les types d'objets attendus (par exemple : le premier paramètre doit être un entier). Il faut utiliser des normes de codage [SKU96].

Java

Java est un langage typé. Les services indiquent les types attendus pour chaque paramètre ainsi que pour le code retour. Une vérification est effectuée lors de la compilation et parfois lors de l'exécution du programme afin d'interdire les conversions erronées.

Java ne possède pas de types symboliques. Il faut les convertir en une constante numérique. Une valeur arbitraire doit être choisie par le développeur pour représenter les différents symboles. C'est pourquoi aucune erreur correspondant à l'utilisation de symboles ne peut être détectée par le compilateur Java.

C++

Le C++ est également un langage typé. Un service doit indiquer les différents paramètres et leurs types, éventuellement enrichis de l'adjectif `const` signalant que la méthode ne désire que les lire et non les modifier.

Le C++ offre un typage symbolique. Il est possible de déclarer un type comme un ensemble de symboles. L'utilisation d'un type symbolique est vérifiée à la compilation.

Sémantique

Il existe deux sémantiques distinctes dans les langages à objets : une sémantique par références et une sémantique par valeurs.

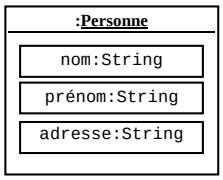
Dans une **sémantique par valeurs**, il est possible de manipuler la valeur d'un objet. Le passage par valeur est couramment utilisé dans les langages traditionnels. En C par exemple, une fonction recevant un entier obtient une copie de l'entier.

```
void f(int x)
{ x=99;
}
void main()
{ int i=100;
  f(i);
  // Ici i est égal à 100 et non à 99
}
```

Le paramètre `x` de la fonction `f` possède une copie de la valeur de `i`. La fonction `f` peut manipuler `x` sans modifier la valeur de `i`.

Il y a création d'une copie lors du passage de la valeur d'un objet. Les copies des objets sont automatiques. Si une méthode attend un paramètre avec passage par valeur, la copie de l'objet s'effectuera automatiquement. La sémantique par valeur se traduit par la gestion automatique de copie d'objet.

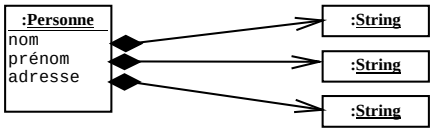
Dans une sémantique par valeur, un objet possède une collection d'attributs qui sont de véritables objets et non des références sur des objets. Les attributs sont regroupés dans la même zone mémoire.



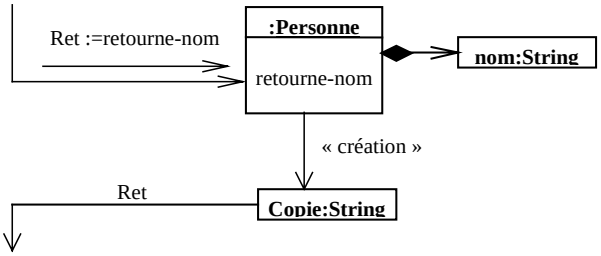
Les constructions et les destructions des objets sont automatiques. Elles entraînent la construction ou la destruction en cascade de tous les objets contenus. La duplication d'un objet entraîne la duplication de tous les attributs de l'objet. Cela permet de traduire partiellement la notion d'agrégation (un objet possédé par un autre objet).

Dans ce type de langage, une référence est considérée comme une valeur. Recevoir une référence en paramètre entraîne la copie de la valeur de la référence dans le paramètre. Cela entraîne une gestion des pointeurs par le programmeur avec les risques que cela comporte. En revanche, il est possible d'avoir une référence sur une référence. C'est utile par exemple pour rédiger une méthode qui inverse deux références.

Dans une **sémantique par références**, un objet est toujours manipulé par l'intermédiaire d'une référence sur celui-ci. Pour avoir un objet, on déclare une référence, puis on la valorise en demandant la création explicite de l'objet dans la mémoire. Ensuite, toutes les manipulations de l'objet s'effectueront par l'intermédiaire de cette référence. Elle peut posséder une valeur particulière indiquant qu'elle ne référence aucun objet (ou qu'elle référence un objet unique particulier, `nil` en Smalltalk). Avec ce type de langage, il n'est pas possible de retourner la valeur d'un objet mais uniquement sa référence. Tous les attributs contenus dans un objet seront en réalité référencés en son sein.

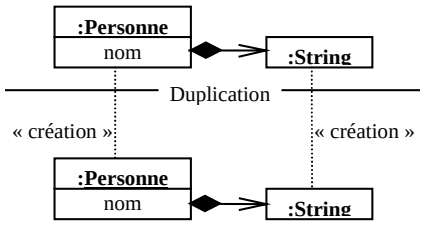


Pour traduire une sémantique par valeur avec une sémantique par référence, il faut explicitement utiliser une méthode de duplication d'un objet. Par exemple, un objet désirant retourner la *valeur* d'un de ses attributs doit retourner une copie de cet attribut.



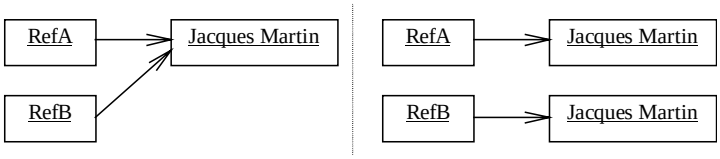
La méthode « retourne-nom » crée une copie de l'attribut `nom` et renvoie une référence sur celle-ci.

De même, lors de la rédaction de la méthode de duplication d'un objet, il faut demander la duplication explicite de tous les attributs.



Il existe une ambiguïté sur la signification de l'opérateur de comparaison entre objets. La syntaxe ou les méthodes doivent offrir des comparaisons distinctes. On doit pouvoir comparer deux références pour savoir si elles pointent sur le même objet ou sur des objets de même valeur.

Il peut exister deux objets personnes s'appelant « Jacques Martin ». Il faut pouvoir demander si l'on parle du même Martin, ou si l'on parle de personnes ayant le même nom.



Il faut pouvoir comparer les références `RefA` et `RefB` pour savoir si elles pointent sur le même objet, ou sur des objets de mêmes valeurs.

Smalltalk

Smalltalk utilise une sémantique par référence pour presque tous les objets. Il existe quelques exceptions : les types caractère et petit entier (inférieur à 32 767) qui, pour des raisons de performance, utilisent une sémantique par valeur, bien qu'ils s'en cachent. Dans Smalltalk, tout est objet, sauf l'élément le plus utilisé, c'est-à-dire la référence sur un objet. Il n'est pas possible d'avoir une référence sur une référence. Les pointeurs n'existent pas. Il est ainsi difficile de rédiger une méthode permutant deux références car ce ne sont pas des objets. Le langage propose des opérateurs différents pour comparer les valeurs des références sur les objets (== est un test d'identité) et les valeurs des objets (= est un test d'égalité). Par défaut, le test d'égalité est équivalent à un test d'identité.

Java

Java utilise explicitement les deux sémantiques. Les types primitifs comme `int` ou `float` utilisent une sémantique par valeur. Les objets utilisent une sémantique par référence. Java est en cela moins homogène que Smalltalk.

Attention, le type `String` est un objet. Il utilise une sémantique par référence.

Pour pouvoir utiliser les types primitifs comme des objets, il existe des classes équivalentes pour presque tous les types primitifs. Par exemple, pour le type `int`, la bibliothèque déclare la classe `Integer`. La version 1.2 de Java propose également une classe `Pointer` pour pouvoir les manipuler comme des objets (permettre d'avoir une référence sur une référence).

La gestion de la *valeur* d'un attribut doit se faire à la main, de même que la duplication de l'objet pour traduire une sémantique par valeur.

Java ne possède pas de pointeur, ce qui interdit la corruption du système.

C++

Le C++ utilise uniquement une sémantique par valeur. Tous les objets sans exception sont équivalents à un type primitif. Il est ainsi possible de construire une classe simulant parfaitement un type primitif.

Il est également possible de créer une nouvelle sorte de référence qui permette par exemple de coder une référence d'agrégation ou une gestion automatique de la mémoire, en utilisant un compteur de référence. Un compteur est ajouté à chaque objet. Lors de la valorisation d'une référence vers un objet, le compteur est incrémenté. Lors de la destruction d'une référence, Il est décrémenté. S'il est à zéro, cela veut dire qu'il n'existe plus de référence sur l'objet. Il est alors détruit. Le compteur de référence est géré automatiquement par l'objet « référence » créé à cet effet. Cet objet est souvent appelé *smart pointer* (pointeur intelligent).

Les constructeurs, les destructeurs et les copies d'objets sont générés automatiquement par le compilateur.

Référence constante

Pour limiter les utilisations possibles d'une référence, il existe le concept de référence constante. L'utilisateur de la référence peut seulement consulter l'objet référencé, mais ne peut pas le modifier. C'est une sorte de référence en lecture uniquement.

Les méthodes sont classées en deux catégories :

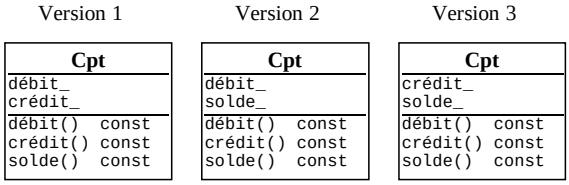
- les méthodes en lecture et écriture,
- les méthodes en lecture seule (ou méthodes constantes).

Une méthode constante peut être considérée comme un **attribut dérivé** de l'objet. Elle ne modifie pas l'objet. C'est une façon d'obtenir de l'information sur l'objet, que cette information soit un attribut ou non.

Pour un objet `Compte`, par exemple, il existe trois méthodes permettant d'obtenir des informations sur sa position :

- une méthode retournant le total du crédit,
- une méthode retournant le total du débit,
- une méthode retournant le solde.

L'objet `Compte` peut posséder deux des trois attributs, il en déduit le troisième, peu importe le choix des deux attributs sauvegardés dans l'objet. Ces trois méthodes ne font que consulter l'objet. Elles doivent alors être déclarées constantes.



Le choix des attributs présents dans l'objet ne modifie pas son interface. Le développeur est libre de modifier l'implantation des méthodes sans modifier l'interface. Ces trois méthodes sont équivalentes aux attributs.

En déclarant une référence en lecture uniquement, on ajoute du sens et le compilateur peut contrôler sa vérification. On maîtrise alors l'impact d'une méthode sur un objet. On sait si une méthode peut modifier une instance. C'est utile lors du déverminage. Lorsqu'on utilise une méthode d'un objet par l'intermédiaire d'une référence constante, seules les méthodes constantes sont accessibles.

Smalltalk

Smalltalk ne permet pas de limiter les utilisations des références sur un objet. Toutes les références permettent d'appeler une méthode qui peut lire et modifier un objet.

Java

Java ne possède pas encore de référence constante. Toutes les références permettent d'appeler une méthode qui peut lire et modifier un objet. Le mot clé `const` étant réservé, on peut supposer que ce sera le cas lors d'une prochaine version. La classe `String` de Java ne propose que des méthodes de lecture afin d'éliminer les effets de bords lorsqu'un objet est utilisé par d'autres.

C++

Une référence avec un adjectif `const` permet de n'autoriser que la lecture d'un objet. Pour ce faire, les méthodes doivent également être enrichies de cet adjectif si elles ne modifient pas l'objet. Seules les méthodes `const` sont utilisables avec un objet constant.

En C++, une méthode d'accès peut retourner la valeur d'un attribut ou une référence constante sur celui-ci. Elle permet de garantir qu'il n'existe aucun moyen de modifier un attribut à l'insu de l'objet.

Il existe des conversions d'une référence constante vers une référence non constante qui permettent de violer la protection offerte par ce concept.

Il est parfois utile de modifier un objet dans une méthode constante. C'est le cas lorsque l'objet mémorise un cache sur ces données. Par exemple, une méthode de calculs complexe peut être exécutée sur les données d'un objet sans le modifier. Cette méthode doit donc être constante. Mais il est tentant de garder le résultat de ce calcul au cas où le même service serait demandé par la suite. Il faut alors déclarer l'attribut mémorisant le résultat du calcul avec l'adjectif `mutable`. La méthode constante peut ainsi le modifier.

Par exemple, un objet mémorisant une liste chaînée peut en garder la taille dans un attribut. Pour optimiser le code, il peut être intéressant de ne pas maintenir la cohérence de cet attribut

lors de chaque manipulation de la liste. Dès que la liste est modifiée, un drapeau est levé, indiquant que la taille de la liste n'est plus valide.

Si par la suite l'utilisateur de l'objet demande à connaître le nombre d'éléments présents dans l'objet, le service constate que l'information présente dans son attribut n'est plus valide. Il lance alors l'algorithme lui permettant de calculer cette information.

Que faire du résultat ? Il faut le retourner à l'appelant, mais il peut être judicieux de le mémoriser dans l'objet pour accélérer les demandes suivantes équivalentes. Le drapeau est baissé, et le nombre d'éléments est mémorisé dans l'objet. Ensuite, ce résultat est retourné à l'appelant.

```
Demande du nombre d'éléments
Le drapeau est-il levé ?
Si non, retourner le dernier résultat
Si oui
  calculer le nombre d'éléments
  baisser le drapeau
  mémoriser le résultat
  retourner le résultat

Ajout d'un élément
modification de l'objet
lever le drapeau
```

Tant qu'aucune méthode ne modifie la liste, cette valeur est valide. Si une nouvelle demande intervient, le drapeau baissé informe de la pertinence de l'attribut. Cette valeur est alors immédiatement retournée. Il n'a pas été nécessaire de recalculer.

Cet algorithme permet d'optimiser le service. Maintenant, plaçons-nous dans l'optique de l'utilisateur de l'objet. La mécanique interne ne le concerne pas. Il demande à connaître la taille de la liste. Cette information ne modifie théoriquement pas l'objet. Ce n'est qu'une consultation. Il s'attend à utiliser une méthode constante.

Oui, mais cette méthode est capable de modifier la valeur de ses attributs. En effet, elle doit mémoriser le nombre d'éléments et modifier la valeur du drapeau. Une méthode qui modifie ces attributs n'est pas une méthode constante. Comment régler cette contradiction ? En utilisant l'adjectif `mutable` offert par le langage. Les deux attributs `drapeau` et `nbElement` utiliseront cet adjectif. Ainsi, les méthodes constantes pourront les modifier. Seuls les attributs `mutables` sont modifiables dans une méthode constante. Cet adjectif permet d'identifier les attributs de caches de l'objet. Ce sont des attributs techniques pouvant être omis dans une autre implémentation de l'objet.

Pureté syntaxique

Suivant leur niveau de pureté syntaxique, ces langages sont plus ou moins riches. L'interprétation de la pureté syntaxique d'un langage varie suivant les écoles. Dans la seule optique de la réduction maximale des éléments syntaxiques, un langage ne permettant que l'appel de méthodes est considéré comme pur. Cela n'est malheureusement pas possible. Il faut obligatoirement ajouter des éléments syntaxiques pour déclarer les variables locales et les successions d'appels de méthodes. De plus, il doit obligatoirement exister un élément sémantique permettant de trouver un objet global.

Certains langages ont cherché à réduire au maximum les éléments nécessaires à la syntaxe afin de faciliter leur apprentissage. D'autres, au contraire, enrichissent si nécessaire la syntaxe pour répondre aux besoins les plus courants.

Smalltalk

Smalltalk est pratiquement pur. Pour résoudre quelques difficultés, un monde minimal est offert, qui ne peut pas avoir été écrit avec le langage. Il doit être l'œuvre d'un créateur.

Par exemple, les variables globales sont possédées par un dictionnaire particulier étant lui-même global ! Lors de l'utilisation d'une variable globale, une recherche dans ce dictionnaire est effectuée. Le monde minimal de Smalltalk doit donc obligatoirement posséder ce dictionnaire, qui se référence lui-même. Il n'est pas possible de compiler une méthode utilisant une variable globale si celle-ci n'existe pas encore. La recherche des variables globales s'effectue lors de l'exécution.

Java

Java possède des éléments syntaxiques pour la gestion de boucles ou pour les conditions. Les variables globales ne peuvent exister que dans une classe. Cela correspond à des attributs des

métaclasses (voir page 9). Il n'existe pas de variables globales en dehors des classes. La syntaxe ou le contexte permettent d'identifier une variable globale par son nom, préfixé du nom de la classe. La validité de l'accès aux variables globales s'effectue lors de la compilation.

C++

Comme Java, le C++ possède des éléments syntaxiques de gestion du flux de traitement (boucle et choix). Il autorise la création de variables globales en dehors d'une classe. Le préfixe est alors à vide. La résolution de l'accès à une variable globale s'effectue lors de la compilation ou lors de l'édition de liens.

Structure de choix

Tous les programmes informatiques doivent offrir des structures de choix. Suivant la valeur d'un objet, un traitement particulier sera exécuté. Il existe deux structures principales de choix. La première est booléenne. Elle n'autorise qu'une alternative : vrai ou faux. La deuxième, plus générique, est une sélection en cascade. Une liste de valeurs ou d'expressions est testée. Selon les résultats des tests, les traitements correspondants sont exécutés. Les langages puristes évitent de devoir ajouter des éléments syntaxiques pour les choix.

Smalltalk

Pour être uniforme, Smalltalk déclare deux classes particulières, `True` et `False`, possédant les mêmes méthodes (`ifTrue:`, `ifFalse:`, `ifTrue:ifFalse:`, `ifFalse:ifTrue:`) mais exécutant ou non les traitements reçus en paramètres. Des instances uniques de ces deux classes, `true` et `false`, permettent de gérer les structures de choix sans sacrifier à l'uniformité du langage. Cette approche permet d'éviter une syntaxe de gestion de choix. Cette gestion est effectuée par l'algorithme de résolution de l'appel d'une méthode.

La méthode `ifTrue:` de la classe `True` exécutera le bloc de commande reçu en paramètre. La méthode `ifFalse:` n'exécutera pas le bloc de commande, mais se contentera de retourner à l'appelant. Inversement, une instance de la classe `False` n'exécutera pas son paramètre dans la méthode `ifTrue:` mais le fera pour la méthode `ifFalse:`.

Expliquons comment la structure de choix est possible. Toutes les opérations logiques retournent `true` ou `false`, instances de la classe `True` ou de la classe `False`. À l'instance retournée par l'opérateur logique, le développeur demande d'appliquer la méthode `ifTrue:` par exemple.

```
self dateDeNaissance < unAnimal dateDeNaissance  La condition
ifTrue: [^self]    Bloc à évaluer si la condition est vérifiée
```

Si la valeur de l'opération logique est l'instance `true`, le bloc de commande est exécuté. Sinon, il ne l'est pas. De même, si la valeur de l'opération logique est l'instance `false`, le bloc de commande n'est pas exécuté, sinon il l'est.

Il suffit de demander l'exécution d'une méthode `ifTrue:` ou `ifFalse:` à une instance `false` ou `true` pour que le mécanisme de résolution exécute ou non le bloc de commande reçu en paramètre. C'est l'algorithme de résolution, présent dans le moteur Smalltalk, qui effectue le choix. Il est ainsi possible de coder un test, sans ajouter d'élément à la syntaxe du langage. L'uniformité est respectée.

La sélection d'un traitement selon un ensemble de valeur s'effectue par une succession de `ifTrue:` ou par l'utilisation du polymorphisme.

Java

Java considère au contraire `true` et `false` comme des *valeurs* d'un type `boolean`. Il existe des structures syntaxiques pour les deux types de choix.

La structure de sélections multiples (`switch`) ne permet que l'identification de constantes. Il est possible de demander l'exécution d'un traitement si une expression est égale à une valeur, mais pas si elle est comprise entre telle et telle valeur. De plus, la valeur de test ne peut pas être calculée lors de l'exécution. Seules les constantes sont autorisées.

Pour contourner cette difficulté, il faut utiliser le choix binaire, en enchaînant en cascade les différents critères (`if then else`).

Il existe enfin un opérateur ternaire permettant, dans une expression, de faire un choix binaire. Le résultat de l'opérateur ternaire sera l'une ou l'autre des expressions suivant la valeur du test (`a<b ? a : b`). Cet opérateur attend trois arguments : un test et des valeurs possibles de retour.

C++

Le C++ utilise strictement les mêmes structures de test que Java.

Structures de boucles

Les **structures de boucles** sont-elles nécessaires à la syntaxe d'un langage ? Suivant la pureté de ce langage, des approches différentes sont utilisées.

Smalltalk

Smalltalk ne possède pas de structure de boucles. Celles-ci s'effectuent en appelant une méthode classique. Pour implanter les boucles, la méthode d'itération est contrainte d'utiliser la récursivité. La méthode s'appelle elle-même [BR97]. Pour l'utilisateur de Smalltalk, l'utilisation de la récursivité est invisible. Il utilise des méthodes classiques qui, en interne, utilisent la récursivité.

Le corps de la méthode `whileTrue:` est codé comme ceci :

```
whileTrue: aBlock
  ^self value  la valeur booléenne du bloc receveur est calculée
    ifTrue:
      [aBlock value. le paramètre est évalué
      [self value] whileTrue: [aBlock value]]  récursion
```

La récursivité interdit théoriquement les boucles infinies ou limite le nombre d'itérations. Le compilateur Smalltalk peut détecter les récursivités terminales à droite pour les convertir en boucle classique. Le langage est optimisé pour traduire uniquement ces méthodes par une véritable structure de boucles en interne. Pour l'utilisateur du langage, utiliser une boucle s'apparente à l'appel d'une méthode. En interne, la boucle est codée par une gestion du flot de traitement traditionnel.

Bien entendu, Smalltalk propose toute une panoplie de méthodes permettant de manipuler les boucles. Cela permet au langage de rester homogène. Avec Smalltalk, il y a une symbiose entre le langage, pauvre mais homogène, et la bibliothèque. La bibliothèque l'enrichit sans modifier la cohérence du langage.

Par exemple :

- La méthode `do:` de la classe `Interval` permet de traduire une itération entre deux valeurs (`1 to: 10 do: aBloc`).
- La méthode `collect:` renvoie une collection composée des éléments obtenus par l'action du bloc argument sur chaque élément de la collection receveuse.
- La méthode `select:` renvoie un tableau d'objet de la collection receveuse qui satisfont la condition exprimée par le bloc argument.
- La méthode `reject:` fait l'inverse de la méthode `select:`.
- La méthode `detect:` renvoie le premier élément de la collection receveuse qui vérifie la condition exprimée par le bloc argument.
- La méthode `inject:into:` retourne le résultat cumulé des traitements effectués avec chacun des éléments de la collection receveuse.

Java

Java a opté pour un enrichissement syntaxique permettant de modifier le flot des traitements par des sauts. Il existe trois syntaxes permettant de gérer les trois types de boucles les plus classiques :

- une syntaxe pour itérer d'une valeur à une autre en utilisant une expression d'incrémentation (`for`),
- une syntaxe permettant d'exécuter un traitement tant qu'une condition est remplie (`while`),
- une syntaxe permettant d'exécuter un traitement jusqu'à la validité d'une condition (`do while`).

Il existe également des éléments syntaxiques permettant d'interrompre le cours normal d'une boucle, pour demander à la boucle de sortir (`break`), ou au contraire de continuer à l'itération suivante (`continue`). Ces commandes permettent de sortir de plusieurs boucles imbriquées.

La commande `goto` n'existe pas.

C++

Le C++ utilise la même syntaxe de boucles que Java, mais cette syntaxe est moins riche en ce qui concerne les possibilités de sortie. Il n'est possible de sortir que d'un seul niveau de boucle. Sinon, il faut utiliser la commande `goto`, qui permet de continuer le traitement en tout point de la méthode.

Fonctions

Les **fonctions** permettent d'écrire des traitements en dehors de tout objet. Elles correspondent au développement procédural classique. Certains langages les refusent pour forcer l'utilisateur à penser objet.

Une fonction peut être considérée comme une méthode appliquée au programme. Pour exécuter une fonction, il faut auparavant lancer le programme. Cela peut être assimilé à créer une instance du programme. En théorie, une fonction n'est rien d'autre qu'une méthode appliquée à l'instance du programme. De même, une variable globale est un attribut de l'instance d'exécution. Les variables globales n'existent que pour chaque exécution du programme.

Les fonctions ne sont pas en contradiction avec le modèle objet. La classe est le fichier programme présent sur le disque. Le message initial est l'appel par le système d'exploitation de la fonction de démarrage du programme. L'exécution d'un programme est alors vue comme une instance. Il n'est pas possible d'appeler une fonction d'un programme sans l'avoir lancé auparavant. Exécuter un programme équivaut à créer une instance. Plusieurs instances du même programme peuvent être lancées simultanément. Les fonctions sont alors les méthodes applicables à chacune. De même, les variables globales sont les attributs de l'instance du programme. Un programme est un objet possédant ses attributs et ses méthodes.

Smalltalk

Avec Smalltalk, il n'est pas possible d'appeler un traitement sans indiquer d'objet destinataire. Pour simuler la notion de fonction, il faut utiliser une méthode de classe.

Les objets `BlockClosure` (voir « Traitements objets », page 84) permettent également de traduire le concept de fonction. Un bloc est un traitement dont l'exécution est différée. Il décrit un traitement n'étant pas forcément dépendant d'un objet. Les paramètres du bloc sont équivalents aux paramètres d'une fonction.

Java

Java interdit l'utilisation de fonction en tant que telle. Il est possible de rédiger des méthodes de classes pouvant se passer d'instance réelle. Ces méthodes sont vues comme des méthodes de la classe utilisant les attributs de la classe. Pour simuler la notion de fonction, il faut déclarer une classe sans attribut, et ne déclarer que des méthodes de classes.

C++

Le C++ autorise les fonctions, ce qui permet de récupérer les développements antérieurs rédigés en C. L'inconvénient de cette approche est que les développeurs risquent de mélanger deux paradigmes. Ils croient développer en objets alors qu'ils ne font que de la programmation procédurale.

Les fonctions servent aussi à ajouter des opérateurs à des objets sans les modifier, si l'on ne possède pas les sources par exemple. Comme on ne peut pas modifier une classe existante si l'on ne possède pas les sources, pour autoriser une écriture comme : `1 + Obj`, il faut, déclarer une fonction opérateur et non une méthode opérateur (voir « Redéfinition des opérateurs », page 53).

Retour de méthode

Une méthode peut retourner un objet ou une référence à un objet.

Retourner un objet veut dire que l'on renvoie la *valeur* de cet objet. Cette valeur est mémorisée dans un objet temporaire, différent de l'original. Il y a création d'une copie lors du retour. L'objet intermédiaire ainsi créé est détruit automatiquement par l'appelant lorsqu'il n'est plus nécessaire.

Retourner une référence veut dire que l'on retourne un pointeur sur l'objet. On retourne de quoi manipuler directement l'objet. Il n'y a ni copie, ni objet temporaire.

Smalltalk

Smalltalk utilisant une sémantique par référence, toutes les méthodes doivent retourner une référence sur un objet. Par défaut, une méthode retourne `self`, c'est-à-dire l'objet venant d'exécuter le traitement. On peut enchaîner les appels de méthodes. La sortie d'un bloc de traitement permet de sortir de la méthode l'utilisant (l'élément syntaxique est « ^ »).

Java

Pour interrompre une méthode, il faut utiliser le mot clé `return`. Dans une même méthode, il peut exister plusieurs points de sortie. Ce mot clé permet de retourner une *référence* sur un objet, ou la *valeur* d'un type primitif.

Il existe un type particulier `void`, indiquant l'absence de retour. Une méthode retournant un objet `void` est l'équivalent des procédures en Pascal. Un objet `void` ne peut être ni lu, ni écrit, ce que le compilateur vérifie. Les erreurs sont détectées avant l'exécution. Une méthode déclarée `void` ne pourra pas retourner de valeur.

C++

Le C++ possède le mot clé `return`, qui a une sémantique différente de celle qui est proposée par Java. Cela permet de retourner un objet et non une référence sur un objet. Il y a copie de l'objet indiqué dans le paramètre du `return`. L'objet temporaire ainsi créé est détruit le plus tôt possible par le compilateur.

Pour retourner une référence à un objet, il faut utiliser les pointeurs. Ceux-ci sont des objets comme les autres (des objets primitifs). Retourner un pointeur entraîne la copie de sa valeur dans un autre pointeur lors du retour.

Le type `void` permet de déclarer des procédures. C'est un type vide indiquant qu'il n'y a pas de retour.

Surcharge

La surcharge est la possibilité de déclarer plusieurs méthodes avec le même nom mais des paramètres différents. Il ne s'agit pas de redéfinir une méthode dans une classe dérivée, mais de proposer plusieurs services de même nom et de comportements différents. Il est possible de **réduire le vocabulaire**. C'est un confort syntaxique. Lors de l'appel d'une méthode, la résolution s'effectue suivant les paramètres de l'appel.

Par exemple, une classe `Chaîne` déclare une méthode `add` recevant un paramètre de type « chaîne de caractères » permettant de l'ajouter à la chaîne courante. Elle déclare aussi une autre méthode `add` recevant un entier. Cet entier est dans un premier temps converti en « chaîne de caractères » avant d'être ajouté à la chaîne courante. Les deux appels, celui nécessitant une chaîne comme argument et celui nécessitant un entier, sont différenciés par le type du paramètre reçu.

Smalltalk

Le nom d'une méthode en Smalltalk est décomposé en un verbe et des articles de liaison, ce qui permet d'avoir des méthodes de même racine dans la classe. La racine d'un nom peut être la même, mais les mots de liaison à ajouter entre les paramètres permettent de différencier les appels. Cette approche permet, comme la surcharge, de réduire le vocabulaire. Ce n'est pas une surcharge à proprement parler. Le préfixe est identique, mais pas les suffixes.

Par exemple, un objet peut posséder deux méthodes, appelées respectivement :

```
DessineDe:A:
DessineDe:A:En:
```

La première permet de dessiner une droite d'un point à un autre ; la seconde d'y ajouter la couleur. Ces méthodes sont utilisées comme ceci :

```
obj DessineDe: 10@10 A: 20@20
obj DessineDe: 50@75 A: 30@12 En: Rouge
```

Java

Java permet de surcharger une méthode avec le même nombre ou un nombre différent de paramètres si les types attendus diffèrent. Le type de retour peut être différent.

Par exemple, il peut exister une méthode `affiche` attendant un paramètre entier et une autre attendant une chaîne de caractères. Ces trois versions de la méthode seront rédigées séparément. Lors d'un appel, le compilateur choisira la bonne version en se basant sur le type du paramètre utilisé.

```
class A
{ public void affiche(int x);
  public void affiche(String s);
}

A p=new A();
p.affiche(1);
p.affiche(«abc»);
```

C++

Le C++ autorise les mêmes surcharges que Java. Le type d'un objet est un élément discriminant pour le choix d'une méthode. De plus, il autorise une surcharge implicite à l'aide de paramètres par défaut. Il est possible de fournir une valeur par défaut pour les derniers paramètres d'une méthode.

Par exemple, une couleur par défaut peut être déclarée pour l'affichage d'un rectangle. Il existera alors deux possibilités d'appel :

- fournir tous les paramètres du rectangle, couleur comprise ;
- ne fournir que les coordonnées du rectangle, une couleur par défaut étant utilisée.

Ainsi, une méthode peut être étendue sans avoir d'impact sur les développements existants. Vous pouvez ajouter un nouveau paramètre à une méthode sans modifier tous les appels à cette méthode.

Par exemple, vous rédigez un programme qui appelle à plusieurs endroits l'affichage d'un rectangle. Vous n'indiquez pas de code couleur, car lors de la rédaction, cette version n'existait pas. Par la suite, vous décidez d'ajouter un paramètre à cette méthode. Il est dorénavant possible d'afficher un rectangle en sélectionnant sa couleur. Votre méthode a changé son protocole d'appel. Dans un langage traditionnel, il faut retrouver tous les appels à l'ancienne version de la méthode pour y ajouter à la main le paramètre manquant. Avec le C++, il suffit d'indiquer une valeur par défaut et de recompiler le programme.

Redéfinition des opérateurs

Certains langages autorisent la **redéfinition des opérateurs** permettant d'étendre les types primitifs pour ajouter, par exemple, l'addition de matrices ou la soustraction de données floues.

Les opérateurs ne sont pas fondamentaux. Certains parlent de « sucre syntaxique ». Il est vrai qu'il est possible de se passer des opérateurs en utilisant toujours des méthodes avec un nom. Les opérateurs permettent néanmoins d'offrir une écriture plus traditionnelle pour les expressions. Il est plus facile de lire « `a+b+c`» que « `a.add(b.add(c))` ».

La notion d'opérateur est difficile à intégrer avec le modèle objet. Il faut obligatoirement identifier spécifiquement le type des deux objets intervenant dans l'opération pour pouvoir la résoudre. Le modèle objet autorise la sélection d'un traitement suivant le type d'un seul objet (voir « Polymorphisme » page 17). Il faut alors utiliser des artifices pour sélectionner le traitement selon le type du deuxième objet.

Par exemple, pour pouvoir additionner un entier et/ou un flottant, il faut identifier quatre combinaisons possibles :

- entier/entier
- entier/flottant
- flottant/entier
- flottant/flottant.

Pour trois types, il y a neuf combinaisons. Les langages offrent généralement six types primitifs, donc trente-six combinaisons.

Smalltalk

Smalltalk autorise une syntaxe particulière pour les opérateurs binaires. La syntaxe est proche de l'écriture traditionnelle. En revanche, le service est implanté par l'appel d'une méthode sur un objet. Pour ne pas ajouter de sémantique sur les opérateurs, il n'y a pas de priorité lors de la

résolution des expressions. Ainsi, « `1+2*3` » renverra « `9` », et non « `7` » comme le veut l'usage. Smalltalk ne permet pas de définir d'opérateur unaire, comme le moins unaire (exemple : `-a`).

Smalltalk n'étant pas typé, il utilise une technique appelée « double sélection » pour rédiger les opérateurs. Chaque type primitif possède différentes méthodes permettant l'addition de types différents. Par exemple, les classes `Integer` et `Float` possèdent les méthodes `addInteger:`, `addFloat:`, etc. Les méthodes d'addition de `Integer` et de `Float` sont rédigées ainsi :

```
Déclare la méthode '+' de la classe Integer
  attend un paramètre obj
  appelle la méthode addInteger sur obj

Déclare la méthode '+' de la classe Float
  attend un paramètre obj
  appelle la méthode addFloat sur obj
```

Ainsi, la sélection de `addInteger` ou de `addFloat` s'effectue suivant l'instance exécutant la méthode « + ».

Voyons comment ajouter un opérateur pour un nouveau type. Permettre « `date + entier` » est trivial. Il faut déclarer une méthode « + » dans la classe `Date`.

Autoriser l'inverse est plus complexe (`entier + Date`). Il faut ajouter une méthode `addDate:` à la classe `Integer`. Cela entraîne la modification d'une classe de la bibliothèque. Ce n'est pas un très bon style de programmation. En effet, lors de l'évolution de la bibliothèque Smalltalk, il faut extraire tous les ajouts et les réinjecter dans la nouvelle version. Le browser facilite cette opération grâce à la notion de projet.

Toutes les opérations des types primitifs sont valides. Par exemple, il existe une classe `SmallInteger` qui peut contenir un nombre compris entre + 32 767 et - 32 768. Si une opération retourne un résultat supérieur ou inférieur à ces valeurs, un objet `LargeInteger` est créé afin de garantir le résultat. Un objet `LargeInteger` peut posséder un nombre infini de chiffres (selon l'espace mémoire).

Java

Java n'autorise pas la redéfinition des opérateurs. L'arithmétique avec les types primitifs est traditionnelle. Il existe des règles de priorité.

Java ne protège pas les opérations avec les types primitifs. Elles doivent rester dans les limites de leur représentation en mémoire. La division par zéro engendre une erreur.

C++

Le C++ autorise la redéfinition des opérateurs à l'aide de deux approches différentes. La première consiste à déclarer les opérateurs comme des méthodes d'un objet ; la deuxième permet de déclarer une fonction recevant deux paramètres.

Le langage étant typé, la résolution des différentes versions des opérateurs se règle à l'aide du type des paramètres. Pour obtenir un opérateur d'addition entre un objet et un entier, il faut déclarer un opérateur en tant que méthode. Par exemple, l'objet `Date` acceptera une écriture du type « `date + 1` ». Pour accepter l'écriture inverse « `1 + date` », il faut déclarer une *fonction* opérateur (voir « Fonctions », page 47). En effet, il n'est pas possible d'ajouter des services à des classes primitives. Il n'est pas possible d'ajouter la méthode d'addition à la classe `int` et recevant un objet de type `Date`. Un opérateur fonction sera alors utilisé, qui recevra deux paramètres : un entier et une date.

Le choix de l'opérateur selon la combinaison des types des deux objets manipulés est effectué lors de la phase de compilation.

Le C++ autorise la redéfinition des opérateurs unaires et binaires. Tous les opérateurs sont redéfinissables sauf l'opérateur « point » (permettant l'appel d'une méthode) et l'opérateur ternaire (un choix binaire dans une expression). Les niveaux de priorité sont respectés.

Il n'y a pas de vérification des opérations pour les types primitifs. Seule la division par zéro engendre une erreur. Un résultat peut être hors limite, il sera alors tronqué.

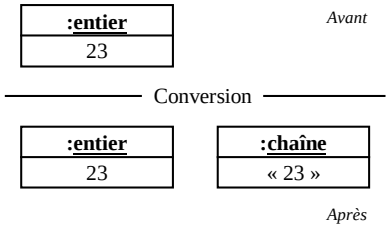
Conversion d’objets

Il est souvent utile de convertir un objet en un autre. Par exemple, convertir un entier en chaîne de caractères est une opération très courante. Les conversions permettent d’avoir une autre vue sur un objet. Une méthode qui retourne un objet ayant des valeurs similaires mais une autre classe est une méthode de conversion. Certains langages offrent une syntaxe particulière pour les conversions de types.

Le seul objectif de ce concept est de permettre une **conversion implicite** et non explicite d’un objet. Les conversions sont rédigées par le développeur dans une classe, mais l'appel est implicite. Ce concept ne peut exister qu’avec les langages typés.

Si une méthode attend un objet d’un certain type et reçoit un objet d’un autre type, le compilateur va chercher automatiquement le moyen de convertir le paramètre vers le type attendu par la méthode. Par exemple, si une méthode attend un nombre réel et que l’appelant fournit un nombre entier, une conversion d’un entier vers un réel est automatiquement générée.

Une conversion crée un nouvel objet ayant une valeur équivalente à l’original. Ces deux objets ne sont pas liés. La modification de l’un n’entraîne pas la modification de l’autre.



Smalltalk

Smalltalk ne définit rien concernant les conversions implicites. Cette impossibilité est due au langage qui n'est pas typé. Le concept de conversion exige un appel automatique suivant l'attente d'une méthode. Si une méthode attend un entier et qu'on lui fournit une chaîne, une conversion doit être appelée. Les méthodes de Smalltalk n'indiquent pas le type des paramètres qu'elles attendent. Il est donc impossible d'effectuer une conversion implicite, car aucune information ne permet de connaître le type attendu.

Cependant, il est parfaitement possible de rédiger des méthodes pour les conversions. Celles-ci retourneront un nouvel objet. Par exemple, la méthode `asString` de la classe `Array` va retourner une version chaîne de caractères de l'objet.

C'est au développeur d'appeler explicitement la méthode de conversion pour répondre aux attentes d'un service. Ce concept n'est pas normalisé. Il existe des règles pour la construction du nom des méthodes de conversion (utilisation du préfixe « `as` »), mais elles ne sont pas imposées par le langage. Cette convention doit être décrite dans les règles de rédaction des projets. Les conversions ne sont pas implicites.

Java

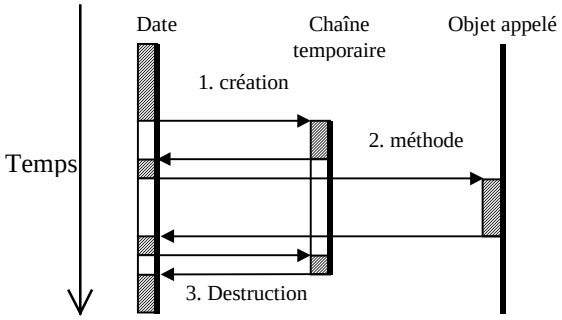
Java possède des promotions implicites entre les types primitifs (`int` vers `float`). Les conversions entre objets ne sont pas implicites.

Sun recommande l'utilisation de méthodes préfixées par `to` pour les conversions d'objets (exemple : `toString`). Comme en Smalltalk, la conversion d'un objet en un autre n'est pas normalisée. Il est possible de rédiger une méthode à cet effet, mais l'appel n'est pas implicite.

C++

Le C++ possède une syntaxe particulière pour les conversions. Il est possible de rédiger un **opérateur de conversion** pour une classe, ce qui permet au compilateur de choisir implicitement la conversion la plus adaptée lors de l'appel d'une méthode.

Examinons un cas d'utilisation. Un objet `Date` possède un opérateur de conversion en une chaîne de caractères. On envoie cet objet comme paramètre à une méthode s'attendant à recevoir une chaîne de caractères. Le compilateur va de lui-même utiliser l'opérateur de conversion pour traduire l'objet `Date` en une chaîne (1) avant de passer l'objet temporaire ainsi créé à la méthode (2). L'objet temporaire sera détruit au retour de cette dernière (3).



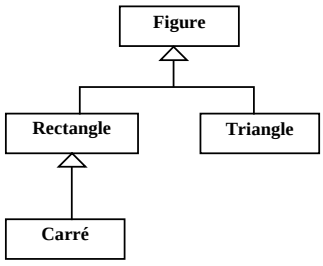
Il existe une autre technique de conversion en C++. S'il existe un constructeur ne recevant qu'un seul paramètre, le compilateur peut décider de construire un objet temporaire afin de satisfaire au type d'objet attendu par la méthode. C'est ce que l'on appelle une **conversion par construction**. L'objet temporaire sera construit par l'appel d'un constructeur et non par l'appel d'un opérateur de conversion. Il sera détruit dès qu'il ne sera plus nécessaire, c'est-à-dire au retour de la méthode appelée.

Il peut y avoir des conflits entre les deux méthodes de conversion. S'il existe plusieurs chemins pour convertir un objet, le compilateur le signale et demande au développeur de lever l'ambiguïté.

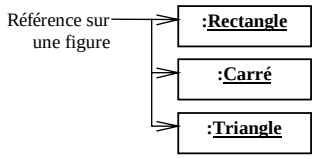
Conversion de références

On peut également convertir une référence. Dans certains langages, les références sont typées. Le développeur doit indiquer le type de l'objet référencé. Cela permet de vérifier les utilisations de la référence. Une référence peut pointer sur tous les objets dérivés du type indiqué.

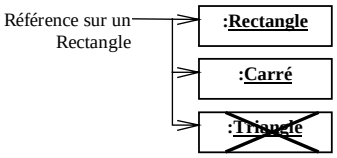
Par exemple, pour le modèle objet suivant :



Une référence sur un objet de type `Figure` peut pointer sur un `Rectangle`, un `Carré` ou un `Triangle`.



En revanche, une référence sur un `Rectangle` ne peut référencer qu'un rectangle et ses dérivés.



On peut désirer convertir une référence en une autre utilisant un type plus général (plus haut dans la hiérarchie des classes). Par exemple, on peut convertir une référence sur un `rectangle` en une référence sur une `figure`. Un `rectangle` étant une sorte de `figure`, la conversion ne pose pas de problème. Il y a compatibilité entre les références. Partout où l'on a besoin d'une référence sur une `figure`, le développeur peut fournir une référence sur un `rectangle`.

La situation inverse est plus complexe. Peut-on convertir une référence sur une `Figure` en une référence sur un `Rectangle` ? Tout dépend de l'objet réellement pointé. Parfois c'est possible, parfois non. La vérification se fait à l'exécution. Si l'objet pointé est un `Rectangle`, la conversion est acceptée. Sinon, une erreur est générée.

Smalltalk

Les références n'étant pas typées, il n'est pas nécessaire de convertir. Elles peuvent pointer sur tous les types d'objets. C'est lors de l'utilisation de l'objet qu'une erreur peut éventuellement être détectée.

Java

Java permet la conversion implicite ou explicite d'une référence sur un objet par une référence sur le même objet, mais d'un autre type. Il n'est pas possible de forcer une conversion erronée. L'objet pointé doit toujours correspondre au type de la référence. On le vérifie à l'exécution.

C++

C++ permet la conversion implicite des références s'il y a compatibilité vis-à-vis de l'héritage. De plus, il permet une conversion explicite dans l'autre sens, pour descendre l'arbre d'héritage. Dans ce cas, il existe plusieurs versions syntaxiques. Certaines ne vérifient pas l'objet pointé. Les performances sont meilleures, mais les risques d'erreurs plus grands. D'autres vérifient l'objet pointé lors de la conversion. Il est également possible de convertir une référence d'un

type en une référence d'un autre type, sans qu'il existe de relation d'héritage entre elles. Cette conversion est une grande source d'erreurs, mais permet des manipulations mémoire fines.

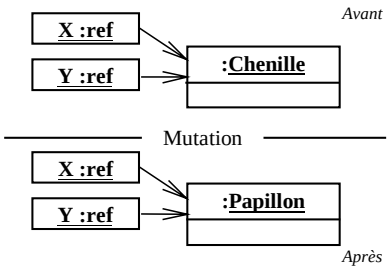
Il existe plusieurs syntaxes de conversion de référence :

- Vérification, à l'exécution, de la validité de la conversion (`dynamic_cast<>`) ;
- Vérification, lors de la compilation, de l'existence d'une relation d'héritage entre les types (`static_cast<>`) ;
- Suppression de l'adjectif `const` pour pouvoir modifier malgré tout un objet (voir « Référence constante », page 38) (`const_cast<>`) ;
- Conversion de n'importe quoi en n'importe quoi (`reinterpret_cast<>`). Aucune vérification n'est faite. Manipuler un `lapin` comme une `carpe` sera autorisé. L'exécution sera certainement erronée par la suite. Cette conversion permet quelques optimisations techniques difficiles à maîtriser.

Mutation

Il est parfois nécessaire de transformer une instance en une instance d'une autre classe. Par exemple, une instance de type `chenille` peut devenir une instance de type `papillon`. Il faut transformer celle-ci en maîtrisant toutes les références sur l'objet.

Il ne faut pas construire un nouvel objet du type `papillon`, puis détruire l'instance `chenille`. Les références sur la chenille en seraient invalidées. Il faut utiliser un mécanisme permettant de modifier toutes les références sur la chenille pour les faire référencer un papillon.



Smalltalk

Smalltalk possède une primitive système (`become:`) qui permet de traduire ce concept. Cette méthode est lente sur certaines implémentations car elle doit adapter toutes les références sur l'ancienne chenille pour les faire pointer sur le papillon. Le comportement exact de cette méthode dépend de l'implémentation. Visual Age IBM modifie tous les pointeurs vers la chenille pour les faire pointer vers le papillon (approche asymétrique), alors que

VisualWorks™ modifie également les pointeurs vers le papillon pour les faire pointer vers la chenille (approche symétrique).

Smalltalk autorise aussi le changement de classe d'une instance (`fixClassTo:` ou `changeClassToThatOf:`), si et seulement si, il y a compatibilité de format entre les classes source et cible. Cette approche est plus rapide car il s'agit d'une simple manipulation de pointeur. Attention : `changeClassToThatOf:` et `become:` ne peuvent se simuler l'un l'autre.

Java

Java ne possède pas ce concept. Il faut utiliser des astuces de programmation pour le traduire.

C++

Le C++ n'offre pas ce concept. Il existe plusieurs approches pour le traduire (cf. [PP96]). Ce n'est pas un concept du langage. Il faut utiliser les commentaires pour signaler son existence.

Gestion de la mémoire

La mémoire est une des ressources les plus utilisées dans un programme. Il y en a d'autres, comme les fichiers ouverts ou les connexions réseaux. Pour simplifier la gestion de la mémoire, certains langages possèdent un mécanisme de libération automatique des objets inutiles.

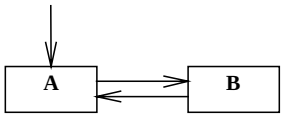
Un objet est inutile s'il est perdu, s'il n'existe plus de référence sur lui, même indirectement. Plus personne ne sait où il se trouve. Un mécanisme de **ramasse-miettes** (*garbarge collector*) peut se mettre en marche afin de parcourir toutes les références et en déduire les objets n'étant plus nécessaires, qui sont alors supprimés.

L'inconvénient de cette approche est que le ramasse-miettes prend un temps conséquent, et indéterminé. Cela peut être gênant pour les applications en temps réel. De plus, si une boucle importante passe son temps à construire un objet temporaire, le ramasse-miettes aura beaucoup d'objets à parcourir pour localiser tous ceux à détruire.

Certains algorithmes ajoutent à la récupération de la mémoire un regroupement des zones vides. En effet, les allocations traditionnelles peuvent créer une sorte de « gruyère » dans la mémoire (ou fragmentation). Il existe alors beaucoup de petites zones de mémoire libres éparpillées. Ce sont les « trous du gruyère ». Une allocation importante risque de ne pas être satisfaite alors que, mis bout à bout, l'ensemble de l'espace disponible serait suffisant. Des algorithmes regroupent ces trous pour n'avoir que le « gruyère » d'un côté, et le « trou » de l'autre.

De nombreuses recherches ont été menées sur les différentes manières d'implémenter un ramasse-miettes. L'algorithme le plus simple consiste à ajouter à chaque objet un compteur de références. Celui-ci maintient le nombre de pointeurs référençant l'objet. Lorsque la valeur du compteur est à zéro, on peut immédiatement détruire l'objet. Cet algorithme est simple, mais oblige à maintenir la valeur du compteur lors de toutes les manipulations des références, ce qui a un coût souvent excessif.

Cet algorithme possède un défaut majeur. Il est incapable de détecter des boucles. Imaginons en effet deux objets se référençant mutuellement.

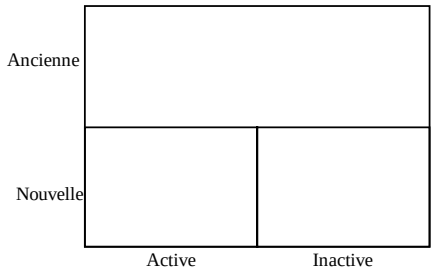


Un pointeur référence l'objet A. Maintenant, supprimons la référence sur cet objet. A et B continuent à se référencer ! Pourtant, il n'est pas possible de les retrouver en mémoire. Il n'existe aucun autre pointeur arrivant sur cet îlot. L'algorithme utilisant des compteurs de référence est incapable de détecter des îlots d'objets n'étant plus reliés à l'application.

Il a donc fallu inventer une autre approche. L'algorithme « Mark and sweep » parcourt tous les objets à partir d'une racine, et lève un drapeau sur chacun des objets traversés. Ensuite, une deuxième passe permet de supprimer tous les objets n'ayant pas de drapeau. Cette approche est peu coûteuse en mémoire, et évite de déplacer les objets. En revanche, il n'est pas possible de regrouper les zones de mémoire disponibles. La mémoire reste fragmentée. La phase de marquage prend un temps proportionnel au nombre d'objets vivants. La phase de balayage prend un temps proportionnel à la taille de la mémoire.

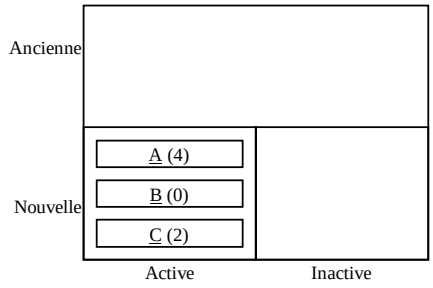
Cet algorithme en deux passes a été amélioré pour supprimer un des passages. L'algorithme de « Baker » propose de décomposer la mémoire en deux sections, une ancienne et une nouvelle. Les objets sont initialement alloués dans l'ancienne. Lorsque celle-ci est pleine, l'algorithme de ramasse-miettes entre en action. Il parcourt tous les objets comme précédemment, en partant d'une racine. Il les déplace dans la section nouvelle. Lorsqu'il ne trouve plus d'objet à déplacer, il peut déclarer la section nouvelle comme ancienne, et la section ancienne comme nouvelle. Tous les objets inutiles sont alors supprimés. Les nouveaux objets seront alloués dans la section ancienne, et ainsi de suite en alternance. Cet algorithme permet de regrouper les objets en mémoire. La fragmentation est supprimée. En revanche, il utilise deux fois plus de place que nécessaire, et la copie des objets prend un temps proportionnel au nombre d'objets.

David Ungar a eu l'idée de mélanger les deux approches précédentes. L'algorithme « Generation scavenging » repose sur l'observation que la plupart des objets « meurent » très jeunes ou très vieux. Par exemple, les objets temporaires liés à une méthode sont rapidement obsolètes. La mémoire disponible est divisée en deux zones — ancienne et nouvelle —, comme dans l'algorithme de « Baker ». La zone nouvelle est beaucoup plus petite que la zone ancienne. Elle est également divisée en deux portions : une active et une inactive.

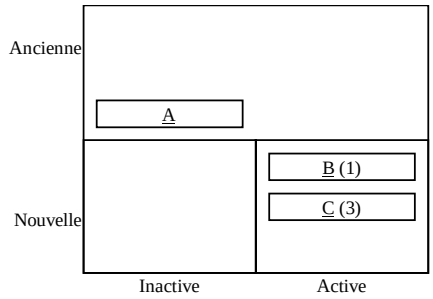


Tous les objets, sauf ceux de grande taille, sont créés dans la portion active. Quand il n'y a plus de place, les objets vivants sont copiés dans la portion inactive, qui devient alors la portion active. Cette opération s'appelle un *flip*. Un flip se déclenche toutes les quelques secondes et dure généralement 50ms.

Chaque fois qu'un objet est copié d'une portion à l'autre, on incrémente son compteur (lorsque l'objet subit le ramasse-miettes et ne meurt pas).



Quand le compteur excède un certain seuil, l'objet n'est plus copié dans la portion inactive, mais cette fois dans la partie ancienne.



La zone ancienne est gérée par l'algorithme « Mark and sweep », qui se déclenche quand cette zone se remplit au-dessus d'un seuil fixé. Ce n'est qu'à ce moment que peut se déclencher le « petit aspirateur ». Cet algorithme se concentre sur les objets fraîchement créés. Il laisse les aïeuls tranquilles, sauf si cela devient vraiment nécessaire.

Smalltalk

Smalltalk possède un ramasse-miettes. Il ne fonctionne pas en vraie tâche de fond, mais il est optimisé pour utiliser l'algorithme « Generation scavenging ». L'utilisateur n'a pas à se préoccuper de la récupération de la mémoire.

Depuis peu, il existe une méthode particulière appelée lorsque la destruction d'un objet intervient sur ordre du ramasse-miettes (`finalize`).

Attention : cette destruction n'est pas contrôlable dans le temps. Si un objet n'est plus référencé et si le ramasse-miettes n'est pas encore passé à l'action, la ressource utilisée n'est toujours pas disponible. Par exemple, si un objet ouvre un fichier et le referme dans sa méthode `finalize`, le fichier ne sera fermé que lorsque le ramasse-miettes sera passé à l'action. Il est probable que cette fermeture arrive plusieurs heures après la perte théorique de l'objet. Pour éviter ce problème, il faut rédiger une méthode permettant de libérer les ressources utilisées par l'objet. Elle sera appelée avant de perdre la référence sur l'objet.

Java

Java possède un ramasse-miettes qui fonctionne en tache de fond préemptif. Il en profite pour regrouper les trous. L'algorithme « Mark and sweep » est généralement utilisé, mais les différentes versions de machine virtuelle peuvent en choisir un autre.

Comme pour Smalltalk, il est possible de rédiger une méthode particulière (`finalize`), qui sera appelée par le ramasse-miettes lors de la destruction d'un objet. On peut par exemple fermer les fichiers ou libérer toutes les autres ressources.

C++

Le C++ n'utilise pas de ramasse-miettes. Il faut explicitement demander la destruction d'un objet, ce qui entraîne des risques d'oubli ou la destruction multiple d'un objet. Des outils du commerce permettent de détecter ces erreurs lors de l'exécution (comme `purify`). Il est impossible en C++ de réunir « les trous du gruyère ».

Il existe trois zones de mémoire distinctes. Lors du démarrage du programme, avant sa première instruction, les variables globales sont construites. Elles sont détruites avant le retour au système d'exploitation. Les variables locales sont présentes dans la pile. Leur durée de vie dépend de leur présence dans cette pile. Toutes les variables locales d'une fonction sont détruites lors de sa sortie.

De plus, il existe une zone mémoire appelée « tas » où l'utilisateur demande explicitement la création et la destruction d'un objet. C'est une des nombreuses sources d'erreurs.

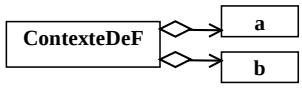
Le développeur peut oublier de détruire un objet, ce qui a pour conséquence de consommer petit à petit la mémoire. Le programme fonctionne, mais après une ou deux heures d'utilisation, la machine se met à ralentir. Le mécanisme de mémoire virtuelle se met en marche, et à la longue, le programme s'interrompt par manque de place.

L'utilisateur peut demander, par erreur, la destruction d'un objet déjà détruit. Pour ne pas ralentir ces algorithmes extrêmement critiques pour la vitesse du programme, aucun test n'est effectué. Une version de déverminage des allocations mémoire peut détecter ces erreurs.

Il est également possible de continuer à utiliser une zone mémoire considérée comme libre, mais les conséquences peuvent être très désagréables. Le programme va ainsi modifier un autre objet qui n'a rien à voir avec le premier. Ce type d'erreur est très difficile à détecter. Il existe des outils permettant de vérifier toutes les utilisations de la mémoire. C'est indispensable pour garantir la qualité d'un logiciel rédigé avec ce langage. Toutes ces erreurs entraînent une instabilité du programme, qui aboutit généralement à son plantage.

Pour résoudre ce problème de gestion de la mémoire, il peut être utile de recourir à des outils sémantiques. Les variables locales à une méthode lui appartiennent. Lorsque l'on sort de la méthode, les variables locales sont détruites. On peut concevoir les variables locales d'une méthode comme des attributs d'un objet représentant le contexte d'exécution de celui-ci (comme la classe `thisContext` en VisualWorks). L'appel d'une méthode équivaut à construire une *instance d'exécution* de celle-ci. Cela revient à réserver la place nécessaire à toutes les variables locales, le temps que la méthode s'exécute.

Appeler la fonction `f`, qui utilise les variables locales `a` et `b`, équivaut à demander la construction d'un objet `ContexteDeF` possédant deux attributs, `a` et `b`.

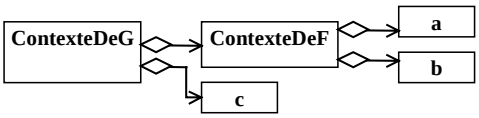


Le retour de la fonction consiste à détruire le contexte d'exécution.

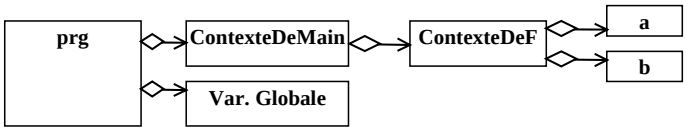
Un appel de fonction peut être décrit comme suit ;

- construit le contexte,
- utilise le contexte,
- détruit le contexte.

Maintenant, voyons ce qui se passe lorsqu'une fonction appelle une autre fonction. Comme pour ses variables locales, elle mémorise le contexte d'exécution de la méthode appelée. On peut dire qu'elle agrège le contexte d'exécution.



La fonction de démarrage est possédée par l'instance d'exécution de ce programme. De même, les variables globales sont possédées par lui.



Si l'on détruit le programme, on détruit également les variables globales, le contexte d'exécution de la fonction de démarrage et, en cascade, tous les objets.

On constate de ce qui précède que tous les objets doivent être possédés quelque part. Il existe toujours un objet qui est le dernier à référencer un autre objet. Tout objet est une agrégation d'un autre objet. Sinon, il doit être détruit.

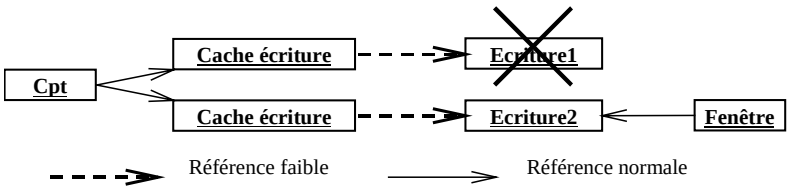
Si l'on possède un objet *référence* qui détruit automatiquement l'objet référencé lors de sa propre destruction, il n'est plus nécessaire de détruire explicitement un objet. Avec un objet représentant l'agrégation (la flèche composée d'un losange), il n'est pratiquement plus nécessaire de détruire les objets. Malgré l'absence de ramasse-miettes, la gestion mémoire est simplifiée. Pour savoir comment utiliser ce type d'objet, consultez l'ouvrage intitulé « La qualité en C++ » [PP96]. Il n'existe pas en C++. Cet objet peut être rédigé mais n'aura pas la puissance d'un ramasse-miettes.

Référence « faible »

Pour optimiser la vitesse d'exécution des programmes, les développeurs ont souvent recours à des caches. Des objets sont momentanément gardés en mémoire afin d'accélérer leurs accès et réduire ainsi l'utilisation des disques.

Un programme utilise généralement peu d'enregistrements à la fois. Il est donc capable de manipuler beaucoup de données, mais rarement toutes en même temps. Dans ce cas, il peut être judicieux de garder les objets les plus fréquemment utilisés, et de délaisser les autres. Un système de roulement permettra d'ajuster le choix des objets en mémoire à un moment donné.

Lors de la rédaction d'un cache mémoire, il est nécessaire de détruire les objets dans le cache lorsqu'ils ne sont plus utilisés depuis longtemps. Il existe des mécanismes permettant de supprimer un objet s'il n'est référencé que par des références faibles. Cette libération de la mémoire s'effectue en tâche de fond.



L'objet `Ecriture1` peut être supprimé de la mémoire car il n'est pointé que par des références faibles. L'objet `Cache écriture` le reconstruira si nécessaire. `Ecriture2` n'est pas détruit car il est également pointé par une référence classique.

Smalltalk

Smalltalk possède cette notion de référence faible. Une méthode permet de signaler qu'un objet devient faible (`makeWeak`). Toutes les références de cet objet deviennent alors des références « faibles ». Dans l'exemple précédent, les instances `Cache écriture` seront déclarées comme « faibles ». Le ramasse-miettes supprime un objet de la mémoire s'il n'est référencé que par des références faibles.

Visual Age Smalltalk d'IBM propose aussi des tableaux « faibles » (`WeakArray`). Tous les objets pointés par le tableau peuvent être supprimés de la mémoire par le ramasse-miettes.

Java

Java possède un mécanisme similaire dans la version 1.2. Une classe `WeakReference` permet d'utiliser des références faibles.

C++

Le C++ n'ayant pas de ramasse-miettes, il ne possède pas ce mécanisme. Il faut alors construire un objet *référence* et utiliser un algorithme de LRU (*Last Recently Used*) [LO77], pour supprimer les objets n'ayant pas été manipulés depuis longtemps.

Tableaux

Suivant les langages, la notion de tableau est syntaxique ou non. Un tableau est un ensemble d'objets. L'accès à l'un des éléments du tableau s'effectue à l'aide d'un indice. Les tableaux sont souvent présents dans les langages car ils permettent un accès rapide à l'information et ils sont faciles à implanter. On utilise souvent le terme de **conteneur** pour les objets ayant vocation à mémoriser différents objets. Un tableau est un conteneur, comme une liste chaînée ou une table de H-code.

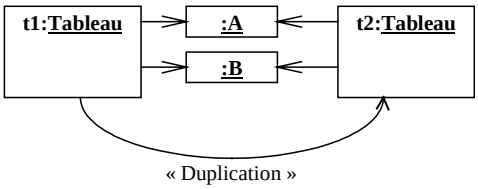
Smalltalk

Les tableaux Smalltalk sont des objets spéciaux (`variableSubclass`, `variablebyteSubclass`, `variableWordSubclass`, `variableLongSubclass`). Ils ne possèdent pas d'attributs nommés mais des attributs indexés. L'accès aux éléments du tableau s'effectue avec un index, à l'aide des méthodes `at:` et `at:put:`. Une erreur est générée si l'index est incompatible avec les dimensions du tableau. La méthode `size` retourne le nombre d'éléments du tableau. La taille du tableau est indiquée lors de la construction de l'objet (`new:`). La mémoire est organisée différemment des objets classiques afin améliorer les performances.

Smalltalk possède une syntaxe particulière (`#(...)`) pour construire les tableaux explicitement dans le code (lors de la compilation de la méthode). Cette syntaxe ne permet d'initialiser un tableau qu'avec des littéraux (des constantes). Il existe également dans la bibliothèque une hiérarchie de classes très riche permettant d'avoir des conteneurs. Chacun utilise un algorithme différent pour améliorer sa performance suivant les contextes d'utilisation (`Dictionary`, `Bag`, `Set`, `OrderedCollection`, `Array`, etc).

Java

Java possède une syntaxe particulière pour les tableaux. Comme pour Smalltalk, les tableaux possèdent des références sur des objets. La duplication d'un tableau n'entraîne pas la duplication des objets qu'il possède.



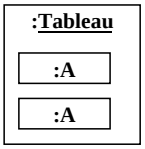
Copier le tableau `t1` ne copie pas les objets `A` et `B`.

- Les tableaux sont des objets mais il n'est pas possible d'en hériter.
- Les tableaux existent pour toutes les classes déclarées (les classes des tableaux sont générées automatiquement par la machine virtuelle lors de leurs premières utilisations).
- À l'exécution, il n'est pas possible d'accéder à un élément n'étant pas présent. Un index supérieur à la taille du tableau génère une exception.
- Une syntaxe particulière permet de construire un tableau en l'initialisant.

En plus de ces tableaux, la bibliothèque propose un arbre d'héritage pour implanter des conteneurs. Leur richesse est équivalente à celle de la bibliothèque de Smalltalk.

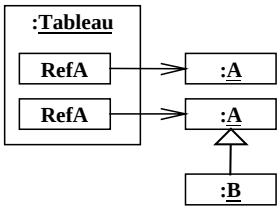
C++

Le C++ utilise une syntaxe particulière pour les tableaux (héritage du C ANSI). Les tableaux sont des zones de mémoire où les objets sont placés les uns à coté des autres. Le compilateur utilise la taille des objets du tableau pour retrouver un des objets. Les tableaux ne peuvent posséder que des objets de même type.



Le compilateur peut calculer la position d'un élément en multipliant la taille des objets du tableau par son indice.

Les références étant des types élémentaires assimilables à des objets, il est possible d'avoir un tableau de références.



Un tableau est pratiquement toujours assimilable à une référence. Il est possible d'utiliser une arithmétique sur les pointeurs pour accéder à un élément du tableau. Rien n'est testé. Il est parfaitement possible d'accéder à une zone mémoire n'étant pas un objet. Si un tableau possède deux éléments et que vous demandez à accéder au troisième, on ne vous dira rien. Le comportement du programme sera simplement imprévisible. Dans le pire des cas, il ne se passera rien. Vous croirez que votre programme est bon, alors qu'il est erroné. Dans le meilleur des cas, le programme plantera immédiatement, vous signalant le problème.

La librairie propose une collection de conteneurs (*Standard Template Library*). Elle n'utilise pas l'héritage mais la **généricité** (voir « Généricité », page 91). L'usage est volontairement analogue à l'utilisation des tableaux proposés par la syntaxe. Cette similitude permet de partager des algorithmes entre ces deux approches. Le choix du conteneur peut ainsi s'effectuer après une période de tests de performance, qui ne remet pas en cause les développements. Ces librairies sont extrêmement efficaces. Des indices O(*n*) permettent de connaître exactement le coût de chaque méthode. Un indice O(*n*) permet de connaître combien de consultations d'objets l'algorithme garantit. Par exemple, ajouter un élément au début d'une liste chaînée aura un indice O(*n*) de 1. En revanche, ajouter un élément au début d'un tableau aura un indice O(*n*) = *n*, où *n* représente le nombre d'éléments du tableau. L'algorithme doit en effet déplacer, un à un, tous les objets du tableau afin de libérer l'espace nécessaire à l'insertion d'un nouvel objet.

Réflexivité

Certains langages offrent un accès aux informations ayant permis de construire un programme (instance, classe, méthodes…). Ces informations sont plus ou moins riches et peuvent éventuellement être modifiées. Toutes les informations du langage sont présentes dans des objets manipulables (appelés **métamodèle**, ils correspondent à la description du modèle). Cela permet une introspection.

Il y a deux approches dans la réflexion :

1. La réflexion structurelle permet de connaître :
   - les structures des objets,
   - les classes,
   - les méthodes avec leurs paramètres,
   - les exceptions…
2. La réflexion comportementale permet de connaître :
   - le comportement des méthodes,
   - la pile d'exécution…

Pouvoir consulter le métamodèle permet de rédiger des outils de documentation automatique des programmes et des outils de conversion vers une base de donnée, ou d'offrir des navigateurs pour faciliter le déverminage des programmes.

Smalltalk

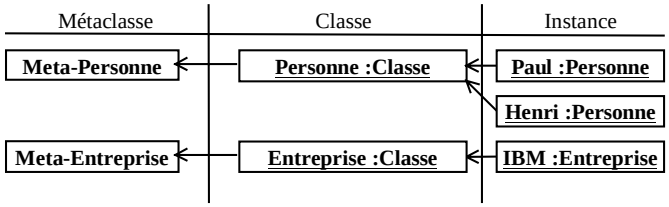
Smalltalk offre un accès complet au métamodèle. Il propose une introspection structurelle et comportementale. Le programme peut s'analyser complètement, jusqu'au code des méthodes ! Il peut même modifier dynamiquement une méthode. Il est possible d'intervenir en cours d'exécution sur les informations qu'il donne pour modifier les instances futures créées avec son aide. Le développement en Smalltalk s'effectue d'ailleurs par un dialogue interactif avec le métamodèle. On peut créer des classes et des instances associées. Il est possible de consulter la

liste des méthodes applicables à une instance, de consulter et de modifier sa classe... Parce qu'il est réflexif, tout est modifiable, ce qui permet de définir de nouvelle sémantique du langage.

Métamodèle	Objet
Lecture/Ecriture	Lecture/Ecriture

Les objets, les classes et même le métamodèle sont modifiables.

Smalltalk propose une métaclasse possédant la description d'une classe (les méthodes et les variables de classes). Il n'existe qu'une seule instance de la métaclasse. Lorsque l'on construit une classe, on construit indirectement une métaclasse anonyme.



Construire la classe `Personne` construit implicitement la métaclasse de la classe `Personne` dont la seule instance est la classe `Personne`.

Java

Java ne permet pas de modifier le métamodèle d'une instance. Il est possible de le consulter (version 1.1). Les informations sont limitées à l'approche structurelle et permettent de connaître :

- le nom de la classe,
- ses attributs,
- ses méthodes,
- la classe dont l'instance hérite,
- les interfaces implantées.

De plus, il est possible de demander la construction d'une instance en partant de sa classe ou d'appeler une méthode identifiée dans le métamodèle.

Métamodèle	Objet
Lecture	Lecture/Ecriture

C++

Le C++ offre un accès minimum au métamodèle grâce au RTTI (*RunTime Type Identification*). Il est possible pour tout objet ou type primitif de connaître le nom de la classe et les relations d'héritage. Il n'est pas possible de connaître l'ensemble des classes héritées par une classe mais il est possible de demander, lors de l'exécution du programme, si une classe hérite d'une autre.

Métamodèle	Objet
Lecture	Lecture/Ecriture

Des outils du commerce permettent d'analyser un programme C++ pour construire l'ensemble des informations du métamodèle et offrir une API de consultation de celui-ci. C'est à l'aide de tels outils que certaines bases de données objet sont conçues.

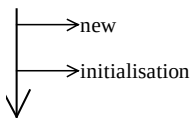
Constructeur

Tous les objets doivent être construits. Pour obtenir un objet, il faut lui réserver une place en mémoire, et initialiser ses attributs. Il existe une phase d'initialisation pendant laquelle l'objet est en chantier. La demande de construction d'un objet s'effectue par l'intermédiaire du métamodèle. Celui-ci peut appeler un service spécifique permettant d'initialiser l'objet. On a la garantie que les objets seront dans un état cohérent juste après avoir été construits.

Smalltalk

Avec Smalltalk, il faut rédiger des méthodes d'instances pour initialiser l'objet. Ces méthodes vont valoriser tous les attributs et éventuellement appeler une version héritée pour continuer avec les attributs qu'elles ne gèrent pas. La méthode de construction, comme toutes les méthodes, peut appeler une méthode de construction héritée. Cela permet d'initialiser seulement les nouveaux attributs et de réutiliser la construction de la classe de base.

Le constructeur, comme élément de fabrication d'un objet, n'est pas un concept présent dans le langage. Par défaut, la construction s'effectue en deux étapes : création de l'objet et appel d'une méthode d'initialisation. La construction d'un objet, en Smalltalk, se traduit par la réservation d'une zone mémoire pour celui-ci et par l'initialisation des références de tous ses attributs à la valeur `nil`. Ensuite, l'utilisateur doit appeler une méthode pour modifier les attributs.

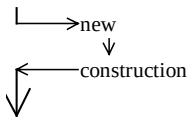


Rien n'interdit à un utilisateur de créer une instance sans passer par les méthodes de construction (s'il n'utilise que la méthode `new`).

La méthode `new` de la classe peut appeler une méthode `initialize` après la création d'une instance. Il n'est pas obligatoire de redéfinir la méthode `new`, mais cette approche permet de garantir une construction correcte des instances dans le cas où l'objet peut être initialisé sans paramètre. Sinon, il faut rédiger des méthodes de classes s'occupant d'initialiser correctement les instances.

Java

Java utilise un nom de méthode particulier pour déclarer les constructeurs. Les méthodes ayant le même nom que le nom de la classe où elles se trouvent sont des constructeurs. Ces derniers peuvent appeler un des constructeurs hérités et finir l'initialisation de l'objet. Un constructeur hérité est toujours appelé. Il est impossible de construire une instance sans utiliser de constructeur.



L'appel de `new` entraîne automatiquement l'appel d'une méthode de construction. Il est ainsi possible d'initialiser les attributs. La construction s'effectue en une seule étape, contrairement à Smalltalk.

La construction d'une instance Java est très coûteuse en termes de performance. Il est préférable de réutiliser une instance existante que d'en créer une nouvelle chaque fois. C'est également une bonne démarche pour réduire le travail du ramasse-miettes.

C++

Le C++ utilise également les méthodes de même nom que la classe pour identifier les constructeurs. Il est obligatoire d'appeler un constructeur des classes héritées avant de valoriser les nouveaux attributs. Si l'appel du constructeur hérité n'est pas explicitement indiqué dans le source, le compilateur le rajoute.

Destructeur

Certains langages à objets autorisent la rédaction d'une méthode qui sera appelée lors de la **destruction** d'un objet. Les ressources que l'objet utilise seront libérées sans que l'appelant ait à utiliser un service spécifique avant la destruction. Par exemple, un objet `fichier` pourra libérer sa ressource lorsqu'il sera détruit.

Smalltalk

Les dernières versions de Smalltalk permettent la rédaction d'un destructeur (`finalize`). Cette méthode doit libérer toutes les ressources utilisées par l'objet. Elle permet d'automatiser leur gestion.

L'appel est effectué par le ramasse-miettes. Lorsque l'objet n'est plus référencé, il est supprimé de la mémoire. Le moment précis où cette détection est effectuée n'est pas contrôlable. Un objet non référencé peut rester longtemps en mémoire sans que le ramasse-miettes intervienne. Celui-ci démarre lorsqu'une demande de mémoire n'est pas satisfaite, ou régulièrement pour détruire les objets récents. Les méthodes `finalize` ne sont appelées que lorsque le ramasse-miettes intervient. Tant qu'il ne fait rien, les ressources allouées ne sont pas disponibles.

Java

Java permet également la rédaction d'une méthode `finalize` qui sera appelée par le ramasse-miettes avant la destruction de l'objet. Elle permet de libérer les ressources mais ne permet pas de maîtriser le moment où celles-ci seront libérées. L'appel intervient sur ordre du ramasse-miettes et non lorsque l'objet n'est plus référencé.

Afin de maîtriser la destruction des objets, les classes offrent souvent une méthode de type `destroy` qui doit être appelée lorsque l'objet n'est plus utile. C'est le cas pour les *applets* Java. Les applets sont de petites applications intégrées dans une page HTML. L'appel de la

méthode `destroy` est équivalent à l'appel d'un destructeur. Il faut maîtriser parfaitement le moment où l'objet ne sera plus utilisé et être sûr qu'aucun autre objet n'ira l'invoquer après l'appel de `destroy`. Cette méthode n'est qu'un changement d'état de l'objet, et non sa destruction réelle. Des traitements peuvent par mégarde continuer à utiliser l'objet. Il y a ambiguïté entre la destruction logique de l'objet (appel à `destroy`) et sa destruction physique (ramasse-miettes).

C++

Le C++ permet la rédaction d'un destructeur. S'il n'est pas explicitement déclaré, un destructeur par défaut est généré par le compilateur. Il appellera le destructeur de chacun des objets agrégés. Le contrôle des ressources est ainsi parfaitement maîtrisé par l'objet. Il a la responsabilité des ressources qu'il utilise. L'appel du destructeur est parfaitement déterminé (même si cela n'apparaît pas explicitement). Les objets C++ peuvent être présents dans trois zones mémoire différentes : la mémoire globale, la pile pour les variables locales et le tas pour les allocations dynamiques. Pour chacune des situations, le langage définit le moment précis de la destruction des objets.

- S'ils sont présents dans la mémoire globale, ils sont détruits avant de sortir du programme.
- S'ils sont présents dans la pile, ils sont détruits avant de sortir de la méthode.
- S'ils sont présents dans le tas, il faut explicitement les détruire à l'aide de l'opérateur `delete`. Cet opérateur demande la libération de la mémoire, et appelle implicitement le destructeur.

La destruction d'un objet appelle automatiquement le destructeur. Il n'y a pas de risque de dissonance entre la destruction logique et la destruction physique. L'appel d'une destruction physique appelle la destruction logique.

Référence sur une méthode

Il est souvent nécessaire d'avoir une référence sur un traitement. Cela permet d'une part de fournir à une méthode le nom d'un traitement devant être exécuté, dans une boucle par exemple, et d'autre part de mémoriser des « appels en retour » (*call-backs*). Un objet peut appeler une méthode paramétrée si une condition particulière est remplie.

Par exemple, imaginons un objet `Compte` étant capable de signaler lorsqu'il se modifie. Un objet `Window` désire s'enregistrer auprès de l'objet `Compte` pour être prévenu de tout changement, et ainsi adapter la fenêtre en conséquence. L'objet `Compte` doit ignorer la présence de l'objet `Window`. Il va alors fournir un service d'enregistrement. Celui-ci attend comme paramètre un objet quelconque, et une référence sur une méthode de cet objet. Lorsqu'une modification intervient sur l'objet `Compte`, celui-ci appelle la méthode enregistrée pour l'objet référencé (la méthode de l'objet `Window` précédemment enregistrée). L'objet `Compte` ignore le type de l'objet enregistré. Il ne fait qu'appeler une méthode sur un objet. Un objet d'un autre type, `TousLesComptes` par exemple, peut s'enregistrer de même, auprès de l'objet `Compte`. Lors de la modification du `Compte`, l'instance `Window` et l'instance `TousLesComptes` seront prévenues.

Un autre exemple : une routine est capable de parcourir séquentiellement un arbre binaire. Le traitement à effectuer pour chacun des éléments parcourus peut être très variable. Pour éviter de devoir réécrire l'algorithme de parcours de l'arbre, il est souhaitable de procéder à une généralisation. Une routine de parcours d'arbre est écrite. Elle reçoit en paramètre le nom d'une méthode devant être exécutée à chaque itération.

Ce protocole n'est possible que si le langage permet de traduire en objet une « référence sur une méthode ».

Smalltalk

Smalltalk utilise le nom d'une méthode sous forme de symbole pour la référencer. Il faut alors utiliser une des versions de `perform:` pour demander de l'exécuter. Le nom des méthodes est

toujours présent lors de l'exécution d'un programme Smalltalk car les symboles sont partagés et uniques. Il est ainsi facile de traduire une chaîne de caractères en une référence sur une méthode. Rien ne garantit la validité de la référence. Elle peut identifier une méthode n'existant pas. Cette erreur apparaîtra lors des tests unitaires.

Java

Depuis la version 1.1 de Java, il est possible d'avoir une référence sur une méthode. Le métamodèle possède des instances de la classe `Method` permettant de représenter une référence sur une méthode. Des services de cette classe permettent d'invoquer la méthode pour un objet.

C++

Le C++ possède une syntaxe particulière pour décrire une référence sur une méthode (`Class::*`). Le nom des méthodes est perdu lors de la compilation. Seule la référence sur la méthode est disponible lors de l'exécution du programme. Il n'est pas possible de retrouver une méthode par son nom. Si c'est nécessaire, il faut construire un dictionnaire pour associer un nom avec une référence sur une méthode.

Traitements objets

Dans un langage à objets, un traitement peut être un objet. On peut le dupliquer et lui faire subir des modifications. Un tel langage permet de modifier les traitements en cours d'exécution.

Un objet de traitement permet de fournir en paramètre le corps d'une boucle par exemple. Une référence sur cet objet est livrée en paramètre. La méthode pourra alors demander son exécution.

Par exemple, un objet peut mémoriser une succession d'appels à des méthodes. Lors de la création de cet objet, les appels ne sont pas exécutés, mais juste mémorisés. Lorsque que le développeur le désire, il peut demander l'exécution de tous les appels indiqués dans l'objet.

Le corps d'une boucle est une succession d'appels de méthode. Le développeur peut construire un objet indiquant les appels du corps de la boucle et fournir l'objet à une méthode s'occupant de l'itération.

Smalltalk

Smalltalk possède un objet de traitement appelé « bloc » (classe `BlockClosure`) qui peut être construit avec une syntaxe particulière ([:i| i]). Il possède des variables temporaires et une méthode d'évaluation (`value`). Il peut également recevoir des paramètres (`value:`, `valueWith:`). C'est un objet de ce type qui est manipulé par les méthodes d'itération ou de choix. Un paramètre de type bloc permet de décrire le traitement à exécuter lors d'une boucle ou lorsqu'une condition est remplie. Il peut y avoir capture d'un environnement, c'est-à-dire capturer des variables utilisées par le bloc lors de sa déclaration.

Les méthodes ne sont pas des blocs. Elles sont de la classe `CompiledMethod` et appartiennent à la classe de l'instance.

Java

Avec Java, il n'est pas possible d'avoir un objet de traitement. Il faut utiliser les références sur une méthode.

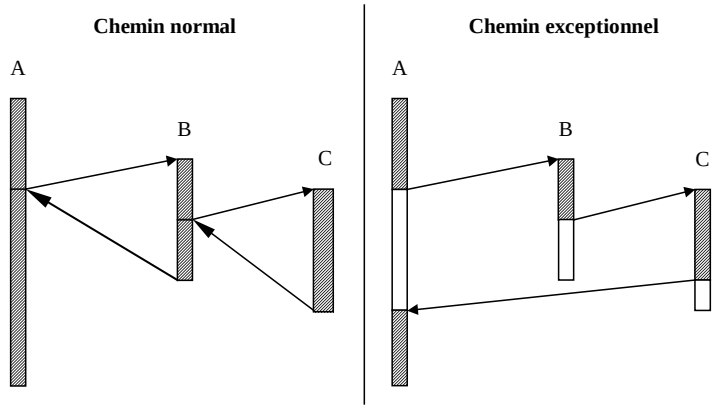
C++

Le C++ ne possède pas de classe de traitement. Il faut utiliser les pointeurs de fonctions et les pointeurs de membres qui peuvent être valorisés avec l'adresse d'une méthode. Pour demander l'exécution d'un traitement, il faut utiliser ces pointeurs. On n'a pas la souplesse de la création à la volée d'un traitement, mais on peut gérer les cas les plus courants en développement. Il est en effet très rare que l'on demande, en cours d'exécution, la génération d'un nouveau traitement.

Exception

Afin de faciliter la gestion des erreurs, un nouveau paradigme a été introduit avec le langage Ada. La portion de code à exécuter est encadrée afin de capturer toutes les erreurs en son sein. Si une méthode rencontre une erreur grave, elle peut générer une **exception**, c'est-à-dire l'envoi d'un objet en remontant la pile d'appels. Le retour normal de la méthode ne sera jamais effectué. Il s'agit d'un raccourci sur le flux normal d'exécution.

Par exemple, une méthode A appelle une méthode B qui appelle une méthode C.



Le cours normal de ces appels est représenté à gauche. Le chemin exceptionnel est représenté à droite.

Lors de la remontée de la pile, une méthode peut capturer une exception. L'objet ainsi envoyé est récupéré et la gestion de l'erreur peut être traitée. Cette approche permet de ne s'occuper que des cas généraux où tout fonctionne correctement et de déporter la gestion des erreurs dans un traitement exceptionnel.

Par exemple, une méthode désirant copier un fichier doit :

- ouvrir le premier fichier en lecture
- ouvrir le deuxième fichier en écriture
- tant que l'on n'a pas rencontré la fin du 1er fichier
- lire les données du premier fichier
- écrire ces données dans le deuxième fichier
- fermer le deuxième fichier
- fermer le premier fichier

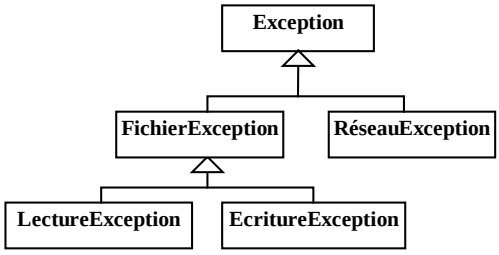
Lors de l'exécution de toutes ces étapes, il peut se produire une foule d'erreurs :

- fichier absent
- ouverture impossible
- lecture erronée
- écriture erronée
- fermeture impossible...

Plutôt que polluer le code par des tests à chaque étape, le développeur va l'encadrer pour qu'il soit géré par une exception. En cas d'erreur, quelle que soit l'étape en cours, elle est interrompue et le code correspondant à sa gestion est exécuté.

- détecte les exceptions
- copie le fichier
- si une erreur arrive pendant le traitement, ferme les deux fichiers.

La capture d'une exception s'effectue sur le *type* de l'objet émis. Par exemple, prenons l'arbre d'héritage suivant, représentant des exceptions :



Des instances de tous ces types peuvent être émises lors d'une exception. Un traitement doit demander la capture d'un objet en indiquant le type attendu. Si un traitement attend une exception de type `FichierException`, il capturera toutes les exceptions de ce type, mais aussi les exceptions des types dérivés. Il capturera des exceptions `FichierException`, `LectureException` et `EcritureException`.

Il faut capturer dans un premier temps les types les plus spécifiques, puis capturer les types de base. Par exemple, la méthode doit effectuer un traitement particulier lors d'une exception `LectureException` et un autre pour les exceptions `FichierException`. Elle doit dans un premier temps capturer une exception `LectureException` et, seulement après, capturer une exception `FichierException`. Sinon, après le traitement de l'exception `FichierException`, il ne lui sera plus possible de capturer l'exception `LectureException`. Elle aura déjà été capturée par `FichierException`. L'ordre de déclaration des exceptions à capturer est important.

Smalltalk

Smalltalk possède un mécanisme d'exception riche. Lorsqu'une exception est capturée, il existe cinq comportements de reprise du traitement :

1. Continuer le traitement après le code ayant généré l'exception. Le traitement de l'exception doit réparer le problème pour que le programme puisse continuer.
2. Recommencer le traitement au point où la capture de l'exception a été déclarée. Tout le traitement normal est recommencé.
3. Rediriger l'exception vers un autre traitement en changeant le type de l'exception. Cela permet de convertir une exception technique en une exception sémantique (« Lecture impossible » devient « Initialisation impossible »).
4. Refuser l'exception pour que celle-ci soit capturée par un autre traitement. Cela permet d'effectuer un traitement correctif partiel et de laisser l'exception continuer sa route.
5. Continuer le traitement après le bloc où la capture de l'exception a été déclarée.

Seuls les « blocs » peuvent capturer les exceptions. Il faut alors encadrer les méthodes dans des blocs pour pouvoir capturer les exceptions.

Java

Java utilise abondamment les exceptions. Une méthode générant une exception doit le signaler dans sa signature (`void traitement() throws IOException`). La signature d'une méthode est constituée de son nom, de ses paramètres et des exceptions qu'elle génère, ce qui permet au compilateur de vérifier que l'appelant la prend bien en compte. L'appelant peut décider de continuer à propager l'exception plus haut, ou au contraire la capturer pour réagir à l'erreur.

Le comportement d'une méthode vis-à-vis des exceptions faisant partie de sa signature, il n'est pas possible, dans une classe dérivée, d'ajouter de nouvelles causes d'exception si cela n'a pas été prévu par la classe de base. Seule une exception dérivée des exceptions de la méthode peut être émise lors de la surcharge.

Par exemple : une classe `Média` déclare une méthode `lecture`. Celle-ci indique qu'elle peut émettre une exception de type `FichierException`. Une classe `Video` surcharge la

méthode `lecture`. Cette nouvelle version de la méthode peut émettre toutes les exceptions dérivant de `FichierException`. Elle ne peut pas émettre une exception `VideoException` n'héritant pas de `FichierException`.

```
class Media
{ void lecture() throws(FichierException);
};

class Video extends Media
{ void lecture() throws(VideoException); // Erreur si VideoException
                                         // n'hérite pas de FichierException
};
```

Par défaut, les méthodes Java déclarent qu'elles ne génèrent aucune exception.

Indiquer dans la signature d'une méthode les exceptions que celle-ci peut propager permet au compilateur de vérifier qu'elles sont toutes capturées, ou que la méthode appelante indique correctement qu'elle propage, par effet de bord, les exceptions. On peut alors être sûr, lors de la compilation, que toutes les exceptions émises seront capturées.

Java offre la possibilité de définir un bloc d'instruction `finally` qui sera exécuté en sortie de bloc d'exception, qu'une exception soit levée ou non. On peut libérer les ressources de la méthode, ce qui correspond au destructeur de l'instance d'exécution.

Il existe des exceptions « système » qui sont générées par le programme lors d'erreurs graves, comme l'utilisation d'une référence vide, le débordement dans l'utilisation d'un tableau ou l'absence de mémoire. Ces exceptions n'ont pas à être déclarées explicitement dans la signature de la méthode. Elles sont considérées comme étant toujours possibles. Elles sont propagées par défaut.

Lors de la capture d'une exception, la pile d'appel est coupée pour remonter directement à la méthode appelante qui capture l'exception. Le contexte d'appel présent dans la pile lors de la génération de l'exception est perdu. Il n'est donc pas possible de connaître par exemple l'état de la pile avant que celle-ci ne soit coupée. Java construit une trace de l'état de la pile avant de la couper. Cela permet de localiser la méthode ayant émis l'exception alors que le contexte est perdu. Java garde une trace du cheminement de l'exception. Les informations mémorisées se limitent aux appels des méthodes. Les variables locales sont perdues.

Java définit une hiérarchie des exceptions générées par le système. Les nouvelles erreurs spécifiques doivent enrichir cette hiérarchie. Il faut utiliser la classe racine de l'arbre d'héritage des exceptions (`Throwable`) pour signaler qu'une méthode peut émettre toute exception.

C++

C++ propose une approche analogue à celle de Java pour vérifier les exceptions lors de la compilation (version officielle de la norme C++). Par défaut, une méthode peut générer toutes les exceptions (contrairement à Java). Pour indiquer qu'une méthode peut générer une exception système — manque de mémoire par exemple —, il faut le signaler dans la signature, la méthode en utilisant la classe racine de ce type d'erreurs (`std::bad_exception`).

Contrairement à Java, il est possible de ne jamais capturer une exception dans le programme. Cette absence n'empêche pas la compilation. L'erreur apparaîtra lors de l'exécution du programme. En final, si une exception remonte toute la pile sans jamais avoir été capturée, une fonction particulière est exécutée qui interrompt le programme (`terminate()`). Ce comportement peut être modifié (`set_terminate()`).

Les exceptions sont le seul moyen permettant de générer une erreur lors de la construction d'un objet. En effet, un constructeur ne peut pas retourner de valeur.

Il existe quelques exceptions normalisées dans le langage qui permettent de détecter la conversion erronée d'une référence ou la fin de la mémoire disponible. Malheureusement, il n'est pas possible de connaître la méthode l'ayant générée. Le C++ ne propose pas de mécanisme permettant de mémoriser l'état de la pile avant de la couper. Lorsque l'exception est capturée, il est trop tard pour connaître son origine.

Généricité

La **généricité** est un concept permettant de concevoir des classes ou des méthodes sans connaître l'ensemble des éléments manipulés. Proposé initialement avec le langage Ada, c'est un concept permettant des **optimisations** agressives. Il permet un polymorphisme statique au niveau compilateur. Tout est résolu lors de la compilation, plutôt qu'à l'exécution.

Par exemple, un algorithme de recherche d'un élément dans un ensemble peut être conçu sans connaître ni le type de l'élément recherché, ni le type de l'ensemble. L'algorithme doit parcourir tous les éléments de l'ensemble et les comparer un à un avec l'élément recherché. Le programmeur ne désire pas adapter, mais produire de façon sûre et automatique un composant d'une application. La notion de type paramétré permet de définir des composants suffisamment génériques afin d'en produire des versions chaque fois que nécessaire.

Lors de l'utilisation d'un service générique, le compilateur remplace les types inconnus par les types demandés ou déduits de l'appel. Une version spécifique de la méthode est générée.

La généricité ne s'applique qu'au source. Le compilateur doit être capable de transformer le source d'une méthode générique par un source équivalent mais spécialisé par l'appel.

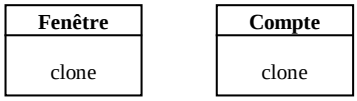
On obtient un niveau d'abstraction supérieur aux classes. Des comportements génériques sont décrits. Seule leur utilisation concrète entraîne la génération du code. Ce concept ne peut fonctionner qu'avec les langages typés car les types permettent d'identifier les différentes versions à générer. Un langage non typé n'a pas besoin de ce concept.

Il est très utile pour les conteneurs. Un conteneur est un objet qui contient d'autres objets. Il peut être implanté à l'aide d'un objet tableau, d'un objet liste chaînée, d'un objet « table de H-code », ou de tout type d'algorithme indépendant des objets manipulés. Il est possible, avec la généricité, de déclarer des classes conteneurs dont les classes des objets manipulés ne sont pas connues. Si l'application exprime le besoin d'utiliser un conteneur d'entiers par exemple, elle va utiliser la classe générique en lui fournissant comme paramètre le type `int`. Si, dans une autre classe, il est nécessaire d'avoir un conteneur de comptes en banque, ce type sera fourni comme paramètre à la classe générique. Il existera dans le code final de l'application deux classes distinctes pour les conteneurs, l'une pour les entiers, l'autre pour les comptes en banque.

La généricité est un mécanisme de déclaration qui doit être résolu lors de la compilation. Il n'est pas possible de demander la génération d'une nouvelle classe par une information connue lors de l'exécution. La généricité est une sorte de macro de génération. La différence essentielle avec les macros est que les types des paramètres sont typés et vérifiables par le compilateur. De plus, ce concept étant intimement lié avec le langage, il y a des rédactions licites, impossibles à proposer avec les macros. Avant l'existence de la généricité, les développeurs utilisaient des macros. Ils devaient explicitement instancier les classes avant de les utiliser. La généricité permet de les instancier lors de l'usage.

On peut paramétrer une classe générique par un type d'objet ou par une constante. C'est un concept permettant d'utiliser un polymorphisme statique. Ce concept permet de réduire les différences entre un langage objet typé et un langage objet non typé. Une méthode de même signature peut être appelée dans un service, même si les objets n'héritent pas entre eux. En effet, pour appeler un service sur un objet, il faut connaître son type. La vérification de l'existence d'une méthode ou d'un attribut pour un objet est effectuée lors de la compilation. Le compilateur utilise l'information de type pour connaître l'interface disponible pour l'objet référencé. Il est impossible d'appeler un service de même nom sur des objets différents s'ils ne partagent pas une classe de base.

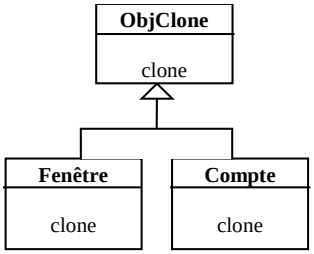
Par exemple, utilisons le modèle objet suivant :



Deux classes, `Fenêtre` et `Compte`, possèdent un service `clone` permettant d'obtenir une copie d'eux-mêmes. Ce service est déclaré dans les deux classes et chacune en modifie le corps pour l'adapter à son contexte. La méthode `clone` de la `Fenêtre` ne sera pas rédigée comme la méthode `clone` d'un `Compte`.

Imaginons un service recevant un objet et appelant sa méthode `clone`. Le langage étant typé, il faut rédiger une version du service recevant un objet de type `Fenêtre`, et une autre version recevant un objet de type `Compte`. Les corps des deux versions du service sont les mêmes, à l'exception du type du paramètre.

Si le modèle objet était organisé comme ceci :



Le service pourrait recevoir un paramètre de type `ObjClone` et n'être rédigé qu'une seule fois dans la classe `ObjClone`. Il n'est pas raisonnable de factoriser des objets qui n'ont en commun qu'un nombre restreint d'interfaces mais ne partagent pas d'information sémantique. Une `Fenêtre` n'a rien à voir avec un `Compte`.

La généricité permet de rédiger une seule fois le traitement, lequel sera automatiquement spécialisé lors de son utilisation. Finalement, le programme sera strictement identique à l'approche possédant les deux versions du traitement rédigées à la main.

Les langages non typés n'ont pas besoin de cela. En effet, l'appel du service `clone` sera résolu à l'exécution par la consultation des services disponibles pour l'instance manipulée. Lors de la compilation, aucune information ne permet de connaître la pertinence de l'appel. C'est à l'exécution que l'appel de `clone` sera aiguillé vers le bon traitement.

La généricité est très rapide car compilable. Les deux versions du traitement seront compilées séparément et optimisées individuellement.

Smalltalk

Smalltalk n'étant pas typé, ce concept n'existe pas. Il n'est d'ailleurs pas nécessaire. Tous les avantages offerts par la généricité sont présents dans Smalltalk. La technologie offerte est moins efficace. La généricité est un concept d'optimisation dont Smalltalk ne bénéficie pas.

Java

Java n'intègre pas la généricité. Des précompilateurs proposent l'ajout de ce concept à Java (http://www.cs.bell-labs.com/~wadler/pizza/), mais cela ne fait pas encore partie du langage.

C++

Le C++ possède ce concept et l'utilise abondamment dans les librairies de conteneurs (*Standard Template Library*). Les algorithmes sont à l'extérieur des conteneurs. Par exemple, la recherche d'un élément ne s'effectue pas en appelant une méthode d'une classe conteneur, mais en appelant une fonction générique en lui fournissant le conteneur concerné.

Les classes génériques sont souvent utilisées pour :

- des Smarts pointers (pointeur avec compteur de référence),
- des pointeurs d'agrégation,
- des conteneurs...

Multitraitement

Les systèmes d'exploitation multitâches préemptifs fonctionnent de la manière suivante : un traitement s'exécute dans un espace mémoire pendant un certain temps, puis le système d'exploitation l'interrompt pour donner la main à une autre instance d'exécution. Le microprocesseur est partagé entre les traitements. Cette approche s'appelle un multitâche préemptif. Il n'est pas possible de savoir quand un traitement sera interrompu.

Le **multitraitement** permet d'avoir plusieurs instances d'exécutions simultanées partageant le même code et les mêmes données. Les instances d'exécution partagent le même espace mémoire.

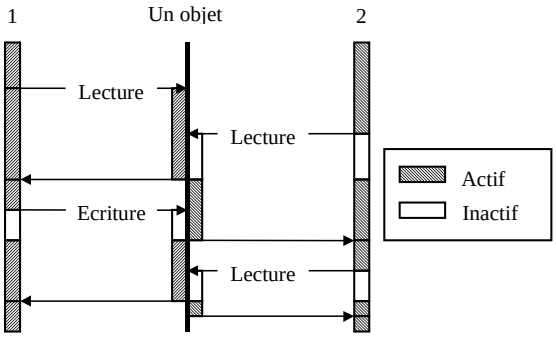
Il est différent du multiprocessus où chaque entité d'exécution est dans un environnement mémoire autonome et ne peut perturber les autres. Le multitraitement se nomme également processus léger (*thread* en anglais).

Les difficultés du multitraitement sont :

- les gestions des accès concurrents,
- la détection des étreintes mortelles (si deux objets s'attendent mutuellement avant de continuer, le programme sera bloqué indéfiniment. Ces situations sont très difficiles à identifier et rendent très complexe la rédaction des programmes).

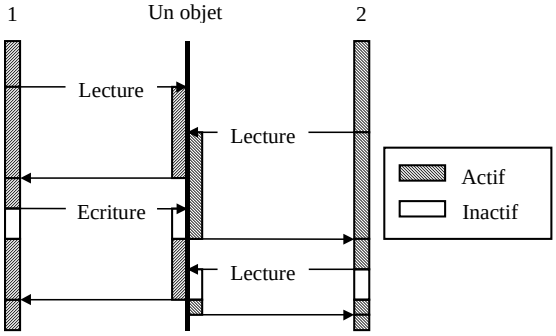
Il faut **protéger** les accès aux données. Pour ce faire, il existe plusieurs approches.

- Il est possible de bloquer, à l'aide d'un drapeau, tous les accès en lecture et en écriture à un attribut ou à une méthode. Si un traitement utilise en lecture un attribut, aucun autre traitement bloquant ne peut être exécuté tant que le premier n'est pas terminé. Il existe alors un drapeau par objet. Celui-ci est souvent appelé **sémaphore**.



Deux traitements accèdent simultanément à l'objet central. Ils doivent tous les deux accéder en lecture à l'objet, puis l'un des deux veut un accès en écriture. Tant qu'un traitement n'est pas terminé, aucun ne peut commencer. Les rectangles blancs indiquent les moments où une tâche est en attente.

- Pour offrir un accès simultané en lecture sur tous les attributs, il faut organiser un blocage particulier. Si une méthode désire lire un attribut, elle vérifie qu'il n'existe pas de tâche en écriture, puis incrémente un compteur qui est décrémenté lors de la fin du traitement de lecture. Lors d'un traitement modifiant l'objet, il faut dans un premier temps signaler l'écriture en cours, puis attendre qu'il n'y ait plus de traitement en lecture (compteur à zéro). Cet algorithme permet de n'avoir qu'une seule écriture à la fois, et, contrairement au précédent, des lectures simultanées.



  Dans cette situation, les deux traitements en lecture sont exécutés simultanément.
- Il est parfois intéressant d'avoir plusieurs mécanismes de blocage sur le même objet. Il peut exister des groupes de méthodes exclusives entre elles, mais pas entre les groupes. Par exemple, les méthodes permettant de manipuler les informations d'identité d'un `CompteEnBanque` seront exclusives entre elles. De même, les méthodes permettant de manipuler les `Ecritures` se protégeront des accès concurrents. En revanche, ces deux groupes de méthodes ne se bloqueront pas mutuellement. Il est possible de consulter les informations d'identification d'un compte pendant qu'un autre traitement ajoute une opération.

Smalltalk

Smalltalk permet le multitraitement non préemptif. Il permet de garantir l'atomicité de certains traitements. Ce système est **coopératif**. Chaque traitement doit indiquer explicitement lorsque qu'il en autorise un autre à prendre la main. Si un traitement dure très longtemps, les autres sont bloqués. Cette approche, également appelée « co-routine », permet de simplifier la gestion des accès concurrents. Un seul traitement est exécuté à la fois. Tant que celui-ci ne donne pas la main à un autre (`self yield`), il peut manipuler tous les objets sans risques.

Smalltalk propose des `Semaphore`s pour synchroniser les traitements. La classe `Promise` permet de lancer un traitement asynchrone et d'attendre la fin de son exécution.

Comme ce langage utilise une sémantique par référence, il faut cloner systématiquement les attributs au retour des accesseurs pour garantir que la consultation de l'attribut ne soit pas perturbée par un autre traitement. Sinon, deux traitements peuvent manipuler simultanément l'attribut à l'insu de l'instance propriétaire.

Par exemple, pour consulter le solde d'un `CompteEnBanque`, la méthode doit retourner une copie de la valeur dans un objet créé à cet effet. Ainsi, l'appelant pourra le consulter sans être perturbé par une modification de la valeur originale de l'attribut.

Java

Java possède dans sa syntaxe les éléments de gestion des différents traitements. Cette gestion est compatible avec une utilisation multiprocesseur. Il existe un sémaphore par instance et un par classe, qui fonctionnent à l'aide d'une exclusion totale. Une seule méthode à la fois peut s'exécuter sur une instance.

L'approche permettant d'autoriser les lectures simultanées n'est pas présente mais peut se développer [PP].

Les méthodes peuvent être `synchronized`. Elles demandent à avoir un accès exclusif sur l'instance ou sur la classe. Il est également possible de demander un accès exclusif lors d'une partie d'un traitement. Des services permettent de demander le blocage d'une instance.

Comme pour Smalltalk, étant donné la sémantique par référence du langage, il faut dupliquer les attributs dans les méthodes d'accès s'ils ne sont pas protégés en interne.

C++

Le C++ est compatible avec le multitraitement mais ne l'implante pas dans sa syntaxe. La gestion de la durée de vie des objets permet de construire un objet `Semaphore` permettant d'avoir les mêmes services que Java. Une instance de cet objet est déclarée dans un attribut. Les accès à l'objet sont bloqués. Lors de la fin de la méthode, le destructeur de cet objet permet de libérer automatiquement le sémaphore.

Paquetage

Le paradigme objet a ouvert un nouveau marché pour les composants logiciels. Des ensembles de classes sont proposés. Les facilités d'adaptation du modèle objet permettent aux développeurs de se focaliser uniquement sur leur métier. Ils achètent les éléments dont ils ont besoin et les intègrent dans leurs applications.

L'intégration de classes venant d'entreprises différentes peut entraîner une confusion dans les noms utilisés. Comment réduire le risque de doublon, empêchant l'intégration de nombreux composants ? L'approche la plus empirique consiste à préfixer le nom de toutes les classes d'un composant, ce qui alourdit le code. Le développeur ne gagne pas en lisibilité. D'autres approches plus ou moins normalisées sont proposées suivant les langages.

Smalltalk

Smalltalk ne propose pas de paquetage. Toutes les classes sont globales. Il peut y avoir des interférences lors de l'intégration de plusieurs composants. Il faut impérativement préfixer les noms des classes. Les variables partagées peuvent servir à empaqueter les variables globales, mais pas les classes.

Java

Java possède une notion de « paquetage » inspirée d'Ada. Elle permet de regrouper un ensemble de classes. Il est alors possible d'utiliser une classe en y ajoutant le nom du paquetage (`java.lang.String`). Il est également possible de demander l'importation d'un paquetage (`import java.lang.*`). Dans ce cas, seul le nom de la classe est nécessaire (`String`). Le compilateur recherche dans l'ensemble des paquetages courants celui qui permet de résoudre le nom de la classe utilisée. Les paquetages peuvent être hiérarchiques.

Pour plus de normalisation, et vu l'optique mondiale des paquetages sur Internet, Sun demande une organisation particulière pour leurs noms. Par convention, le plus haut niveau est réservé aux abréviations en majuscules des domaines les plus élevés d'Internet (`EDU`, `COM`, `GOV`, `FR`, etc.). Le niveau suivant doit correspondre à l'organisation éditrice du paquetage (`COM.worldcompany`). Ensuite, la hiérarchie est libre, mais devrait comporter le nom du paquetage avec éventuellement le nom du service concerné. L'organisation des répertoires contenant les fichiers compilés doit suivre la même hiérarchie que les paquetages : un répertoire pour « `COM` » et un sous-répertoire pour « `worldcompany` », dans lequel se trouve les fichiers Java compilés.

C++

Le C++ définit un équivalent aux paquetages de Java sous la forme d'un espace de noms (`namespace`) qui permet d'ajouter ou non les préfixes du paquetage utilisé. Il n'y a pas de recommandation sur le choix des noms de ces espaces. Ils peuvent être imbriqués.

Extensibilité

Au-delà de l'encapsulation, il est souvent nécessaire d'enrichir une classe existante. Dans ce cas, deux situations se présentent :

- Vous êtes l'auteur de la classe, il est normal que vous puissiez la modifier.
- Vous n'êtes pas l'auteur de la classe, vous n'avez pas les sources. Peut-on vous autoriser à la modifier ?

Il semble préférable d'interdire la modification d'un source dont vous n'êtes pas l'auteur, car il doit pouvoir la remettre en cause sans perturber les développements (principe de l'encapsulation). Une modification apportée à une classe existante complique énormément la prise en compte d'une nouvelle version. Il faut alors extraire les modifications apportées, mettre à jour la classe avec sa nouvelle version, puis réintroduire les modifications et les adapter si nécessaire.

Les principes d'héritage et de polymorphisme sont normalement prévus pour pouvoir modifier une classe sans la perturber. Toutefois, cela peut être nécessaire pour apporter une modification mineure à une classe existante.

Smalltalk

Smalltalk ne fait pas la différence entre les auteurs des classes. Il est toujours possible de modifier une classe. Les sources sont généralement disponibles, sauf pour certaines méthodes. Ces modifications de méthodes permettent, entre autres, de pallier des difficultés techniques dues à l'absence de l'héritage multiple. Ajouter une méthode ou un attribut à une classe s'effectue de façon incrémentale.

On peut ajouter une méthode ou un attribut à une classe dont il existe des instances. On ajoute au fur et à mesure tout ce que l'on veut sur une classe. Interdire la modification d'une classe appartenant à un paquetage dont on n'est pas l'auteur fait partie des règles d'usage.

Des outils comme Envy/Developer permettent d'ajouter, pour une application particulière, des méthodes à une classe définie dans une autre application. Cela permet une classification des méthodes suivant l'application. Il est ainsi facile d'étendre une classe et de bénéficier des mises à jours de l'environnement. Toutes les méthodes ajoutées seront alors réintégrées dans la nouvelle version de la librairie. Les modifications sont limitées à l'ajout de méthodes. Si le développeur ajoute un attribut ou modifie une méthode, le lien avec la classe originale est coupé. Il faudra alors, lors de la mise à jour de l'environnement, intégrer à la main les modifications des classes de la librairie.

Java

Avec Java, la déclaration d'une classe s'effectue dans un seul et même fichier. La classe est définie dans son entier. Il n'est pas possible d'ajouter à une classe existante une méthode ou un attribut. Il faut utiliser l'héritage pour cela. L'auteur a toute latitude pour modifier profondément la classe sans impacter les développements existants. L'encapsulation est alors respectée.

C++

Comme en Java, une classe C++ est déclarée dans son entier. Si l'on ne possède pas les sources, il est impossible de modifier une classe existante. L'encapsulation est, là aussi, respectée.

Persistance

Les objets sont créés dans la mémoire de l'ordinateur. Si l'on interrompt un programme, toutes ses données sont détruites. L'état courant d'un logiciel peut être sauvegardé pour pouvoir ensuite le reprendre dans l'état où il était avant son arrêt.

Smalltalk

Smalltalk permet de sauvegarder une image du contexte courant du programme. Il permet de sauvegarder dans un seul fichier toutes les informations courantes. La mémoire de l'ordinateur est ainsi sauvegardée. Lors de l'exécution d'un programme, il faut indiquer à la machine virtuelle de Smalltalk le nom du fichier « image » à charger.

Mis à part la demande explicite de la sauvegarde de l'image, cette technique permet de ne pas se préoccuper de la persistance des objets.

Java

La version 1.1 de Java permet de sauver les objets dans un flux. Celui-ci peut être une zone mémoire, un fichier, ou une communication réseau. C'est une technologie permettant la sauvegarde de la mémoire de Java. Un mécanisme de version permet d'intégrer les évolutions des classes. Si un programme Java lit un ancien fichier, le programme pourra réagir pour convertir les objets. L'adjectif `transient` permet d'indiquer qu'un attribut ne doit pas être sauvegardé.

Une API est en cours de normalisation pour offrir une persistance dans une base de données objet. Elle utilisera une persistance par attachement : tout objet référencé par un objet persistant deviendra persistant. Pour sauver un objet, il faut l'accrocher à un objet persistant.

C++

Le C++ ne possède pas de notion de persistance. C'est au programmeur de rédiger les méthodes de lecture et de sauvegarde d'un objet. Des outils permettent de sauver les objets C++ dans une base de données objet (ObjectStore™ ou O2™ par exemple). L'utilisation de ces outils est presque transparente pour le langage. L'utilisateur doit simplement indiquer lors de sa création si l'objet est persistant ou transiant.

PARTIE 2

Autres caractéristiques

La deuxième partie ajoute quelques points de comparaison pour avoir une meilleure vue des caractéristiques des trois langages. Il ne s'agit pas de concepts, mais de choix techniques spécifiques à chaque langage.

Syntaxe

Il existe deux approches principales dans les syntaxes des langages orientés objets : une approche du type fonction, proche de l'écriture mathématique traditionnelle, et une approche que l'on peut qualifier de « grammaticale», qui s'apparente au langage usuel.

Dans l'approche fonction, la syntaxe demande le nom de la méthode puis attend une liste de paramètres distingués par un séparateur (virgule, point-virgule...). En général, les paramètres sont encadrés par des parenthèses, afin de permettre à l'analyseur syntaxique d'en détecter le dernier. Parfois, un séparateur particulier doit être placé après le dernier paramètre. Par exemple, pour dessiner une ligne à l'écran, l'appel pourrait être :

```
ecran.dessine(10,10,100,100,Rouge)
```

Dans l'approche « grammaticale », les paramètres sont séparés les uns des autres par des mots de transition. Ces mots sont généralement choisis pour traduire l'appel en une phrase proche du français ou de l'anglais (suivant le choix de l'utilisateur). Pour le dessin d'une ligne, l'appel sera : `ecran dessineDe: 10@10 a: 100@100 en: Rouge`

Smalltalk

Smalltalk utilise l'approche « grammaticale ». Un ordre de résolution simple permet de différencier les commandes « grammaticales » des expressions arithmétiques. Le premier mot de la phrase indique l'objet manipulé (le receveur du message). Le second identifie le nom de la méthode (le verbe). Les mots de transition font également partie du sélecteur de la méthode. Le sélecteur de la méthode Smalltalk ci-dessus est le suivant : « `dessineDe:a:en:` »

Java

Java utilise une syntaxe du type « fonction ». Le caractère « point » permet de séparer l'objet manipulé du nom de la méthode.

C++

Le C++ utilise aussi une syntaxe du type « fonction ». Le caractère « point » ou les caractères « -> » permettent de séparer l'objet ou la référence sur l'objet de la méthode appelée. « -> » n'est qu'un raccourci syntaxique permettant d'éviter d'utiliser deux opérateurs.

Robustesse

Suivant les langages, certaines erreurs sont détectées. Le langage est plus ou moins tolérant. Lors de la présence d'une erreur, plusieurs comportements sont possibles :

- l'arrêt du programme avec éventuellement le lancement du dévermineur,
- l'appel d'un service particulier récupérant l'erreur et ayant la charge de rétablir la situation,
- la gestion d'une notion de **transaction** pour supprimer tous les traitements déstabilisant le programme.

Une transaction est un ensemble de traitements devant être exécutés entièrement ou pas du tout. Si une erreur intervient lors d'une transaction, toutes les modifications sont annulées. Les transactions fonctionnent correctement pour les bases de données. Ce concept doit tenir compte de tous les traitements possibles. Par exemple, il peut être nécessaire de reconstruire les objets détruits, de rétablir les connexions réseau ou de supprimer des fenêtres de l'interface utilisateur...

Tous les traitements doivent pouvoir être annulés lors d'une transaction. Les transactions limitées aux données ne sont pas suffisantes. Il faut un concept de transaction gérant toutes les ressources.

Par exemple, une méthode ouvre un fichier, puis demande des informations sur un réseau. Si celui-ci ne fonctionne pas, le mécanisme de transaction doit fermer automatiquement le fichier.

Aucun langage objet ne possède le concept de transaction.

Smalltalk

Smalltalk détecte toutes les erreurs techniques et affiche une boîte de dialogue. Lors de la fermeture de cette dernière, le programme n'est pas interrompu mais reste dans un état éventuellement instable.

Cette approche peut être gênante pour une application en production. Si une erreur grave arrive, faut-il sauver les données au risque de détruire leur intégrité ou faut-il au contraire tout arrêter et perdre ainsi les derniers traitements effectués ? Il est possible cependant de modifier le comportement par défaut pour certaines erreurs.

Les exceptions permettent de capturer les erreurs générées lors de la valorisation d'un « bloc ».

Java

Toutes les erreurs techniques sont détectées :

- utilisation d'une référence nulle,
- débordement dans l'utilisation d'un tableau,
- division par zéro...

Elles génèrent une **exception** qui peut être capturée par le programme. Java est un langage très robuste.

De plus, Java est protégé. Le byte-code généré possède une sémantique forte. Par exemple, il existe deux bytes-codes différents pour l'affectation d'un entier et d'un pointeur. Pour le microprocesseur, le traitement est le même. Il s'agit de modifier 64 bits à une adresse mémoire. Le code machine de ces deux affectations est le même. Java utilise deux codes différents pour pouvoir vérifier qu'il n'existe pas de code utilisant un entier comme s'il s'agissait d'un pointeur. Avant d'interpréter une méthode, un test rigoureux est effectué pour interdire la présence d'un byte-code vérolé.

Par exemple, le byte-code référence les méthodes par leurs noms et non par un décalage numérique, ce qui permet de vérifier sa cohérence. Il n'est pas possible d'écrire un byte-code accédant à un attribut privé. Il n'y a pas d'accès direct à la mémoire. Tout passe par une référence intermédiaire interdisant tous les accès involontaires. Le vérificateur de byte-code analyse tous les chemins possibles du programme et vérifie la cohérence du programme (la pile est-elle nettoyée avant de sortir d'une méthode ? Les zones mémoire sont-elles utilisées en respectant toujours le même type ? Les restrictions d'accès sont-elles respectées ?

La dernière barrière de protection est présente dans les API natives. Suivant le contexte, un code Java peut, ou non, accéder au disque dur du poste. Un programme Java est dans une sorte de bocal hermétique lui interdisant d'en sortir. Il est théoriquement impossible d'écrire un virus avec Java (les quelques failles identifiées ont été corrigées… jusqu'aux prochaines).

C++

Le C++ ne détecte pratiquement pas d'erreurs. La division par zéro est identifiée et génère un signal qui est capturable par le programme. Quelques conversions invalides peuvent générer une exception. Le C++ n'est **pas robuste**. Il est facile de casser sérieusement le programme sans que l'on puisse identifier le problème. Les programmes doivent être rédigés avec beaucoup de rigueur.

Compilation

La qualité d'un langage de développement se juge selon plusieurs critères pas toujours conciliables. Un bon langage doit permettre un développement rapide, une évolution facilitée et une **exécution efficace**. Les langages orientés objet cherchent à répondre essentiellement aux deux premiers points. Concernant la vitesse d'exécution, plusieurs approches s'affrontent.

Smalltalk

Un développement incrémental est possible par ajout successif de nouvelles classes et de nouvelles instances (les programmes Smalltalk s'effectuent par un enrichissement successif du modèle objet). Il est également possible de sauver l'image courante du monde ainsi créé, ou de repartir avec un nouveau monde possédant le minimum nécessaire. Il est délicat de modifier une classe s'il en existe une instance quelque part. Seuls l'ajout et la modification d'une méthode sont sans danger, car ils n'impactent pas les instances existantes.

Smalltalk est initialement un langage interprété (disponible sur de nombreux environnements). Pour améliorer la vitesse d'exécution, certaines méthodes critiques ont été compilées pour les différentes machines cibles. De plus, le code source des méthodes est compilé pour être traduit en un code intermédiaire appelé byte-code. Celui-ci est interprété par une machine virtuelle. Il n'est pas nécessaire de recompiler l'ensemble des bytes-codes si une classe est modifiée.

Au-delà de cette semi-compilation, le choix du langage de ne pas être typé, l'oblige, pour chaque appel de méthode, à rechercher pour l'instance manipulée si la méthode est disponible. Ce test est effectué la première fois. Ensuite, un accès direct est mémorisé dans un cache de la classe. Malgré des algorithmes efficaces, la vitesse d'exécution est considérablement réduite. Il existe des compilateurs « Just-In-Time » permettant de générer un code natif lors de l'exécution. Ce n'est pas important pour des interfaces utilisateur, car ce dernier ira toujours plus lentement que la machine. C'est plus grave pour les traitements lourds.

Java

Java ne permet pas d'ajouter une classe lors de l'exécution. Il faut dialoguer avec un compilateur pour demander de construire un monde pour l'application (construire des classes).

Le dialogue s'effectue à l'aide d'un fichier texte et d'une syntaxe particulière. Cette syntaxe disparaît complètement dans le monde ainsi créé (pas de réflexivité comportementale). Le programme est compilé et n'a plus la connaissance du source l'ayant généré. Ce dialogue s'effectue avant l'exécution.

Java permet d'avoir des morceaux de monde partiellement décrits. Chaque création de classe entraîne la création d'un morceau de monde indiquant les autres classes qu'il utilise ainsi que la classe dont il hérite. Lors de l'exécution du programme, tous ces morceaux sont réunis pour former le monde complet nécessaire à l'exécution du programme. La liaison de ces morceaux de monde s'effectue en regroupant des fichiers dans des répertoires. Lors de l'exécution d'un programme, celui-ci peut demander à utiliser une classe. La machine virtuelle de Java recherche dans les répertoires la présence de la classe. Il alimente alors sa mémoire, en ayant pris soin d'initialiser quelques variables. L'édition de liens se fait à l'exécution.

Cette approche permet de modifier facilement une classe pour en faire bénéficier tous les logiciels. En effet, il suffit de remplacer un fichier dans un répertoire pour que tous les programmes soient mis à jour. Ce procédé ressemble à l'approche DLL (*Librairie dynamique*) mais possède une granularité plus fine, limitée à une classe.

L'inconvénient de cette démarche est qu'il est difficile de garantir la bonne livraison d'un programme. Tant que celui-ci n'a pas demandé toutes les classes qui lui sont nécessaires, il est impossible de vérifier l'installation. Le programme peut en effet utiliser une partie seulement des classes et, à un moment particulier, demander l'utilisation d'une nouvelle classe qui sera recherchée dans le répertoire. Une exception sera alors générée si elle n'est pas présente. Les tests unitaires doivent être rigoureux sur ce point.

Java est compilé dans un byte-code qui est interprété par une machine virtuelle. Cet interpréteur a été porté sur de nombreux environnements. Les paquetages de base ont également été portés. Cela garantit une portabilité des applications écrites avec ce langage.

Java étant typé, il n'y a pas de recherche à effectuer pour retrouver une méthode. L'appel d'une méthode utilise une indirection via une table de saut. Il existe une table de saut par classe. Les choix possibles sont limités. Pour augmenter la vitesse du code généré, il est possible d'ajouter des adjectifs aux méthodes pour indiquer qu'elles ne peuvent plus être surchargées, ce qui permet au compilateur d'utiliser des techniques agressives de compilation en générant par exemple le corps de la méthode pour chaque appel.

Le byte-code généré est très efficace. Les traitements les plus lents sont :

- la création d'une instance,
- l'appel d'une méthode synchronisée pour le multitâches.

Les rapports de vitesse sont d'environ 20 fois plus lents que le C++. Pour les améliorer, des compilateurs « Just-In-Time » ont été développés, qui permettent de compiler en code machine natif le byte-code Java. Cette compilation est effectuée à la volée. Lors de la lecture d'une classe, une version compilée remplacera le byte-code des méthodes. Avec cette technologie, il

est possible d'améliorer la vitesse d'exécution d'un facteur proche de deux par rapport au même programme interprété. Il existe des compilateurs natifs qui convertissent (avec quelques limitations) le byte-code Java en assembleur pour une machine cible particulière, ce qui permet d'atteindre un rapport 1,5/2 par rapport au C++. Mais dans ce cas, l'avantage de la portabilité est perdu. Le programme ainsi généré est spécifique à une architecture. Cette technique est généralement utilisée sur les serveurs dont on maîtrise la configuration logicielle et matérielle.

Interprété	Just-in-Time	Compilation native
100	66	11

Plus le chiffre est petit, plus le programme est rapide.

Certaines versions de l'interpréteur Java améliorent fortement les performances, au point d'égaler ou de dépasser certains compilateurs à la volée. Elles détectent lors de l'exécution les lignes de code les plus souvent utilisées et les optimisent spécifiquement. Les programmes passent généralement 90% du temps dans 10% du code. L'identification de ces 10% ne peut être connue qu'à l'exécution. Cette information ne peut pas être déduite à la lecture du source. Les compilateurs à la volée sont moins efficaces car ils ne connaissent pas les 10% devant demander plus d'attention. Ils recherchent une compilation rapide au détriment d'une exécution rapide, le temps de compilation pouvant faire perdre les bénéfices de la traduction en langage machine.

De plus, l'industrie va fabriquer des microprocesseurs capables d'interpréter directement le code Java. Cela permet d'avoir une vitesse sans égale, mais oblige à utiliser une unité centrale spécialisée, les Networks Computers, des terminaux spécialisés pour exécuter du code Java.

Java permet généralement de recompiler une classe de base sans devoir tout recompiler. Cela permet des évolutions des classes de base en garantissant une compatibilité ascendante, sans recompilation. Il n'existe pas de « programme » en Java comme on l'entend habituellement. Les classes sont transformées en fichiers distincts possédant le byte-code correspondant. Tous ces fichiers sont placés dans un répertoire. Pour lancer un programme Java, il faut sélectionner la classe qui doit démarrer. Au fur et à mesure des développements, le répertoire possède de plus en plus de classes. Différents programmes peuvent se partager certaines classes. Si une modification intervient sur l'une d'entre elles, le fichier de byte-code sera modifié, et toutes les applications en bénéficieront. Ce mécanisme de chargement et d'édition de liens dynamiques permet de modifier des classes sans recompiler toutes les classes les utilisant.

C++

Le C++ ne permet pas un dialogue à l'exécution avec le compilateur. Les classes sont compilées dans des mondes partiels (les fichiers objet) qui indiquent les classes et les services nécessaires à la constitution d'un monde entier (le programme). Contrairement à Java, les mondes partiels sont très dépendants des autres parties qu'ils utilisent. Si une classe est modifiée, il faut pratiquement régénérer tous les mondes partiels utilisant cette classe.

L'éditeur de liens est l'unificateur des mondes partiels. C'est par l'utilisation de cet utilitaire que le monde dans son entier peut être généré. Il est possible de constituer des ensembles de mondes partiels qui seront introduits dans le programme, si et seulement si, ils sont nécessaires (les librairies).

Le C++ est le langage objet le plus rapide du marché. Le code est compilé pour la machine-cible. Tout est pensé dans ce langage en termes d'optimisation possible. Il existe toujours un moyen de rédiger le programme afin d'augmenter sa vitesse. La résolution des appels de méthodes est extrêmement rapide car il utilise une simple indirection (ou une double indirection en cas d'héritage virtuel).

De plus, il est possible de déclarer des méthodes ou des fonctions pour qu'elles soient dupliquées à chaque appel (`inline`). Le compilateur optimise ainsi fortement le code.

Contrairement à Smalltalk ou Java, lors de la modification d'une classe, il faut généralement recompiler pratiquement toutes les classes l'utilisant. Seule la modification du corps des méthodes non `inline` peut être rectifiée.

Préprocesseur

Certains langages permettent d'utiliser une sur couche macro grâce à laquelle peut être généré le code suivant différents contextes. On peut alors utiliser la compilation conditionnelle ou le paramétrage de la génération du programme.

Par exemple, le programme peut effectuer plus de vérifications au moment de la phase de développement. Lors de la génération de l'exécutable pour livraison, les routines de tests seront supprimées. Le programme ayant passé avec succès tous les tests unitaires, ces derniers ne sont plus utiles. Si une méthode attend un paramètre entier dont la valeur doit être comprise entre zéro et 100, la valeur peut être testée en phase de déverminage. En revanche, en exploitation, cette situation ne doit jamais arriver. Cela correspondrait à une mauvaise utilisation de la méthode. Pour ne pas pénaliser les performances du programme, le test sera supprimé.

Le préprocesseur va permettre d'encadrer le code afin de moduler la génération du programme suivant différents contextes. Les spécificités d'une machine cible seront encadrées à l'aide de cet outil. Lors de la compilation, le développeur indiquera la machine cible désirée, et le source compilé sera adapté en conséquence.

Smalltalk

Smalltalk ne permet pas d'utiliser un préprocesseur. Il faut utiliser des variables globales pour paramétrer l'exécution du programme ; les méthodes modifieront leur comportement en fonction de la valeur de ces variables.

Java

Java ne possède pas de préprocesseur. Il est ainsi difficile de générer une version du programme pour le mode déverminage. L'extraction des autotests supplémentaires de la

version bêta est difficile. En revanche, Java étant directement portable, les cas d'utilisation du préprocesseur sont moins nécessaires.

C++

Le C++ utilise abondamment un préprocesseur qui lui permet de gérer les imports des objets et qui rend possible la compilation conditionnelle. Les préconditions, les postconditions et les invariants peuvent être facilement supprimés du programme lors de la génération de la version de livraison. Ce préprocesseur est bien intégré au langage, de telle sorte que les débogueurs ne sont pas perturbés. En revanche, il ne permet pas de véritable macro car il est dépourvu de structure de boucles.

Interface avec les autres langages

Un langage de programmation doit pouvoir s'interfacer avec le système d'exploitation et avec d'autres programmes, éventuellement rédigés dans un langage différent. Il doit pouvoir s'intégrer correctement avec son environnement. Le langage C est devenu une des références incontournables en matière de librairie. La plupart des librairies techniques sont rédigées avec ce langage. Il faut donc pouvoir s'interfacer avec une routine rédigée en C. Le langage C définit un protocole d'appel de fonction. Il organise l'ordre et la taille des paramètres dans la pile du microprocesseur. Tous les nouveaux langages doivent être capables de générer un appel à une routine C en respectant ce protocole.

Smalltalk

Smalltalk peut appeler des routines rédigées à l'aide du langage C si celles-ci sont présentes dans des bibliothèques dynamiques. Certaines méthodes sont ainsi exécutées à l'aide du microprocesseur natif de l'environnement d'exécution.

Des API permettent également d'appeler des méthodes Smalltalk à partir d'une routine C, mais le démarrage du programme doit s'effectuer sous Smalltalk. Des recherches sont en cours pour lever cette limitation.

Il est possible de rédiger un serveur Smalltalk et d'utiliser les dialogues DDE (*Dynamic Data Exchange*) ou tout autre moyen de communication (Socket, NetBios...) pour dialoguer avec lui. Dans ce cas, un programme externe peut appeler une méthode Smalltalk en traduisant son appel par un ordre DDE, qui est capturé par l'application Smalltalk (approche client/serveur). La méthode correspondante est alors exécutée. Il va de soi que ce mécanisme ne doit pas être utilisé trop souvent. Le temps CPU nécessaire à la résolution d'un appel de ce type est souvent supérieur au temps d'exécution de la méthode appelée.

Java

Il est possible d'appeler du code C à partir d'un programme Java. C'est ainsi que les librairies livrées avec le langage sont adaptées aux environnements d'exécution.

Il est également possible d'appeler du code Java à partir d'un programme C. La machine virtuelle de Java étant dans une librairie partagée, il est possible de démarrer le programme en C ou en C++. Le programme lancera la machine virtuelle en appelant la librairie dynamique.

Microsoft permet d'écrire un programme Java comme un composant ActiveX™, qui peut alors être appelé de tout type d'applications (Visual Basic, C++, Excel...). L'appelant n'a pas connaissance de l'existence de Java. La machine virtuelle de Java est démarrée avec le composant. La technologie J/Direct permet d'appeler un DLL Windows directement à partir de Java. Il faut alors utiliser la machine virtuelle de Microsoft.

C++

Le C++ étant une sur-couche du C, il est possible de faire en C++ tout ce que l'on peut faire à l'aide du C. Smalltalk ou Java peuvent appeler des routines C++. Pour faciliter le mélange entre le C et le C++, la syntaxe permet de signaler dans la signature d'une fonction si celle-ci a été rédigée en C ou en C++. Cette distinction est nécessaire car le C++ offre la surcharge (plusieurs fonctions de même nom mais ayant des paramètres différents), alors que le C ne l'offre pas. Certains compilateurs permettent également de s'interfacer avec d'autres langages (Ada, Cobol, Fortran...). Les procédures d'appel de ces langages sont générées par le compilateur.

Les programmes C peuvent aussi appeler les programmes C++. Initialement, la première version du C++ était un préprocesseur. Il générait un programme C avant de le compiler. Les compilateurs actuels ont gardé cette compatibilité.

Portabilité

Les environnements informatiques sont hétérogènes. Les développements doivent pouvoir s'adapter à différentes architectures sans devoir modifier en profondeur les programmes.

Smalltalk

Il existe plusieurs fournisseurs du langage Smalltalk. Chacun propose une bibliothèque propriétaire. Ces environnements ont été portés sur différentes machines. Un programme Smalltalk peut alors s'exécuter sans modifications sous Windows, OS/2, Unix. Il faut pour cela acquérir les interpréteurs de machine virtuelle correspondante. La portabilité dépend des fournisseurs de ces interpréteurs.

Une normalisation des bibliothèques de base existe. Elle permet de rédiger des classes métiers indépendantes de l'interface utilisateur, portables entre les différents Smalltalks.

Java

Java est un langage fortement portable. C'est un élément fondamental ayant guidé sa conception. Un code Java semi-compilé pour la machine virtuelle est directement exécutable dans différents environnements. Ce code peut être téléchargé via Internet. Il sera interprété par le client.

C++

Le C++ est normalisé. Depuis le début de l'année 1998, le document final indique tous les détails d'implantation que doivent suivre les compilateurs pour prétendre être à la norme. Toutes les entreprises développant des compilateurs C++ sont en train d'adapter leurs produits pour offrir rapidement sur le marché un compilateur mis à jour. Lorsque cette étape sera franchie, les sources C++ pourront être recompilés sur différentes plates-formes.

Normalisation

Les langages ont besoin d'une norme internationale pour garantir la conformité de leur environnement. Des démarches dans ce sens sont en cours.

Smalltalk

Il existe très peu de fournisseurs pour le langage Smalltalk (ParcPlace, Digitalk, IBM, Dolphin, Quasar, Smalltalk MT), mais ils ont décidé de se réunir pour converger leurs solutions sur une base normalisée [ST] appelée X3J20. Ce travail est en cours d'achèvement.

Java

Java est un langage jeune. Sun en est l'inventeur. Il ne désirait pas normaliser le langage car il préférait pouvoir apporter des modifications rapidement. Les organismes de normalisation doivent travailler de longs mois avant la sortie d'un document. Sun voulait garder une réactivité face aux évolutions permanentes du monde Internet. Un document (voir [Java]) est publié pour éclaircir tous les détails du langage et permettre à des tiers de rédiger leurs compilateurs. Depuis, Sun a fait acte de candidature pour une normalisation ISO afin de répondre à la demande du marché. À ce jour, cette candidature a été rejetée.

C++

Le [C++] est normalisé ISO et ANSI sous la référence X3J16. La version finale du document devrait sortir courant 1998.

Maîtrise

Il est facile de connaître un langage, mais plus difficile de le maîtriser suffisamment pour être capable de chiffrer correctement les temps de développement. Les délais d'apprentissage ci-dessous ne sont que des indications. Tout dépend de l'expérience de chacun et de sa capacité à s'adapter à de nouveaux environnements.

Il faut un certain temps pour penser objet et se détacher de la programmation procédurale. Une formation initiale de quinze jours sur le paradigme objet est indispensable.

Smalltalk

Smalltalk est un langage simple. La syntaxe se décrit en deux pages. En trois jours, un élève connaît les rudiments de ce langage. Il faut ensuite qu'il maîtrise la bibliothèque. L'environnement l'aide beaucoup. Après neuf mois, il devient correctement productif. En dix-huit mois, il maîtrise parfaitement Smalltalk. C'est un langage très productif. Il permet de développer des applications complexes en peu de temps.

Java

Java est relativement simple. La syntaxe se maîtrise en cinq jours. Au bout de dix mois, le programmeur est productif. Il faut vingt mois pour devenir un expert. La difficulté avec Java n'est pas tellement le langage, mais de suivre en permanence les évolutions des bibliothèques. De nouvelles versions sortent avec une périodicité de trois mois !

C++

Le C++ est très riche, donc difficile à maîtriser. Il faut deux semaines pour maîtriser correctement toute la syntaxe. Au bout de dix-huit mois, le programmeur devient correctement productif. Il faut plus de deux ans pour maîtriser très correctement le langage. Il existe très peu d'experts dans le monde.

Développement en équipe

Il est rare de nos jours de réaliser une application sans l'aide d'une équipe. Les environnements doivent offrir un cadre rigide pour faciliter ce travail. Avec un langage objet, chaque intervenant sera responsable d'une ou plusieurs classes. Lors de l'évolution de l'une d'entre elles, quel est l'impact sur les autres classes l'utilisant ?

Smalltalk

Avec Smalltalk, si une classe est modifiée en respectant l'interface initiale, il n'y aura aucun impact sur les autres parties du programme. La souplesse promise par le modèle objet est parfaitement respectée. Le développement en équipe est ainsi très facile. Des interfaces utilisateur permettent de maintenir un référentiel pour rendre possibles l'importation et l'exportation de classes, la gestion des versions… Les environnements à cet effet sont très riches (`Visual Age`, `Smalltalk Entreprise` ou `Envy/Developer`).

Java

Java permet également de modifier une classe sans perturber le programme. Il y a néanmoins quelques restrictions. Une méthode peut être déclarée comme devant être générée à chaque appel. Elle permet d'optimiser le code. En général, ces méthodes sont très petites et n'évoluent pas beaucoup. Dans ce cas, si elles sont modifiées, il faut recompiler toutes les classes les utilisant. La modification des autres méthodes, l'ajout d'une méthode ou d'un attribut n'ont pas d'impact sur les programmes existants. Le développement en équipe est facile si l'on suit quelques règles de codage.

C++

Si l'on ajoute ou si l'on modifie un attribut, même caché dans le corps de la classe, il faut recompiler tous les programmes l'utilisant. Le C++ est très rigide sur les latitudes de modification d'une classe. Le développement en équipe devient alors un véritable problème.

Si l'on désire modifier une classe utilisée par beaucoup d'autres classes, il faut toutes les recompiler. Il n'est pas rare de devoir recompiler tout un programme suite à l'ajout d'un attribut. Il existe quelques artifices de codage pour éviter cet inconvénient, mais ils compliquent énormément les développements. Le travail en équipe est difficile avec le C++.

Partie 3

Conclusion

Après ce tour d'horizon des différents concepts présents dans ces trois langages orientés objets, il est possible de synthétiser les qualités et les défauts des différentes approches. Il existe d'autres langages objet. On peut citer : Ada, Eiffel, Self ou CLOS, pour les plus connus. Il existe d'autres concepts dans ces langages (pré ou postconditions et invariants ; méthodes avant, après, pendant ; héritage dynamique ; syntaxe déclarative et/ou impérative ; prototypes...). N'étant pas présents dans les trois langages choisis pour ce livre, ils ne sont pas décrits. Suivant la qualité des équipes, les performances attendues, les ressources nécessaires à l'exécution, la qualité des programmes produits et leur évolution, différents langages conviendront.

Attention, l'accent n'est porté que sur les concepts présents dans les langages. Il n'est pas question des concepts et des bibliothèques présents dans les différents environnements. Il ne faut pas juger un langage uniquement sur la richesse de sa syntaxe, mais également sur les outils d'aide aux développements. Les informations ci-dessous ne sont que des éléments de comparaison et en aucun cas des informations sur les qualités globales de chaque *environnement*.

Les bibliothèques sont rédigées à l'aide des langages qui les supportent. L'utilisation des bibliothèques dépend des capacités des langages. Le modèle objet apporte des facilités indéniables quant à leur utilisation et à leur adaptation à un contexte particulier.

Smalltalk

Smalltalk est un langage très **simple** d'emploi. Le développement est réellement **incrémental**. On ajoute au fur et à mesure les attributs et les méthodes nécessaires. L'apprentissage est très rapide. En neuf mois, un développeur devient correctement productif.

Il n'est pas nécessaire de maîtriser parfaitement l'ensemble des concepts décrits ci-dessus car ils sont dilués dans l'approche utilisée. Il faut utiliser des principes de traduction correspondants et documenter correctement les programmes. Smalltalk n'étant pas typé, les erreurs sont souvent masquées. Une modification ultérieure peut alors faire apparaître les erreurs, qui ont peu de chance d'être identifiées sans des **tests unitaires poussés**. Smalltalk n'aide pas à la qualité en amont, car il ne détecte pas les erreurs avant qu'elles n'arrivent. Il n'y a pas d'erreur technique détectée par le compilateur. La sémantique est pauvre par rapport à Java ou au C++. Heureusement, il existe des outils (`RefactoryBrowser`) permettant d'effectuer une soixantaine de tests sémantiques.

Les tests sont faciles à rédiger car il est possible à tout moment de demander l'exécution instantanée d'un traitement. Il n'est pas nécessaire de modifier le programme pour effectuer un test. Il est malgré tout préférable de rédiger des classes de tests pour vérifier la non-régression du logiciel. Les erreurs en Smalltalk sont facilement identifiables. Tant que l'on ne manipule pas les mécanismes de base du langage, il est difficile d'avoir des erreurs aléatoires car il n'y a pas de manipulation de pointeur.

Smalltalk est fondé sur un petit nombre de concepts appliqués uniformément. Il faut utiliser les commentaires pour éclairer les programmes. Il n'y a aucune vérification syntaxique, ni aucun outil syntaxique d'aide à l'implantation des concepts. Sauf pour l'héritage, le développeur n'est pas aidé pour maîtriser correctement son programme.

En revanche, la **cohérence** et la **réflexivité** de Smalltalk permettent de lui offrir de nouveaux paradigmes très facilement. Les concepts fondamentaux sont simples. Il est aisé d'ajouter des extensions aux objets. Smalltalk est un langage extrêmement permissif. Tout est modifiable. Le compilateur de byte-code est une classe de Smalltalk. Il est alors possible d'étendre Smalltalk ou de le modifier. Smalltalk est une machine **puissante**. Il est facile d'en modifier les fonctionnalités pour concevoir un nouveau langage. Le même moteur sera utilisé.

Smalltalk est **le plus lent**. Il excelle dans les interfaces utilisateurs mais doit être proscrit pour des traitements algorithmiques complexes. Le programme minimal exige un **coût mémoire important**. Si le programme est conséquent, ce droit d'entrée sera proportionnellement négligeable. La vitesse des programmes Smalltalk dépend des interpréteurs de byte-code (il existe des compilateurs Just-in-time).

Smalltalk n'est pas mis en valeur par l'optique de cet ouvrage. Ce sont les **environnements** de développement qui font toute la différence. La **productivité** en Smalltalk est grandement améliorée par des outils ergonomiques et d'un plus haut niveau conceptuel. Une batterie de navigateurs permettent de consulter le programme dans tous les sens. Le développement incrémental facilite les tests unitaires. Il est possible de tester une méthode immédiatement après sa rédaction. Ce confort indéniable évite les éternels cycles de rédaction, compilation, test.

- Les développeurs traditionnels (COBOL, Basic, C) n'auront pas trop de mal à s'adapter à ce langage. Ils devront accepter de modifier leurs réflexes de programmation pour utiliser le paradigme objet. Smalltalk contraindra les développeurs à penser objet.
- Un développeur Java ne sera pas surpris par ce langage. Il regrettera peut-être l'absence de types (permettant de de localiser les erreurs). La notion de bloc (objet de traitement) sera, dans un premier temps, difficile à maîtriser correctement.
- Un développeur C++ appréciera la cohésion et la simplicité du langage. Il appréciera la modélisation objet d'un traitement. Il regrettera l'absence de `template`, de l'héritage multiple, du contrôle des destructeurs et du typage.

Java

Java constitue un bon compromis entre le C++ et Smalltalk. Il propose des **limitations** par rapport aux approches plus radicales. Il utilise certaines des bonnes idées sans les exploiter toutes. Il apporte la prise en compte syntaxique du **multitâche** et les **classes contextuelles**. Le mélange des deux sémantiques (par référence et par valeur) ne facilite pas l'encapsulation. De plus, la **rigidité des exceptions** complique les phases de conception.

De nombreux concepts existent dans la syntaxe. Il est ainsi facile de relire un programme. L'identification des concepts est aisée car ils apparaissent dans le code. Les commentaires techniques sont alors souvent superflus. En revanche, Java n'est **pas évolutif**. Il est impossible d'ajouter de nouveaux concepts.

La **qualité** est un des points forts du langage. Sur le plan syntaxique, il existe beaucoup de vérifications, et en cours d'exécution, le programme est sous contrôle. Une erreur technique a très peu de chances de passer inaperçue. L'absence de manipulation de pointeur supprime les erreurs aléatoires dues à leur mauvaise utilisation.

La révolution Java relève essentiellement de son approche **portable** et **sécurisée**. Le chargement des classes en cours d'exécution est également un des points forts. On peut parler de « révolution » pour le langage Java, car son acceptation par le marché a été extrêmement rapide et inattendue.

Afin de permettre l'exécution de programmes en toute sécurité, Java définit plusieurs couches de contrôles :

- Le langage Java lui-même et son compilateur sont spécifiés pour respecter un ensemble de règles de sécurité.
- Une **vérification du byte-code** durant la phase de chargement dynamique permet de vérifier que le compilateur qui l'a généré est digne de confiance.
- Le **chargement de classes** assure qu'une nouvelle classe chargée dans la machine virtuelle ne viendra pas interférer avec l'environnement d'exécution existant.
- Une dernière couche pour les **accès fichier** et les **accès réseau** interdit aux applications téléchargées depuis Internet de modifier ou de consulter le poste utilisateur.

Le **byte-code** Java étant **interprété**, sa rapidité dépend de la qualité de l'interpréteur. Des techniques de compilation à la volée (voir « Compilation », page 109) permettent des vitesses d'exécution se rapprochant d'un programme C++. Le droit d'entrée minimal dépend de la taille de l'interpréteur. La version de base est peu gourmande en mémoire. Java a été initialement écrit pour fonctionner sur des systèmes embarqués.

Il existe maintenant des compilateurs Java, au sens classique du terme. Le programme Java est compilé pour une machine cible particulière. Il n'est plus portable, mais bénéficie de la vitesse du processeur. Il existe des compilateurs natifs Java pour différentes plates-formes. Cela permet de compiler le programme lors de son installation. On bénéficie alors de la portabilité de Java sans en payer le coût en termes de vitesse.

Il ne faut pas utiliser Java pour des programmes techniques nécessitant un contrôle strict des ressources. C'est un langage intéressant pour des applications, mais pas pour des *drivers* ou des moniteurs transactionnels par exemple.

Java utilise les caractères UNICODE qui permettent de rédiger des programmes fonctionnant quel que soit le langage utilisé dans le monde. L'UNICODE est un code sur 16 bits en remplacement du code ASCII 7 bits. Il contient tous les caractères du monde. Une chaîne de caractères en Java est une succession de codes 16 bits.

Il existe de plus en plus d'environnements performants pour ce langage. Java étant proche du C++, les ténors du marché ont rapidement adapté leurs environnements C++ pour Java.

- Les développeurs traditionnels n'auront pas trop de mal à s'adapter à ce langage. Il leur faudra accepter de modifier leurs réflexes de programmation pour utiliser le paradigme objet.
- La population Smalltalk ne devrait avoir aucun mal à s'adapter à Java. La sémantique par relation est similaire dans les deux langages. Les difficultés concerneront la gestion des exceptions, les conversions de références et le multitâche.
- La population C++ ne devrait également pas avoir de difficultés à migrer vers ce langage. La syntaxe est proche. Le passage à une sémantique par références troublera dans un premier temps les développeurs C++, car ils ont l'habitude de tout contrôler. Ils seront certainement frustrés par les manques de Java par rapport au C++, mais ils apprécieront sa robustesse et sa portabilité Internet.

C++

Le C++ est le langage objet le plus **rapide** du marché. C'est l'évolution naturelle du C ANSI. Tout est prévu dans ce langage pour améliorer la performance. De plus, les programmes C++ ne sont **pas gourmands** en mémoire.

Ces qualités ont un coût. Le C++ n'est absolument **pas robuste**. Les pièges sont innombrables. Il est très difficile de rédiger correctement sans être un *gourou* du langage. Il est très facile de rencontrer des erreurs techniques sans savoir d'où elles viennent. Pour pallier ce manque de robustesse, beaucoup d'erreurs sont détectées en phase de compilation. On juge d'ailleurs la qualité d'un compilateur par la liste des erreurs qu'il détecte. Un problème technique apparaîtra à l'exécution par des effets de bords difficiles à interpréter. Si le programme fonctionne lors des tests unitaires poussés, il est rare qu'il en existe encore. Citons Steve Clamage : « C++ vous protège de Murphy, mais pas de Machiavel ».

Le C++ cherche à réduire le travail du développeur en proposant de nombreuses traductions implicites. Des objets temporaires peuvent être créés ou détruits, des appels à des méthodes de conversion peuvent être générés, des initialisations peuvent être appelées, sans que le développeur ait à s'en occuper. Le paradoxe de cette approche est qu'il faut parfaitement maîtriser ces comportements par défaut pour pouvoir les utiliser correctement. En revanche, cela permet de diminuer les risques d'oublis du développeur. Par exemple, si un fichier n'est pas fermé, il le sera automatiquement lorsqu'il ne sera plus nécessaire. Le développeur a le droit d'omettre certaines gestions, les traitements par défaut faisant le travail pour lui.

Le langage C++ offre **le plus de concepts** vérifiables par le compilateur. Il est extrêmement puissant en vitesse. L'ajout de nouveaux concepts est envisageable, mais ils n'auront pas la puissance d'une modification de la syntaxe. Il est possible de l'étendre raisonnablement sans pouvoir le modifier. Presque tous les concepts évoqués ci-dessus possèdent une traduction syntaxique. L'apprentissage initial du langage n'est pas facilité, mais permet par la suite de maîtriser facilement ces concepts. Il y a beaucoup de sens dans les programmes car les concepts ont généralement une traduction syntaxique, mais la syntaxe est difficile.

Un comité travaille depuis plusieurs années à une normalisation du langage. La version officielle est sortie fin 1997 [C++]. Depuis cette date, tous les compilateurs du marché migrent

progressivement vers cette norme. Le portage des développements en sera facilité. Les différences actuelles sont trop nombreuses entre les compilateurs pour utiliser tous les concepts décrits ici sans problème de portabilité.

Les environnements de développement permettent d'améliorer la productivité. La concurrence est rude sur ce terrain. Des bibliothèques C++ spécifiques à un fournisseur sont épaulées par des outils de réalisation d'interfaces utilisateur modifiant directement les sources du logiciel.

- Les développeurs traditionnels peuvent avoir des difficultés à utiliser ce langage complexe. Sa richesse ainsi que ses comportements implicites seront difficiles à maîtriser. Les concepts seront intégrés les uns après les autres. Il faut accepter de modifier les réflexes de programmation pour utiliser correctement le paradigme objet. Il est très facile de ne pas utiliser les avantages de l'objet avec le C++. Un **support** de grande qualité doit accompagner les développeurs.
- Un développeur Smalltalk aura des difficultés à entrer dans la philosophie du C++. Il sera troublé par les gestions mémoire et sera certainement dérouté par des erreurs qu'il ne maîtrisera pas. Il aura besoin d'un support de qualité pour faciliter ce passage.
- Un développeur Java sera également frustré de ne plus avoir de ramasse-miettes. Il appréciera les `templates` et la vitesse d'exécution.

Récapitulatif

Voici un tableau récapitulatif des différents concepts et des solutions proposés par les langages à objets.

	Smalltalk	Java	C++
Encapsulation	Faible	Moyen	Fort
Héritages	Simple	Simple + interface	Multiple
Classe contextuelle	Non	Oui	Non
Polymorphisme	Oui	Oui	Oui
Classe abstraite	Non	Explicite	Effet de bord
Langage typé	Non	Oui	Oui
Sémantique	Référence	Référence pour les objets Valeur pour les types primitifs	Valeur
Référence constante	Non	Non	Oui
Pureté syntaxique	Oui	Non	Non
Structure de choix	Effet de bord	Syntaxique	Syntaxique
Structure de boucle	Effet de bord	Syntaxique, riche	Syntaxique
Fonctions	Non	Non	Oui
Retour	Référence	Référence pour les objets, sinon par valeur	Valeur
Surcharge	Pas nécessaire	Oui	Oui
Redéfinition des opérateurs	Binaire sans priorité	Non	Binaire et unaire avec priorité
Conversion d'objets	Non	Non	Oui
Conversion de références	Pas nécessaire	Oui	Oui
Mutation	Oui	Non	Non
Gestion mémoire	Automatique	Automatique	Manuelle

	Smalltalk	Java	C++
Référence faible	Oui	Oui (Depuis 1.2)	Non
Tableaux	Solides Construction syntaxique limitée aux littéraux	Solide Construction syntaxique sans limites	Fragile Construction syntaxique sans limites
Métamodèle	Lecture et écriture Complet	Lecture Complet	Lecture Pauvre
Constructeur	Non	Oui	Oui
Destructeur	Oui (Retardé)	Oui (Retardé)	Oui
Référence sur une méthode	Oui (symbole)	Oui (Depuis 1.1)	Oui (pointeur)
Traitements objet	Oui	Non	Non
Exception	Partielle	Oui	Oui
Généricité	Non	Non	Oui
Multitraitement	Oui (coopératif)	Oui, syntaxique	Oui, avec librairie
Paquetage	Non	Oui	Oui
Extensibilité	Oui	Non	Non
Persistance	Oui	Oui	Non
Syntaxe	Grammaticale	Fonction	Fonction
Robustesse	Oui (moyen)	Oui	Non
Compilation	Pseudo-code	Pseudo-code, Just-In-Time et sur une puce	Oui
Préprocesseur	Non	Non	Oui
Interface autre langage	Appelant (DLL)	Appelant (DLL) Appelé (DLL)	Appelant Appelé
Portabilité	Natif si même fournisseur.	Natif	Par compilation
Normalisation	ANSI X3J20 (en cours)	Candidature	ANSI X3J16
Maîtrise	Facile (9 mois)	Moyen (10 mois)	Difficile (18 mois)
Développement en équipe	Facile	Facile	Difficile

Le choix d'un langage de développement a un impact important sur la réussite d'un projet. Il faut le sélectionner en respectant plusieurs critères répondant à la spécificité du développement à réaliser :

- la qualité du langage lui-même,
- la richesse des bibliothèques,
- la qualité de l'environnement de développement.

Seul le premier point est abordé dans cet ouvrage. Le langage est l'ossature sans laquelle aucune conception, si brillante soit-elle, ne pourrait s'exprimer. Ce n'est pas parce que l'on possède un langage de qualité qu'il sera utilisé convenablement. On a écrit des merveilles avec le Basic, et des horreurs avec Smalltalk. La technologie ne remplacera jamais le talent. Elle l'aidera et l'inspirera, mais ne le supplantera jamais.

Lexique

Accesseur	Méthode modifiant ou retournant la valeur d'un attribut. Elle permet d'en interdire l'accès direct.
Agrégation	État de possession par un autre objet. Un objet est une agrégation d'attributs.
Attribut	Donnée appartenant à un objet.
Attribut dérivé	Donnée déduite de l'état d'un objet.
Classe	Moule décrivant les attributs et les services d'un objet.
Classe de base	Classe héritée par une autre classe.
Classe dérivée	Classe héritant d'une autre classe.
Classe métier	Classe modélisant une activité d'un métier. Les classes techniques ne sont pas des classes métiers.
Conversion par construction	Action de construire un objet en lui fournissant un seul paramètre. Cela permet de convertir le paramètre à l'aide de la construction.
Encapsulation	Capacité à cacher l'intérieur d'un objet.
Héritage	Capacité à bénéficier de tous les attributs et de tous les services d'une classe. Permet de créer une spécialisation.
Héritage multiple	Capacité à hériter simultanément de plusieurs classes. Permet de modéliser une intersection entre deux ensembles.
Héritage virtuel	Capacité à hériter d'une classe qui sera partagée par les dérivées en cas d'héritage multiple.
Instance	Un exemplaire d'une classe.

Métaclasse	La classe d'une classe.
Méthode	Service disponible pour un objet.
Opérateur de conversion	Opérateur permettant de convertir la valeur d'un objet.
Polymorphisme	Mécanisme permettant d'appeler une méthode et de provoquer un comportement différent suivant l'objet receveur.
Réflexivité	Capacité d'un système à s'autoreprésenter, permettant ainsi une introspection.
Sémaphore	Drapeau permettant de bloquer un accès concurrent lors de traitements multitâches.
Sélecteur	Objet particulier identifiant une méthode.
Signature	Identifiant composé du nom d'une méthode et du type de ses paramètres. Deux méthodes de même nom, ayant des paramètres différents, possèdent des signatures différentes.

Bibliographie

DU MEME AUTEUR

[PP96] PHILIPPE PRADOS *La qualité en C++*, Eyrolles, 1996 (ISBN : 2-212-08917-1)

[PP] http://www.cyber-espace.com/pprados/

AUTRES OUVRAGES

[BR97] XAVIER BRIFFAULT et GERARD SABAH , *Smalltalk, programmation orientée objet et développement d'applications*, Eyrolles, 1997 (ISBN : 2-212-08914-7)

[C++] *Working Paper for Draft Proposed International Standard for Information Systems*, X3J16/96 (http://www.maths.warwick.ac.uk/cpp/pub/wp/html/cd2/index.html)

[CO92] JAMES COPLIEN , *Advanced C++ programming Styles and Idioms*, Addison-Wesley, 1992, (ISBN : 0-201-54855-0)

[DUC97] STEPHANE DUCASSE, *Des techniques de contrôle de l'envoi de messages en Smalltalk*, IAM-97-004

[GO89] ADELE GOLDBERG et DAVE ROBSON, *Smalltalk-80: The Language and its implementation*, Addison-Wesley, 1989 (ISBN : 0-201-13688-0)

[Java] JAMES GOSLING, BILL JOY, GUY STEELE, *The Java Language Specification*, Addison Wesley (ISBN : 0-201-63451-1)

[KE97] KENT BECK, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997 (ISBN : 0-13-476904-X)

[KU70] THOMAS KUHN, *The structure of scientific revolutions*, University of Chicago Press, Chicago, Illinois, 2nd edition, 1970

[LO77] LORIE, « Physical integrity in a large segmented database », *ACM TODS*, Vol. 2 N°1, mars 1977

[MEY96] SCOTT MEYERS, *More Effective C++*, Addison-Wesley, 1996 (ISBN : 0-201-63371-X)

[SKU96] S. SKUBLICS, E. KLIMAS et D. THOMAS, *Smalltalk with Style*, Prentice-Hall , 1996, (ISBN : 0-13-165549-3)

[ST] *ANSI X3J20 Technical Committee Smalltalk standard*

Index

—A—

abstract (*classe)*, **25**
accesseur, **6**, 95, 134
agrégation, **13**, 134
applet (*classe*), **79**
Array (*classe*), **71**
asString (*méthode*), **56**
at: (*méthode*), **71**
at:put: (*méthode*), **71**
attribut, **3**, 6, 134
  dérivé, **37**, 134

—B—

bad_exception (*classe*), **88**
Bag (*classe*), **71**
become: (*methode*), **61**
BlockClosure (*classe*), 46, **83**
break (*mot clé*), **45**

—C—

call-back, **81**
case (*mot clé*), **15**
changeClassToThatOf: (*méthode*), **62**
classe, **3**, 134
  abstraite, **24**
  contextuelle, **27**
  de base, **134**
  dérivée, 8, **11**, 134
  inner, 9, 10, **27**
  métier, **18**, 134
collect: (*méthode*), **45**
compilation, 21, **109**
CompiledMethod (*classe*), **83**
const (*mot clé*), **38**
const_cast<> (*conversion*), **60**
constructeur, 57, **77**, 89
continue (*mot clé*), **45**
conversion, **55**
  de références, **58**
  implicite, **55**
  par construction, **57**, 134

—D—

DDE, **115**
delete (*mot clé*), **80**
destroy (*méthode*), **79**
destructeur, 21, 36, **79**, 88
detect: (*méthode*), **45**
développement en équipe, **122**

Dictionary (*classe*), **71**
do while (*mots clés*), **45**
doesNotUnderstand: (*méthode*), **17**
dynamic_cast<> (*conversion*), **60**

—E—

encapsulation, **3**, 9, 20, 99, 100, 134
exception, 72, **85**, 108
extensibilité, **99**

—F—

false (*mot clé*), **42**
finalize (*méthode*), 66, **79**
fixClassTo: (*méthode*), **62**
fonction, **46**
for (*mot clé*), **45**

—G—

généralisation, **12**
généricité, 21, 73, **90**
gestion mémoire, **63**
goto (*mot clé*), **45**

—H—

héritage, **11**, 134
  multiple, **12**, 18, 19, 20, 134
  private, **21**
  protected, **21**
  public, **21**
  virtuel, **13**, 20, 134
HTML (*syntaxe*), **80**

—I—

if then else (*mots clés)*, **43**
ifFalse: (*méthode*), **42**
ifTrue: (*méthode*), **42**
implements (*mot clé*), **19**
import (*mot clé*), **97**
initialize (*méthode*), **7**
inject:into: (*méthode*), **45**
inline (*mot clé*), **112**
inner class, 9, **27**
instance, **4**, 134
interface (*mot clé*), **19**
interface autres langages, **115**
Interval (*classe*), **45**

—J—

just-in-time (*compilateur*), **109**

—L—

langage
  non typé, **30**
  typé, **30**
LargeInteger (*classe*), **53**

—M—

maîtrise, **120**
Mark and sweep (*algorithme*), **64**
méta
  classe, **5**, 135
  modèle, **74**
Method (*classe*), **82**
méthode, **3**, 135
  de classe, **6**
  d'instance, **6**
  surcharge, **50**
multitraitement
  coopératif, **95**
  non préemptif, **95**
  préemptif, **93**
mutable (*mot clé*), **39**
mutation, **61**

—N—

new: (*méthode*), **71**
nil (*objet*), **33**
normalisation

X3J16, **119**
X3J20, **119**

—O—

Object (*classe*), **17**
objet, **3**
opérateur, **52**
de conversion, **56**, 135
OrderedCollection (*classe*), **71**

—P—

package (*mot clé*), **97**
paramètre, **6**
persistance, **101**
polymorphisme, **14**, 20, 99, 135
portabilité, **117**
préprocesseur, **113**
private, **8**
héritage, **21**
Promise (*classe*), **95**
protected, **8**
héritage, **21**
pseudo variable, **19**
public, **8**
héritage, **21**
pureté syntaxique, **40**

—R—

ramasse miettes, **63**
refactoringBrowser (*utilitaire*), **25**
référence, 31, **33**
agrégation, **35**
compteur de, **35**
constante, 37, **38**
faible, **69**
nil, **77**
sémantique par, **33**
sur un objet, **33**, 48
sur une méthode, **81**
tableau de, **73**
réflexivité, **74**, 135
reinterpret_cast<> (*conversion*), **60**
reject: (*méthode*), **45**
retour, **48**
return (*mot clé*), **48**, 49
robustesse, **107**
RTTI, **76**

—S—

select: (*méthode*), **45**
sélecteur, **105**, 135
self (*pseudo variable)*, **4**, 48
sémantique
par références, **33**, 95
par valeur, **32**
sémaphore, **93**, 135
Set (*classe*), **71**
set_terminate() (*fonction*), **89**
signature, **14**, 135
SmallInteger (*classe*), 53
sous-classe, **6**
spécialisation, **12**
static_cast<> (*conversion)*, **60**
String (*classe*), **97**
structure
de boucle, **44**
de choix, **42**
subclassResponsibility (*méthode*), **25**
super (*pseudo variable)*, **17**, 19, 20
surcharge, **50**
switch (*mot clé*), 15, **43**
syntaxe
fonction, **105**
phrase, 50, **105**

—T—

tableau, **71**
template (*mot clé*), **90**

terminate() (*méthode*), **89**
this (*pseudo variable*), **4**
thisContext (*classe*), **67**
thread, **93**
Throwable (*classe*), **88**
to:do: (*méthode*), **45**
toString (*méthode*), **56**
traitement objet, 46, 48, **83**, 87
transaction, **107**
transient (*mot clé*), **101**
true (*mot clé*), **42**
type symbolique, **30**

—V—

valeur, 7, 9, **32**
value (*méthode*), **83**
valueWith: (*méthode*), **83**
variable
  de classe, **6**
  de pool, **7**
  d'instance, **6**
  globale, **6**
  locale, **6**
  pseudo, **19**
variablebyteSubclass (*classe*), **71**
variableLongSubclass (*classe*), **71**
variableSubclass (*classe*), **71**
variableWordSubclass (*classe*), **71**
Visual Age, **3**
void (*mot clé*), **48**, 49

—W—

WeakReference (*classe*), **70**
while (*mot clé*), **45**
whileTrue: (*méthode*), **44**

—X—

X3J16 (*normalisation*), **119**
X3J20 (*normalisation*), **119**

—Y—

yield (*méthode*), **95**

Philippe PRADOS est ingénieur expert à IBM Global Services West. Il a déjà publié « La qualité en C++ » chez le même éditeur.

Le paradigme objet devient incontournable pour les développeurs. Le choix d'un nouveau langage de développement doit s'effectuer suivant des critères rigoureux afin de répondre aux objectifs de l'entreprise.

Ce livre décrit les éléments les plus importants des trois langages objet les plus utilisés actuellement : C++, Java, Smalltalk. Ces langages sont comparables mais néanmoins différents. Ils offrent nativement plus ou moins de concepts de base, libérant le développeur de leurs traductions.

Les chefs de projet et les développeurs trouveront dans ce livre un éclaircissement aux questions qu'ils pourront se poser devant le choix d'un langage.