

28.04.2001 - version 5.7

- Remise en forme du code html [Armel avec HTML-Kit], balises font, a...

04.08.2000 - version 5.6

- Corrections apportées par Daniel Le Berre

26.06.2000 - version 5.5

- Insertion du journal de log.

18.06.2000 - version 5.4

- Modification des tags {url: http://www.blahblah.com} en www.blahblah.com.
- "langage de scripting" modifié en "langage de script".
- Modification des guillemets " " en « ».

06.06.2000 - version 5.3

- Corrections apportées par Philippe Boîte.

05.06.2000 - version 5.2

- Corrections apportées par Jean-Pierre Vidal.

02.06.2000 - version 5.1

- Première publication sur eGroups.

1: Introduction sur les Objets

La révolution informatique a pris naissance dans une machine. Nos langages de programmation ont donc tendance à ressembler à cette machine.

Mais les ordinateurs ne sont pas tant des machines que des outils au service de l'esprit (« des vélos pour le cerveau », comme aime à le répéter Steve Jobs) et un nouveau moyen d'expression. Ainsi, ces outils commencent à moins ressembler à des machines et plus à des parties de notre cerveau ou d'autres formes d'expressions telles que l'écriture, la peinture, la sculpture ou la réalisation de films. La Programmation Orientée Objet (POO) fait partie de ce mouvement qui utilise l'ordinateur en tant que moyen d'expression.

Ce chapitre présente les concepts de base de la POO, y compris quelques méthodes de développement. Ce chapitre et ce livre présupposent que vous avez déjà expérimenté avec un langage de programmation procédural, bien que celui-ci ne soit pas forcément le C. Si vous pensez que vous avez besoin de plus de pratique dans la programmation et / ou la syntaxe du C avant de commencer ce livre, vous devriez explorer le CD ROM fourni avec le livre, *Thinking in C: Foundations for C++ and Java*, aussi disponible sur www.BruceEckel.com.

Ce chapitre tient plus de la culture générale. Beaucoup de personnes ne veulent pas se lancer dans la programmation orientée objet sans en comprendre d'abord les tenants et les aboutissants. C'est pourquoi nous allons introduire ici de nombreux concepts afin de vous donner un solide aperçu de la POO. Au contraire, certaines personnes ne saisissent les concepts généraux qu'après en avoir vu quelques mécanismes mis en oeuvre ; ces gens-là se sentent perdus s'ils n'ont pas un bout de code à se mettre sous la dent. Si vous faites

partie de cette catégorie de personnes et êtes impatients d'attaquer les spécificités du langage, vous pouvez sauter ce chapitre - cela ne vous gênera pas pour l'écriture de programme ou l'apprentissage du langage. Mais vous voudrez peut-être y revenir plus tard pour approfondir vos connaissances sur les objets, les comprendre et assimiler la conception objet.

Les bienfaits de l'abstraction

Tous les langages de programmation fournissent des abstractions. On peut dire que la complexité des problèmes qu'on est capable de résoudre est directement proportionnelle au type et à la qualité de nos capacités d'abstraction. Par « type », il faut comprendre « Qu'est-ce qu'on tente d'abstraire ? ». Le langage assembleur est une petite abstraction de la machine sous-jacente. Beaucoup de langages « impératifs » (tels que Fortran, BASIC, et C) sont des abstractions du langage assembleur. Ces langages sont de nettes améliorations par rapport à l'assembleur, mais leur abstraction première requiert une réflexion en termes de structure ordinateur plutôt qu'à la structure du problème qu'on essaye de résoudre. Le programmeur doit établir l'association entre le modèle de la machine (dans « l'espace solution », qui est l'endroit où le problème est modélisé, tel que l'ordinateur) et le modèle du problème à résoudre (dans « l'espace problème », qui est l'endroit où se trouve le problème). Les efforts requis pour réaliser cette association, et le fait qu'elle est étrangère au langage de programmation, produit des programmes difficiles à écrire et à maintenir, et comme effet de bord a mené à la création de l'industrie du « Génie Logiciel ».

L'autre alternative à la modélisation de la machine est de modéliser le problème qu'on tente de résoudre. Les premiers langages tels que LISP ou APL choisirent une vue particulière du monde (« Tous les problèmes se ramènent à des listes » ou « Tous les problèmes sont algorithmiques », respectivement). PROLOG convertit tous les problèmes en chaînes de décisions. Des langages ont été créés en vue de programmer par contrainte, ou pour programmer en ne manipulant que des symboles graphiques (ces derniers se sont révélés être trop restrictifs). Chacune de ces approches est une bonne solution pour la classe particulière de problèmes pour laquelle ils ont été conçus, mais devient une horreur dès lors que vous les sortez de leur domaine d'application.

L'approche orientée objet va un cran plus loin en fournissant des outils au programmeur pour représenter des éléments dans l'espace problème. Cette représentation se veut assez générale pour ne pas restreindre le programmeur à un type particulier de problèmes. Nous nous référons aux éléments dans l'espace problème et leur représentation dans l'espace solution en tant qu'« objets ». (Bien sûr, on aura aussi besoin d'autres objets qui n'ont pas leur analogue dans l'espace problème). L'idée est que le programme est autorisé à s'adapter à l'esprit du problème en ajoutant de nouveaux types d'objet, de façon à ce que, quand on lit le code décrivant la solution, on lit aussi quelque chose qui décrit le problème. C'est une abstraction plus flexible et puissante que tout ce qu'on a pu voir jusqu'à présent. Ainsi, la POO permet de décrire le problème avec les termes mêmes du problème plutôt qu'avec les termes de la machine où la solution sera mise en oeuvre. Il y a tout de même une connexion avec l'ordinateur, bien entendu. Chaque objet ressemble à un mini-ordinateur ; il a un état, et il a à sa disposition des opérations qu'on peut lui demander d'exécuter. Cependant, là encore on retrouve une analogie avec les objets du monde réel - ils ont tous des caractéristiques et des comportements.

Des concepteurs de langage ont décrété que la programmation orientée objet en elle-même n'était pas adéquate pour résoudre facilement tous les problèmes de programmation, et recommandent la combinaison d'approches variées dans des langages de programmation *multiparadigmes*.[\[2\]](#)

Alan Kay résume les cinq caractéristiques principales de Smalltalk, le premier véritable langage de programmation orienté objet et l'un des langages sur lequel est basé Java. Ces caractéristiques représentent une approche purement orientée objet :

1. **Toute chose est un objet.** Il faut penser à un objet comme à une variable améliorée : il stocke des données, mais on peut « effectuer des requêtes » sur cet objet, lui demander de faire des opérations sur

lui-même. En théorie, on peut prendre n'importe quel composant conceptuel du problème qu'on essaye de résoudre (un chien, un immeuble, un service administratif, etc...) et le représenter en tant qu'objet dans le programme.

2. **Un programme est un ensemble d'objets se disant les uns aux autres quoi faire en s'envoyant des messages.** Pour qu'un objet effectue une requête, on « envoie un message » à cet objet. Plus concrètement, on peut penser à un message comme à un appel de fonction appartenant à un objet particulier.
3. **Chaque objet a son propre espace de mémoire composé d'autres objets.** Dit d'une autre manière, on crée un nouveau type d'objet en créant un paquetage contenant des objets déjà existants. Ainsi, la complexité d'un programme est cachée par la simplicité des objets mis en oeuvre.
4. **Chaque objet est d'un type précis.** Dans le jargon de la POO, chaque objet est une *instance* d'une *classe*, où « classe » est synonyme de « type ». La plus importante caractéristique distinctive d'une classe est : « Quels messages peut-on lui envoyer ? ».
5. **Tous les objets d'un type particulier peuvent recevoir le même message.** C'est une caractéristique lourde de signification, comme vous le verrez plus tard. Parce qu'un objet de type « cercle » est aussi un objet de type « forme géométrique », un cercle se doit d'accepter les messages destinés aux formes géométriques. Cela veut dire qu'on peut écrire du code parlant aux formes géométriques qui sera accepté par tout ce qui correspond à la description d'une forme géométrique. Cette *substituabilité* est l'un des concepts les plus puissants de la programmation orientée objet.

Un objet dispose d'une interface

Aristote fut probablement le premier à commencer une étude approfondie du concept de *type* ; il parle de « la classe des poissons et la classe des oiseaux ». L'idée que tous les objets, tout en étant uniques, appartiennent à une classe d'objets qui ont des caractéristiques et des comportements en commun fut utilisée directement dans le premier langage orienté objet, Simula-67, avec son mot clef fondamental **class** qui introduit un nouveau type dans un programme. Simula, comme son nom l'indique, a été conçu pour développer des simulations telles qu'un guichet de banque. Dans celle-ci, vous avez un ensemble de guichetiers, de clients, de comptes, de transactions et de devises - un tas « d'objets ». Des objets semblables, leur état durant l'exécution du programme mis à part, sont groupés ensemble en tant que « classes d'objets » et c'est de là que vient le mot clef **class**. Créer des types de données abstraits (des classes) est un concept fondamental dans la programmation orientée objet. On utilise les types de données abstraits exactement de la même manière que les types de données prédéfinis. On peut créer des variables d'un type particulier (appelés *objets* ou *instances* dans le jargon OO) et manipuler ces variables (ce qu'on appelle *envoyer des messages* ou des *requêtes* ; on envoie un message et l'objet se débrouille pour le traiter). Les membres (éléments) d'une même classe partagent des caractéristiques communes : chaque compte dispose d'un solde, chaque guichetier peut accepter un dépôt, etc... Cependant, chaque élément a son propre état : chaque compte a un solde différent, chaque guichetier a un nom. Ainsi, les guichetiers, clients, comptes, transactions, etc... peuvent tous être représentés par la même entité au sein du programme. Cette entité est l'objet, et chaque objet appartient à une classe particulière qui définit ses caractéristiques et ses comportements.

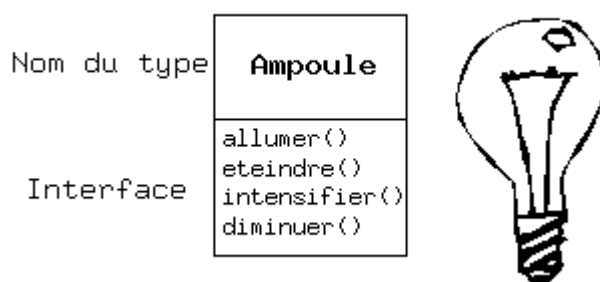
Donc, comme la programmation orientée objet consiste en la création de nouveaux types de données, quasiment tous les langages orientés objet utilisent le mot clef « class ». Quand vous voyez le mot « type » pensez « classe » et inversement [\[3\]](#).

Comme une classe décrit un ensemble d'objets partageant des caractéristiques communes (données) et des comportements (fonctionnalités), une classe est réellement un type de données. En effet, un nombre en virgule flottante par exemple, dispose d'un ensemble de caractéristiques et de comportements. La différence est qu'un programmeur définit une classe pour représenter un problème au lieu d'être forcé d'utiliser un type de données conçu pour représenter une unité de stockage de l'ordinateur. Le langage de programmation est étendu en ajoutant de nouveaux types de données spécifiques à nos besoins. Le système de programmation accepte la nouvelle classe et lui donne toute l'attention et le contrôle de type qu'il fournit aux types prédéfinis.

L'approche orientée objet n'est pas limitée aux simulations. Que vous pensiez ou non que tout programme n'est qu'une simulation du système qu'on représente, l'utilisation des techniques de la POO peut facilement réduire un ensemble de problèmes à une solution simple.

Une fois qu'une classe est créée, on peut créer autant d'objets de cette classe qu'on veut et les manipuler comme s'ils étaient les éléments du problème qu'on tente de résoudre. En fait, l'une des difficultés de la programmation orientée objet est de créer une association un-à-un entre les éléments de l'espace problème et les éléments de l'espace solution.

Mais comment utiliser un objet ? Il faut pouvoir lui demander d'exécuter une requête, telle que terminer une transaction, dessiner quelque chose à l'écran, ou allumer un interrupteur. Et chaque objet ne peut traiter que certaines requêtes. Les requêtes qu'un objet est capable de traiter sont définies par son *interface*, et son type est ce qui détermine son interface. Prenons l'exemple d'une ampoule électrique :



```
Ampoule amp = new Ampoule();  
amp.allumer();
```

L'interface précise *quelles* opérations on peut effectuer sur un objet particulier. Cependant, il doit exister du code quelque part pour satisfaire cette requête. Ceci, avec les données cachées, constitue l'*implémentation*. Du point de vue de la programmation procédurale, ce n'est pas si compliqué. Un type dispose d'une fonction associée à chaque requête possible, et quand on effectue une requête particulière sur un objet, cette fonction est appelée. Ce mécanisme est souvent résumé en disant qu'on « envoie un message » (fait une requête) à un objet, et l'objet se débrouille pour l'interpréter (il exécute le code associé).

Ici, le nom du type / de la classe est **Ampoule**, le nom de l'objet **Ampoule** créé est **amp**, et on peut demander à un objet **Ampoule** de s'allumer, de s'éteindre, d'intensifier ou de diminuer sa luminosité. Un objet **Ampoule** est créé en définissant une « référence » (**amp**) pour cet objet et en appelant **new** pour créer un nouvel objet de ce type. Pour envoyer un message à cet objet, il suffit de spécifier le nom de l'objet suivi de la requête avec un point entre les deux. Du point de vue de l'utilisateur d'une classe prédéfinie, c'est tout ce qu'il est besoin de savoir pour programmer avec des objets.

L'illustration ci-dessus reprend le formalisme UML (*Unified Modeling Language*). Chaque classe est représentée par une boîte, avec le nom du type dans la partie supérieure, les *données membres* qu'on décide de décrire dans la partie du milieu et les *fonctions membres* (les fonctions appartenant à cet objet qui reçoivent les messages envoyées à cet objet) dans la partie du bas de la boîte. Souvent on ne montre dans les diagrammes UML que le nom de la classe et les fonctions publiques, et la partie du milieu n'existe donc pas. Si seul le nom de la classe nous intéresse, alors la portion du bas n'a pas besoin d'être montrée non plus.

L'implémentation cachée

Il est plus facile de diviser les personnes en *créateurs de classe* (ceux qui créent les nouveaux types de données) et *programmeurs clients* [4] (ceux qui utilisent ces types de données dans leurs applications). Le but des programmeurs clients est de se monter une boîte à outils pleine de classes réutilisables pour le développement rapide d'applications (RAD, Rapid Application Development en anglais). Les créateurs de classes, eux, se focalisent sur la construction d'une classe qui n'expose que le nécessaire aux programmeurs clients et cache tout le reste. Pourquoi cela ? Parce que si c'est caché, le programmeur client ne peut l'utiliser, et le créateur de la classe peut changer la portion cachée comme il l'entend sans se préoccuper de l'impact que cela pourrait avoir chez les utilisateurs de sa classe. La portion cachée correspond en général aux données de l'objet qui pourraient facilement être corrompues par un programmeur client négligent ou mal informé. Ainsi, cacher l'implémentation réduit considérablement les bugs.

Le concept d'implémentation cachée ne saurait être trop loué : dans chaque relation il est important de fixer des frontières respectées par toutes les parties concernées. Quand on crée une bibliothèque, on établit une relation avec un programmeur client, programmeur qui crée une application (ou une bibliothèque plus conséquente) en utilisant notre bibliothèque.

Si tous les membres d'une classe sont accessibles pour tout le monde, alors le programmeur client peut faire ce qu'il veut avec cette classe et il n'y a aucun moyen de faire respecter certaines règles. Même s'il est vraiment préférable que l'utilisateur de la classe ne manipule pas directement certains membres de la classe, sans contrôle d'accès il n'y a aucun moyen de l'empêcher : tout est exposé à tout le monde.

La raison première du contrôle d'accès est donc d'empêcher les programmeurs clients de toucher à certaines portions auxquelles ils ne devraient pas avoir accès - les parties qui sont nécessaires pour les manipulations internes du type de données mais n'appartiennent pas à l'interface dont les utilisateurs ont besoin pour résoudre leur problème. C'est en réalité un service rendu aux utilisateurs car ils peuvent voir facilement ce qui est important pour leurs besoins et ce qu'ils peuvent ignorer.

La deuxième raison d'être du contrôle d'accès est de permettre au concepteur de la bibliothèque de changer le fonctionnement interne de la classe sans se soucier des effets que cela peut avoir sur les programmeurs clients. Par exemple, on peut implémenter une classe particulière d'une manière simpliste afin d'accélérer le développement, et se rendre compte plus tard qu'on a besoin de la réécrire afin de gagner en performances. Si l'interface et l'implémentation sont clairement séparées et protégées, cela peut être réalisé facilement.

Java utilise trois mots clefs pour fixer des limites au sein d'une classe : **public**, **private** et **protected**. Leur signification et leur utilisation est relativement explicite. Ces *spécificateurs d'accès* déterminent qui peut utiliser les définitions qui suivent. **public** veut dire que les définitions suivantes sont disponibles pour tout le monde. Le mot clef **private**, au contraire, veut dire que personne, le créateur de la classe et les fonctions internes de ce type mis à part, ne peut accéder à ces définitions. **private** est un mur de briques entre le créateur de la classe et le programmeur client. Si quelqu'un tente d'accéder à un membre défini comme **private**, ils récupéreront une erreur lors de la compilation. **protected** se comporte comme **private**, en moins restrictif : une classe dérivée a accès aux membres **protected**, mais pas aux membres **private**. L'héritage sera introduit bientôt.

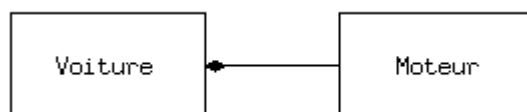
Java dispose enfin d'un accès « par défaut », utilisé si aucun de ces spécificateurs n'est mentionné. Cet accès est souvent appelé accès « amical » car les classes peuvent accéder aux membres amicaux des autres classes du même package, mais en dehors du package ces mêmes membres amicaux se comportent comme des attributs **private**.

Réutilisation de l'implémentation

Une fois qu'une classe a été créée et testée, elle devrait (idéalement) représenter une partie de code utile. Il

s'avère que cette réutilisabilité n'est pas aussi facile à obtenir que cela ; cela demande de l'expérience et de l'anticipation pour produire un bon design. Mais une fois bien conçue, cette classe ne demande qu'à être réutilisée. La réutilisation de code est l'un des plus grands avantages que les langages orientés objets fournissent.

La manière la plus simple de réutiliser une classe est d'utiliser directement un objet de cette classe, mais on peut aussi placer un objet de cette classe à l'intérieur d'une nouvelle classe. On appelle cela « créer un objet membre ». La nouvelle classe peut être constituée de n'importe quel nombre d'objets d'autres types, selon la combinaison nécessaire pour que la nouvelle classe puisse réaliser ce pour quoi elle a été conçue. Parce que la nouvelle classe est composée à partir de classes existantes, ce concept est appelé *composition* (ou, plus généralement, *agrégation*). On se réfère souvent à la composition comme à une relation « possède-un », comme dans « une voiture possède un moteur ».



(Le diagramme UML ci-dessus indique la composition avec le losange rempli, qui indique qu'il y a un moteur dans une voiture. J'utiliserai une forme plus simple : juste une ligne, sans le losange, pour indiquer une association [\[5\]](#).)

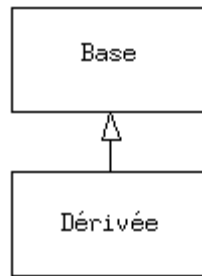
La composition s'accompagne d'une grande flexibilité : les objets membres de la nouvelle classe sont généralement privés, ce qui les rend inaccessibles aux programmeurs clients de la classe. Cela permet de modifier ces membres sans perturber le code des clients existants. On peut aussi changer les objets membres lors la phase d'exécution, pour changer dynamiquement le comportement du programme. L'héritage, décrit juste après, ne dispose pas de cette flexibilité car le compilateur doit placer des restrictions lors de la compilation sur les classes créées avec héritage.

Parce que la notion d'héritage est très importante au sein de la programmation orientée objet, elle est trop souvent martelée, et le nouveau programmeur pourrait croire que l'héritage doit être utilisé partout. Cela mène à des conceptions ultra compliquées et cauchemardesques. La composition est la première approche à examiner lorsqu'on crée une nouvelle classe, car elle est plus simple et plus flexible. Le design de la classe en sera plus propre. Avec de l'expérience, les endroits où utiliser l'héritage deviendront raisonnablement évidents.

Héritage : réutilisation de l'interface

L'idée d'objet en elle-même est un outil efficace. Elle permet de fournir des données et des fonctionnalités liées entre elles par concept, afin de représenter une idée de l'espace problème plutôt que d'être forcé d'utiliser les idiomes internes de la machine. Ces concepts sont exprimés en tant qu'unité fondamentale dans le langage de programmation en utilisant le mot clef **class**.

Il serait toutefois dommage, après s'être donné beaucoup de mal pour créer une classe de devoir en créer une toute nouvelle qui aurait des fonctionnalités similaires. Ce serait mieux si on pouvait prendre la classe existante, la cloner, et faire des ajouts ou des modifications à ce clone. C'est ce que l'*héritage* permet de faire, avec la restriction suivante : si la classe originale (aussi appelée classe de *base*, *superclasse* ou classe *parent*) est changée, le « clone » modifié (appelé classe *dérivée*, *héritée*, *enfant* ou *sousclasse*) répercutera aussi ces changements.

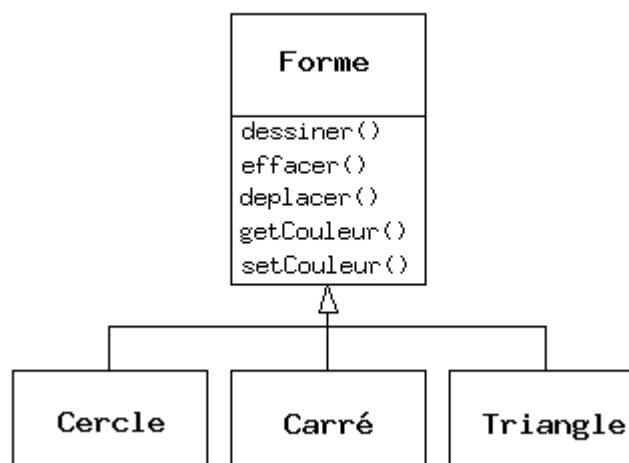


(La flèche dans le diagramme UML ci-dessus pointe de la classe dérivée vers la classe de base. Comme vous le verrez, il peut y avoir plus d'une classe dérivée.)

Un type fait plus que décrire des contraintes sur un ensemble d'objets ; il a aussi des relations avec d'autres types. Deux types peuvent avoir des caractéristiques et des comportements en commun, mais l'un des deux peut avoir plus de caractéristiques que l'autre et peut aussi réagir à plus de messages (ou y réagir de manière différente). L'héritage exprime cette similarité entre les types en introduisant le concept de types de base et de types dérivés. Un type de base contient toutes les caractéristiques et comportements partagés entre les types dérivés. Un type de base est créé pour représenter le coeur de certains objets du système. De ce type de base, on dérive d'autres types pour exprimer les différentes manières existantes pour réaliser ce coeur.

Prenons l'exemple d'une machine de recyclage qui trie les détritux. Le type de base serait « détritux », caractérisé par un poids, une valeur, etc.. et peut être concassé, fondu, ou décomposé. A partir de ce type de base, sont dérivés des types de détritux plus spécifiques qui peuvent avoir des caractéristiques supplémentaires (une bouteille a une couleur) ou des actions additionnelles (une canette peut être découpée, un container d'acier est magnétique). De plus, des comportements peuvent être différents (la valeur du papier dépend de son type et de son état général). En utilisant l'héritage, on peut bâtir une hiérarchie qui exprime le problème avec ses propres termes.

Un autre exemple classique : les « formes géométriques », utilisées entre autres dans les systèmes d'aide à la conception ou dans les jeux vidéos. Le type de base est la « forme géométrique », et chaque forme a une taille, une couleur, une position, etc... Chaque forme peut être dessinée, effacée, déplacée, peinte, etc... A partir de ce type de base, des types spécifiques sont dérivés (hérités) : des cercles, des carrés, des triangles et autres, chacun avec des caractéristiques et des comportements additionnels (certaines figures peuvent être inversées par exemple). Certains comportements peuvent être différents, par exemple quand on veut calculer l'aire de la forme. La hiérarchie des types révèle à la fois les similarités et les différences entre les formes.

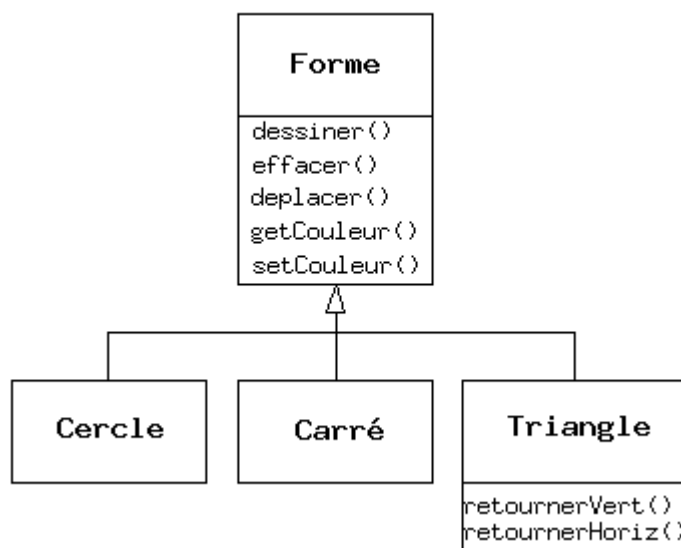


Représenter la solution avec les mêmes termes que ceux du problème est extraordinairement bénéfique car on n'a pas besoin de modèles intermédiaires pour passer de la description du problème à la description de la solution. Avec les objets, la hiérarchie de types est le modèle primaire, on passe donc du système dans le monde réel directement au système du code. En fait, l'une des difficultés à laquelle les gens se trouvent confrontés lors de la conception orientée objet est que c'est trop simple de passer du début à la fin. Les esprits habitués à des solutions compliquées sont toujours stupéfaits par cette simplicité.

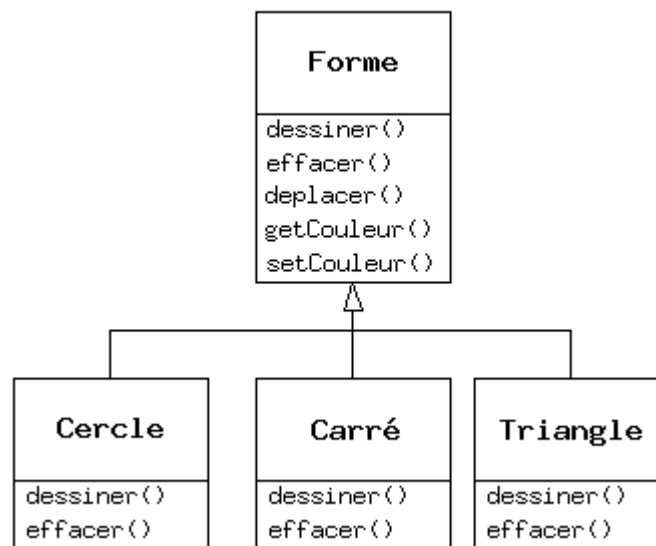
Quand on hérite d'un certain type, on crée un nouveau type. Ce nouveau type non seulement contient tous les membres du type existant (bien que les membres privés soient cachés et inaccessibles), mais plus important, il duplique aussi l'interface de la classe de la base. Autrement dit, tous les messages acceptés par les objets de la classe de base seront acceptés par les objets de la classe dérivée. Comme on connaît le type de la classe par les messages qu'on peut lui envoyer, cela veut dire que la classe dérivée *est du même type que la classe de base*. Dans l'exemple précédent, « un cercle est une forme ». Cette équivalence de type via l'héritage est l'une des notions fondamentales dans la compréhension de la programmation orientée objet.

Comme la classe de base et la classe dérivée ont toutes les deux la même interface, certaines implémentations accompagnent cette interface. C'est à dire qu'il doit y avoir du code à exécuter quand un objet reçoit un message particulier. Si on ne fait qu'hériter une classe sans rien lui rajouter, les méthodes de l'interface de la classe de base sont importées dans la classe dérivée. Cela veut dire que les objets de la classe dérivée n'ont pas seulement le même type, ils ont aussi le même comportement, ce qui n'est pas particulièrement intéressant.

Il y a deux façons de différencier la nouvelle classe dérivée de la classe de base originale. La première est relativement directe : il suffit d'ajouter de nouvelles fonctions à la classe dérivée. Ces nouvelles fonctions ne font pas partie de la classe parent. Cela veut dire que la classe de base n'était pas assez complète pour ce qu'on voulait en faire, on a donc ajouté de nouvelles fonctions. Cet usage simple de l'héritage se révèle souvent être une solution idéale. Cependant, il faut tout de même vérifier s'il ne serait pas souhaitable d'intégrer ces fonctions dans la classe de base qui pourrait aussi en avoir l'usage. Ce processus de découverte et d'itération dans la conception est fréquent dans la programmation orientée objet.



Bien que l'héritage puisse parfois impliquer (spécialement en Java, où le mot clef qui indique l'héritage est **extends**) que de nouvelles fonctions vont être ajoutées à l'interface, ce n'est pas toujours vrai. La seconde et plus importante manière de différencier la nouvelle classe est de *changer* le comportement d'une des fonctions existantes de la superclasse. Cela s'appelle *redéfinir* cette fonction.

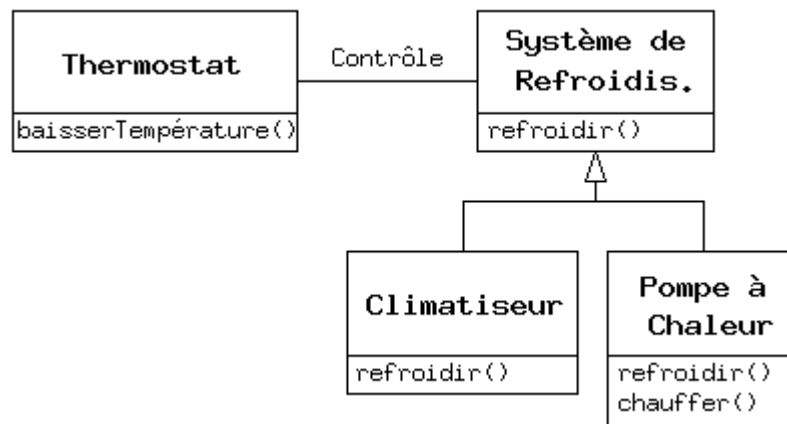


Pour redéfinir une fonction, il suffit de créer une nouvelle définition pour la fonction dans la classe dérivée. C'est comme dire : « j'utilise la même interface ici, mais je la traite d'une manière différente dans ce nouveau type ».

Les relations est-un vs. est-comme-un

Un certain débat est récurrent à propos de l'héritage : l'héritage ne devrait-il pas *seulement* redéfinir les fonctions de la classe de base (et ne pas ajouter de nouvelles fonctions membres qui ne font pas partie de la superclasse) ? Cela voudrait dire que le type dérivé serait *exactement* le même que celui de la classe de base puisqu'il aurait exactement la même interface. Avec comme conséquence logique le fait qu'on puisse exactement substituer un objet de la classe dérivée à un objet de la classe de base. On fait souvent référence à cette *substitution pure* sous le nom de *principe de substitution*. Dans un sens, c'est la manière idéale de traiter l'héritage. La relation entre la classe de base et la classe dérivée dans ce cas est une relation *est-un*, parce qu'on peut dire « un cercle *est une* forme ». Un test pour l'héritage est de déterminer si la relation est-un entre les deux classes considérées a un sens.

Mais parfois il est nécessaire d'ajouter de nouveaux éléments à l'interface d'un type dérivé, et donc étendre l'interface et créer un nouveau type. Le nouveau type peut toujours être substitué au type de base, mais la substitution n'est plus parfaite parce que les nouvelles fonctions ne sont pas accessibles à partir de la classe parent. On appelle cette relation une relation *est-comme-un* [6]; le nouveau type dispose de l'interface de l'ancien type mais il contient aussi d'autres fonctions, on ne peut donc pas réellement dire que ce soient exactement les mêmes. Prenons le cas d'un système de climatisation. Supposons que notre maison dispose des tuyaux et des systèmes de contrôle pour le refroidissement, autrement dit elle dispose d'une interface qui nous permet de contrôler le refroidissement. Imaginons que le système de climatisation tombe en panne et qu'on le remplace par une pompe à chaleur, qui peut à la fois chauffer et refroidir. La pompe à chaleur *est-comme-un* système de climatisation, mais il peut faire plus de choses. Parce que le système de contrôle n'a été conçu que pour contrôler le refroidissement, il en est restreint à ne communiquer qu'avec la partie refroidissement du nouvel objet. L'interface du nouvel objet a été étendue, mais le système existant ne connaît rien qui ne soit dans l'interface originale.



Bien sûr, quand on voit cette modélisation, il est clair que la classe de base « Système de refroidissement » n'est pas assez générale, et devrait être renommée en « Système de contrôle de température » afin de pouvoir inclure le chauffage - auquel cas le principe de substitution marcherait. Cependant, le diagramme ci-dessus est un exemple de ce qui peut arriver dans le monde réel.

Quand on considère le principe de substitution, il est tentant de se dire que cette approche (la substitution pure) est la seule manière correcte de modéliser, et de fait *c'est* appréciable si la conception fonctionne ainsi. Mais dans certains cas il est tout aussi clair qu'il faut ajouter de nouvelles fonctions à l'interface d'une classe dérivée. En examinant le problème, les deux cas deviennent relativement évidents.

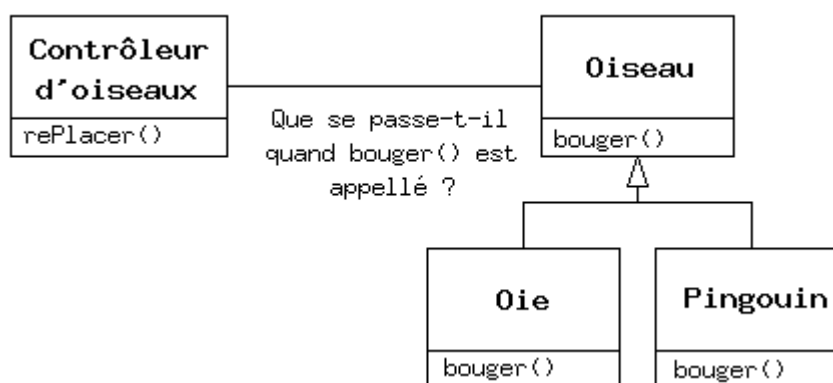
Polymorphisme : des objets interchangeables

Il arrive qu'on veuille traiter un objet non en tant qu'objet du type spécifique qu'il est, mais en tant qu'objet de son type de base. Cela permet d'écrire du code indépendant des types spécifiques. Dans l'exemple de la forme géométrique, les fonctions manipulent des formes génériques sans se soucier de savoir si ce sont des cercles, des carrés, des triangles ou même des formes non encore définies. Toutes les formes peuvent être dessinées, effacées, et déplacées, donc ces fonctions envoient simplement un message à un objet forme, elles ne se soucient pas de la manière dont l'objet traite le message.

Un tel code n'est pas affecté par l'addition de nouveaux types, et ajouter de nouveaux types est la façon la plus commune d'étendre un programme orienté objet pour traiter de nouvelles situations. Par exemple, on peut dériver un nouveau type de forme appelé pentagone sans modifier les fonctions qui traitent des formes génériques. Cette capacité à étendre facilement un programme en dérivant de nouveaux sous-types est important car il améliore considérablement la conception tout en réduisant le coût de maintenance.

Un problème se pose cependant en voulant traiter les types dérivés comme leur type de base générique (les cercles comme des formes géométriques, les vélos comme des véhicules, les cormorans comme des oiseaux, etc...). Si une fonction demande à une forme générique de se dessiner, ou à un véhicule générique de tourner, ou à un oiseau générique de se déplacer, le compilateur ne peut savoir précisément lors de la phase de compilation quelle portion de code sera exécutée. C'est d'ailleurs le point crucial : quand le message est envoyé, le programmeur ne *veut* pas savoir quelle portion de code sera exécutée ; la fonction dessiner peut être appliquée aussi bien à un cercle qu'à un carré ou un triangle, et l'objet va exécuter le bon code suivant son type spécifique. Si on n'a pas besoin de savoir quelle portion de code est exécutée, alors le code exécuté lorsque on ajoute un nouveau sous-type peut être différent sans exiger de modification dans l'appel de la fonction. Le compilateur ne peut donc précisément savoir quelle partie de code sera exécutée, donc que va-t-il faire ? Par exemple, dans le diagramme suivant, l'objet **Contrôleur d'oiseaux** travaille seulement avec des objets **Oiseaux** génériques, et ne sait pas de quel type ils sont. Cela est pratique du point de vue de **Contrôleur d'oiseaux** car il n'a pas besoin d'écrire du code spécifique pour déterminer le type exact d'**Oiseau** avec lequel il travaille, ou le comportement

de cet **Oiseau**. Comment se fait-il donc que, lorsque **bouger()** est appelé tout en ignorant le type spécifique de l'**Oiseau**, on obtienne le bon comportement (une **Oie** court, vole ou nage, et un **Pingouin** court ou nage) ?



La réponse constitue l'astuce fondamentale de la programmation orientée objet : le compilateur ne peut faire un appel de fonction au sens traditionnel du terme. Un appel de fonction généré par un compilateur non orienté objet crée ce qu'on appelle une *association prédéfinie*, un terme que vous n'avez sans doute jamais entendu auparavant car vous ne pensiez pas qu'on puisse faire autrement. En d'autres termes, le compilateur génère un appel à un nom de fonction spécifique, et l'éditeur de liens résout cet appel à l'adresse absolue du code à exécuter. En POO, le programme ne peut déterminer l'adresse du code avant la phase d'exécution, un autre mécanisme est donc nécessaire quand un message est envoyé à un objet générique.

Pour résoudre ce problème, les langages orientés objet utilisent le concept d'*association tardive*. Quand un objet reçoit un message, le code appelé n'est pas déterminé avant l'exécution. Le compilateur s'assure que la fonction existe et vérifie le type des arguments et de la valeur de retour (un langage omettant ces vérifications est dit *faiblement typé*), mais il ne sait pas exactement quel est le code à exécuter.

Pour créer une association tardive, Java utilise une portion spéciale de code en lieu et place de l'appel absolu. Ce code calcule l'adresse du corps de la fonction, en utilisant des informations stockées dans l'objet (ce mécanisme est couvert plus en détails dans le Chapitre 7). Ainsi, chaque objet peut se comporter différemment suivant le contenu de cette portion spéciale de code. Quand un objet reçoit un message, l'objet sait quoi faire de ce message.

Dans certains langages (en particulier le C++), il faut préciser explicitement qu'on souhaite bénéficier de la flexibilité de l'association tardive pour une fonction. Dans ces langages, les fonctions membres ne sont *pas* liées dynamiquement par défaut. Cela pose des problèmes, donc en Java l'association dynamique est le défaut et aucun mot clef supplémentaire n'est requis pour bénéficier du polymorphisme.

Reprenons l'exemple de la forme géométrique. Le diagramme de la hiérarchie des classes (toutes basées sur la même interface) se trouve plus haut dans ce chapitre. Pour illustrer le polymorphisme, écrivons un bout de code qui ignore les détails spécifiques du type et parle uniquement à la classe de base. Ce code est *déconnecté* des informations spécifiques au type, donc plus facile à écrire et à comprendre. Et si un nouveau type - un **Hexagone**, par exemple - est ajouté grâce à l'héritage, le code continuera de fonctionner aussi bien pour ce nouveau type de **Forme** qu'il le faisait avec les types existants. Le programme est donc extensible. Si nous écrivons une méthode en Java (comme vous allez bientôt apprendre à le faire) :

```

void faireQuelqueChose(Forme f) {
    f.effacer();
    // ...
}
  
```

```
f.dessiner();
}
```

Cette fonction s'adresse à n'importe quelle **Forme**, elle est donc indépendante du type spécifique de l'objet qu'elle dessine et efface. Si nous utilisons ailleurs dans le programme cette fonction **faireQuelqueChose()** :

```
Cercle c = new Cercle();
Triangle t = new Triangle();
Ligne l = new Ligne();
faireQuelqueChose(c);
faireQuelqueChose(t);
faireQuelqueChose(l);
```

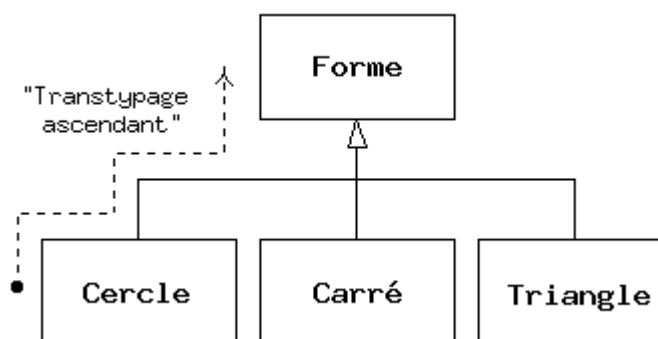
Les appels à **faireQuelqueChose()** fonctionnent correctement, sans se préoccuper du type exact de l'objet.

En fait c'est une manière de faire très élégante. Considérons la ligne :

```
faireQuelqueChose(c);
```

Un **Cercle** est ici passé à une fonction qui attend une **Forme**. Comme un **Cercle** *est-une* **Forme**, il peut être traité comme tel par **faireQuelqueChose()**. C'est à dire qu'un **Cercle** peut accepter tous les messages que **faireQuelqueChose()** pourrait envoyer à une forme. C'est donc une façon parfaitement logique et sûre de faire.

Traiter un type dérivé comme s'il était son type de base est appelé *transtypage ascendant*, *surtypage* ou *généralisation* (upcasting). L'adjectif ascendant vient du fait que dans un diagramme d'héritage typique, le type de base est représenté en haut, les classes dérivées s'y rattachant par le bas. Ainsi, changer un type vers son type de base revient à remonter dans le diagramme d'héritage : transtypage « ascendant ».



Un programme orienté objet contient obligatoirement des transtypes ascendants, car c'est de cette manière que le type spécifique de l'objet peut être délibérément ignoré. Examinons le code de **faireQuelqueChose()** :

```
f.effacer();
// ...
```

```
f.dessiner();
```

Remarquez qu'il ne dit pas « Si tu es un **Cercle**, fais ceci, si tu es un **Carré**, fais cela, etc... ». Ce genre de code qui vérifie tous les types possibles que peut prendre une **Forme** est confus et il faut le changer à chaque extension de la classe **Forme**. Ici, il suffit de dire : « Tu es une forme géométrique, je sais que tu peux te **dessiner()** et t'**effacer()**, alors fais-le et occupe-toi des détails spécifiques ».

Ce qui est impressionnant dans le code de **faireQuelqueChose()**, c'est que tout fonctionne comme on le souhaite. Appeler **dessiner()** pour un **Cercle** exécute une portion de code différente de celle exécutée lorsqu'on appelle **dessiner()** pour un **Carré** ou une **Ligne**, mais lorsque le message **dessiner()** est envoyé à une **Forme** anonyme, on obtient le comportement idoine basé sur le type réel de la **Forme**. Cela est impressionnant dans la mesure où le compilateur Java ne sait pas à quel type d'objet il a affaire lors de la compilation du code de **faireQuelqueChose()**. On serait en droit de s'attendre à un appel aux versions **dessiner()** et **effacer()** de la classe de base **Forme**, et non celles des classes spécifiques **Cercle**, **Carré** et **Ligne**. Mais quand on envoie un message à un objet, il fera ce qu'il a à faire, même quand la généralisation est impliquée. C'est ce qu'implique le polymorphisme. Le compilateur et le système d'exécution s'occupent des détails, et c'est tout ce que vous avez besoin de savoir, en plus de savoir comment modéliser avec.

Classes de base abstraites et interfaces

Dans une modélisation, il est souvent souhaitable qu'une classe de base ne présente qu'une interface pour ses classes dérivées. C'est à dire qu'on ne souhaite pas qu'il soit possible de créer un objet de cette classe de base, mais seulement pouvoir surtyper jusqu'à elle pour pouvoir utiliser son interface. Cela est possible en rendant cette classe *abstraite* en utilisant le mot clef **abstract**. Le compilateur se plaindra si une tentative est faite de créer un objet d'une classe définie comme **abstract**. C'est un outil utilisé pour forcer une certaine conception.

Le mot clef **abstract** est aussi utilisé pour décrire une méthode qui n'a pas encore été implémentée - comme un panneau indiquant « voici une fonction de l'interface dont les types dérivés ont hérité, mais actuellement je n'ai aucune implémentation pour elle ». Une méthode **abstract** peut seulement être créée au sein d'une classe **abstract**. Quand cette classe est dérivée, cette méthode doit être implémentée, ou la classe dérivée devient **abstract** elle aussi. Créer une méthode **abstract** permet de l'inclure dans une interface sans être obligé de fournir une portion de code éventuellement dépourvue de sens pour cette méthode.

Le mot clef **interface** pousse le concept de classe **abstract** un cran plus loin en évitant toute définition de fonction. Une **interface** est un outil très pratique et très largement répandu, car il fournit une séparation parfaite entre l'interface et l'implémentation. De plus, on peut combiner plusieurs interfaces, alors qu'hériter de multiples classes normales ou abstraites est impossible.

Environnement et durée de vie des objets

Techniquement, les spécificités de la programmation orientée objet se résument au typage abstrait des données, à l'héritage et au polymorphisme, mais d'autres particularités peuvent se révéler aussi importantes. Le reste de cette section traite de ces particularités.

L'une des particularités les plus importantes est la façon dont les objets sont créés et détruits. Où se trouvent les données d'un objet et comment sa durée de vie est-elle contrôlée ? Différentes philosophies existent. En C++, qui prône que l'efficacité est le facteur le plus important, le programmeur a le choix. Pour une vitesse optimum à l'exécution, le stockage et la durée de vie peuvent être déterminés quand le programme est écrit, en plaçant les objets sur la pile (ces variables sont parfois appelées *automatiques* ou *de portée*) ou dans l'espace de stockage statique. La vitesse d'allocation et de libération est dans ce cas prioritaire et leur contrôle peut être vraiment appréciable dans certaines situations. Cependant, cela se fait aux dépens de la flexibilité car il faut connaître la

quantité exacte, la durée de vie et le type des objets pendant qu'on écrit le programme. Si le problème à résoudre est plus général, tel que de la modélisation assistée par ordinateur, de la gestion d'entrepôts ou du contrôle de trafic aérien, cela se révèle beaucoup trop restrictif.

La deuxième approche consiste à créer les objets dynamiquement dans un pool de mémoire appelé le segment. Dans cette approche, le nombre d'objets nécessaire n'est pas connu avant l'exécution, de même que leur durée de vie ou leur type exact. Ces paramètres sont déterminés sur le coup, au moment où le programme s'exécute. Si on a besoin d'un nouvel objet, il est simplement créé dans le segment au moment où on en a besoin. Comme le stockage est géré de manière dynamique lors de l'exécution, le temps de traitement requis pour allouer de la place dans le segment est plus important que le temps mis pour stocker sur la pile (stocker sur la pile se résume souvent à une instruction assembleur pour déplacer le pointeur de pile vers le bas, et une autre pour le redéplacer vers le haut). L'approche dynamique fait la supposition généralement justifiée que les objets ont tendance à être compliqués, et que le surcoût de temps dû à la recherche d'une place de stockage et à sa libération n'aura pas d'impact significatif sur la création d'un objet. De plus, la plus grande flexibilité qui en résulte est essentielle pour résoudre le problème modélisé par le programme.

Java utilise exclusivement la deuxième approche [7]. Le mot clef **new** est utilisé pour créer une instance dynamique d'un objet à chaque fois qu'on en a besoin.

Une autre particularité importante est la durée de vie d'un objet. Avec les langages qui autorisent la création d'objets dans la pile, le compilateur détermine combien de temps l'objet est amené à vivre et peut le détruire automatiquement. Mais si l'objet est créé dans le segment, le compilateur n'a aucune idée de sa durée de vie. Dans un langage comme le C++, il faut déterminer dans le programme quand détruire l'objet, ce qui peut mener à des fuites de mémoire si cela n'est pas fait correctement (et c'est un problème courant en C++). Java propose une fonctionnalité appelée ramasse-miettes (garbage collector) qui découvre automatiquement quand un objet n'est plus utilisé et le détruit. Java propose donc un niveau plus élevé d'assurance contre les fuites de mémoire. Disposer d'un ramasse-miettes est pratique car cela réduit le code à écrire et, plus important, le nombre de problèmes liés à la gestion de la mémoire (qui ont mené à l'abandon de plus d'un projet C++).

Le reste de cette section s'attarde sur des facteurs additionnels concernant l'environnement et la durée de vie des objets.

Collections et itérateurs

Si le nombre d'objets nécessaires à la résolution d'un problème est inconnu, ou combien de temps on va en avoir besoin, on ne peut pas non plus savoir comment les stocker. Comment déterminer l'espace nécessaire pour créer ces objets ? C'est impossible car cette information n'est connue que lors de l'exécution.

La solution à la plupart des problèmes en conception orientée objet est simple : il suffit de créer un nouveau type d'objet. Le nouveau type d'objets qui résout ce problème particulier contient des références aux autres objets. Bien sûr, un tableau ferait aussi bien l'affaire. Mais il y a plus. Ce nouvel objet, appelé *conteneur* (ou *collection*, mais la bibliothèque Java utilise ce terme dans un autre sens ; nous utiliserons donc le terme « conteneur » dans la suite de ce livre), grandira automatiquement pour accepter tout ce qu'on place dedans. Connaître le nombre d'objets qu'on désire stocker dans un conteneur n'est donc plus nécessaire. Il suffit de créer un objet conteneur et le laisser s'occuper des détails.

Heureusement, les langages orientés objet décents fournissent ces conteneurs. En C++, ils font partie de la bibliothèque standard (STL, Standard Template Library). Le Pascal Objet dispose des conteneurs dans sa Bibliothèque de Composants Visuels (VCL, Visual Component Library). Smalltalk propose un ensemble vraiment complet de conteneurs. Java aussi propose des conteneurs dans sa bibliothèque standard. Dans certaines bibliothèques, un conteneur générique est jugé suffisant pour tous les besoins, et dans d'autres (Java par exemple), la bibliothèque dispose de différents types de conteneurs suivant les besoins : des vecteurs (appelé

ArrayList en Java) pour un accès pratique à tous les éléments, des listes chaînées pour faciliter l'insertion, par exemple, on peut donc choisir le type particulier qui convient le mieux. Les bibliothèques de conteneurs peuvent aussi inclure les ensembles, les files, les dictionnaires, les arbres, les piles, etc...

Tous les conteneurs disposent de moyens pour y stocker des choses et les récupérer ; ce sont habituellement des fonctions pour ajouter des éléments dans un conteneur et d'autres pour les y retrouver. Mais retrouver des éléments peut être problématique, car une fonction de sélection unique peut se révéler trop restrictive. Comment manipuler ou comparer un ensemble d'éléments dans le conteneur ?

La réponse à cette question prend la forme d'un itérateur, qui est un objet dont le travail est de choisir les éléments d'un conteneur et de les présenter à l'utilisateur de l'itérateur. En tant que classe, il fournit de plus un niveau d'abstraction supplémentaire. Cette abstraction peut être utilisée pour séparer les détails du conteneur du code qui utilise ce conteneur. Le conteneur, via l'itérateur, est perçu comme une séquence. L'itérateur permet de parcourir cette séquence sans se préoccuper de sa structure sous-jacente - qu'il s'agisse d'une **ArrayList** (vecteur), une **LinkedList** (liste chaînée), une **Stack** (pile) ou autre. Cela permet de changer facilement la structure de données sous-jacente sans perturber le code du programme. Java commença (dans les versions 1.0 et 1.1) avec un itérateur standard, appelé **Enumeration**, pour toutes ses classes conteneurs. Java 2 est accompagné d'une bibliothèque de conteneurs beaucoup plus complète qui contient entre autres un itérateur appelé **Iterator** bien plus puissant que l'ancienne **Enumeration**.

Du point de vue du design, tout ce dont on a besoin est une séquence qui peut être manipulée pour résoudre le problème. Si un seul type de séquence satisfaisait tous les besoins, il n'y aurait pas de raison d'en avoir de types différents. Il y a deux raisons qui font qu'on a besoin d'un choix de conteneurs. Tout d'abord, les conteneurs fournissent différents types d'interfaces et de comportements. Une pile a une interface et un comportement différents de ceux d'une file, qui sont différents de ceux fournis par un ensemble ou une liste. L'un de ces conteneurs peut se révéler plus flexible qu'un autre pour la résolution du problème considéré. Deuxièmement, les conteneurs ne sont pas d'une même efficacité pour les mêmes opérations. Prenons le cas d'une **ArrayList** et d'une **LinkedList**. Les deux sont de simples séquences qui peuvent avoir la même interface et comportement. Mais certaines opérations ont des coûts radicalement différents. Accéder à des éléments au hasard dans une **ArrayList** est une opération qui demande toujours le même temps, quel que soit l'élément auquel on souhaite accéder. Mais dans une **LinkedList**, il est coûteux de se déplacer dans la liste pour rechercher un élément, et cela prend plus de temps pour trouver un élément qui se situe plus loin dans la liste. Par contre, si on souhaite insérer un élément au milieu d'une séquence, c'est bien plus efficace dans une **LinkedList** que dans une **ArrayList**. Ces opérations et d'autres ont des efficacités différentes suivant la structure sous-jacente de la séquence. Dans la phase de conception, on peut débiter avec une **LinkedList** et lorsqu'on se penche sur l'optimisation, changer pour une **ArrayList**. Grâce à l'abstraction fournie par les itérateurs, on peut passer de l'une à l'autre avec un impact minime sur le code.

En définitive, un conteneur n'est qu'un espace de stockage où placer des objets. Si ce conteneur couvre tous nos besoins, son implémentation réelle n'a pas grande importance (un concept de base pour la plupart des objets). Mais il arrive que la différence de coûts entre une **ArrayList** et une **LinkedList** ne soit pas à négliger, suivant l'environnement du problème et d'autres facteurs. On peut n'avoir besoin que d'un seul type de séquence. On peut même imaginer le conteneur « parfait », qui changerait automatiquement son implémentation selon la manière dont on l'utilise.

La hiérarchie de classes unique

L'une des controverses en POO devenue prééminente depuis le C++ demande si toutes les classes doivent être finalement dérivées d'une classe de base unique. En Java (et comme dans pratiquement tous les autres langages OO) la réponse est « oui » et le nom de cette classe de base ultime est tout simplement **Object**. Les bénéfices d'une hiérarchie de classes unique sont multiples.

Tous les objets dans une hiérarchie unique ont une interface commune, ils sont donc tous du même type fondamental. L'alternative (proposée par le C++) est qu'on ne sait pas que tout est du même type fondamental. Du point de vue de la compatibilité ascendante, cela épouse plus le modèle du C et peut se révéler moins restrictif, mais lorsqu'on veut programmer en tout objet il faut reconstruire sa propre hiérarchie de classes pour bénéficier des mêmes avantages fournis par défaut par les autres langages OO. Et dans chaque nouvelle bibliothèque de classes qu'on récupère, une interface différente et incompatible sera utilisée. Cela demande des efforts (et éventuellement l'utilisation de l'héritage multiple) pour intégrer la nouvelle interface dans la conception. Est-ce que la « flexibilité » que le C++ fournit en vaut réellement le coup ? Si on en a besoin - par exemple si on dispose d'un gros investissement en C - alors oui. Mais si on démarre de zéro, d'autres alternatives telles que Java se révèlent beaucoup plus productives.

Tous les objets dans une hiérarchie de classes unique (comme celle que propose Java) sont garantis d'avoir certaines fonctionnalités. Un certain nombre d'opérations élémentaires peuvent être effectuées sur tous les objets du système. Une hiérarchie de classes unique, accompagnée de la création des objets dans le segment, simplifie considérablement le passage d'arguments (l'un des sujets les plus complexes en C++).

Une hiérarchie de classes unique facilite aussi l'implémentation d'un ramasse-miettes (qui est fourni en standard en Java). Le support nécessaire est implanté dans la classe de base, et le ramasse-miettes peut donc envoyer le message idoine à tout objet du système. Sans une hiérarchie de classe unique et un système permettant de manipuler un objet via une référence, il est difficile d'implémenter un ramasse-miettes.

Comme tout objet dispose en lui d'informations dynamiques, on ne peut se retrouver avec un objet dont on ne peut déterminer le type. Ceci est particulièrement important avec les opérations du niveau système, telles que le traitement des exceptions, et cela permet une plus grande flexibilité dans la programmation.

Bibliothèques de collections et support pour l'utilisation aisée des collections

Parce qu'un conteneur est un outil qu'on utilise fréquemment, il est logique d'avoir une bibliothèque de conteneurs conçus de manière à être réutilisables, afin de pouvoir en prendre un et l'insérer dans le programme. Java fournit une telle bibliothèque, qui devrait satisfaire tous les besoins.

Transtypes descendants vs. patrons génériques

Pour rendre ces conteneurs réutilisables, ils stockent le type universel en Java précédemment mentionné : **Object**. La hiérarchie de classe unique implique que tout est un **Object**, un conteneur stockant des **Objects** peut donc stocker n'importe quoi. Cela rend les conteneurs aisément réutilisables.

Pour utiliser ces conteneurs, il suffit d'y ajouter des références à des objets, et les redemander plus tard. Mais comme le conteneur ne stocke que des **Objects**, quand une référence à un objet est ajoutée dans le conteneur, il subit un transtypage ascendant en **Object**, perdant alors son identité. Quand il est recherché par la suite, on récupère une référence à un **Object**, et non une référence au type qu'on a inséré. Comment le récupérer et retrouver l'interface de l'objet qu'on a stocké dans le conteneur ?

On assiste ici aussi à un transtypage, mais cette fois-ci il ne remonte pas dans la hiérarchie de classe à un type plus général, mais descend dans la hiérarchie jusqu'à un type plus spécifique, c'est un transtypage descendant ou spécialisation, ou soustypage. Avec la généralisation, on sait par exemple qu'un **Cercle** est un type de **Forme**, et que le transtypage est donc sans danger ; mais on ne sait pas qu'un **Object** est aussi un **Cercle** ou une **Forme**, il est donc rarement sur d'appliquer une spécialisation à moins de savoir exactement à quoi on a affaire.

Ce n'est pas trop dangereux cependant, car si une spécialisation est tentée jusqu'à un type incompatible le système d'exécution générera une erreur appelée *exception*, qui sera décrite plus loin. Quand une référence d'objet est rapatriée d'un conteneur, il faut donc un moyen de se rappeler exactement son type afin de pouvoir le spécialiser correctement.

La spécialisation et les contrôles à l'exécution génèrent un surcoût de temps pour le programme, et des efforts supplémentaires de la part du programmeur. Il semblerait plus logique de créer le conteneur de façon à ce qu'il connaisse le type de l'objet stocké, éliminant du coup la spécialisation et la possibilité d'erreur. La solution est fournie par les types paramétrés, qui sont des classes que le compilateur peut personnaliser pour les faire fonctionner avec des types particuliers. Par exemple, avec un conteneur paramétré, le compilateur peut personnaliser ce conteneur de façon à ce qu'il n'accepte que des **Formes** et ne renvoie que des **Formes**.

Les types paramétrés sont importants en C++, en particulier parce que le C++ ne dispose pas d'une hiérarchie de classe unique. En C++, le mot clef qui implémente les types paramétrés est « template ». Java ne propose pas actuellement de types paramétrés car c'est possible de les simuler - bien que difficilement - via la hiérarchie de classes unique. Une solution de types paramétrés basée sur la syntaxe des templates C++ est actuellement en cours de proposition.

Le dilemme du nettoyage : qui en est responsable ?

Chaque objet requiert des ressources, en particulier de la mémoire. Quand un objet n'est plus utilisé il doit être nettoyé afin de rendre ces ressources pour les réutiliser. Dans la programmation de situations simples, la question de savoir comment un objet est libéré n'est pas trop compliquée : il suffit de créer l'objet, l'utiliser aussi longtemps que désiré, et ensuite le détruire. Il n'est pas rare par contre de se trouver dans des situations beaucoup plus complexes.

Supposons qu'on veuille concevoir un système pour gérer le trafic aérien d'un aéroport (ou pour gérer des caisses dans un entrepôt, ou un système de location de cassettes, ou un chenil pour animaux). Cela semble simple de prime abord : créer un conteneur pour stocker les avions, puis créer un nouvel avion et le placer dans le conteneur pour chaque avion qui entre dans la zone de contrôle du trafic aérien. Pour le nettoyage, il suffit de détruire l'objet avion correspondant lorsqu'un avion quitte la zone.

Mais supposons qu'une autre partie du système s'occupe d'enregistrer des informations à propos des avions, ces données ne requérant pas autant d'attention que la fonction principale de contrôle. Il s'agit peut-être d'enregistrer les plans de vol de tous les petits avions quittant l'aéroport. On dispose donc d'un second conteneur des petits avions, et quand on crée un objet avion on doit aussi le stocker dans le deuxième conteneur si c'est un petit avion. Une tâche de fond s'occupe de traiter les objets de ce conteneur durant les moments d'inactivité du système.

Le problème est maintenant plus compliqué : comment savoir quand détruire les objets ? Quand on en a fini avec un objet, une autre partie du système peut ne pas en avoir terminé avec. Ce genre de problème arrive dans un grand nombre de situations, et dans les systèmes de programmation (comme le C++) où les objets doivent être explicitement détruits cela peut devenir relativement complexe.

Avec Java, le ramasse-miettes est conçu pour s'occuper du problème de la libération de la mémoire (bien que cela n'inclut pas les autres aspects du nettoyage de l'objet). Le ramasse-miettes « sait » quand un objet n'est plus utilisé, et il libère automatiquement la mémoire utilisée par cet objet. Ceci (associé avec le fait que tous les objets sont dérivés de la classe de base fondamentale **Object** et que les objets sont créés dans le segment) rend la programmation Java plus simple que la programmation C++. Il y a beaucoup moins de décisions à prendre et d'obstacles à surmonter.

Ramasse-miettes vs. efficacité et flexibilité

Si cette idée est si bonne, pourquoi le C++ n'intègre-t-il pas ce mécanisme ? Bien sûr car il y a un prix à payer pour cette facilité de programmation, et ce surcoût se traduit par du temps système. Comme on l'a vu, en C++ on peut créer des objets dans la pile et dans ce cas ils sont automatiquement nettoyés (mais dans ce cas on n'a pas la flexibilité de créer autant d'objets que voulu lors de l'exécution). Créer des objets dans la pile est la façon la plus efficace d'allouer de l'espace pour des objets et de libérer cet espace. Créer des objets dans le segment est bien plus coûteux. Hériter de la même classe de base et rendre tous les appels de fonctions polymorphes prélèvent aussi un tribut. Mais le ramasse-miettes est un problème à part car on ne sait pas quand il va démarrer ou combien de temps il va prendre. Cela veut dire qu'il y a une inconsistance dans le temps d'exécution de programmes Java ; on ne peut donc l'utiliser dans certaines situations, comme celles où le temps d'exécution d'un programme est critique (appelés programmes en temps réel, bien que tous les problèmes de programmation en temps réel ne soient pas aussi astreignants).

Les concepteurs du langage C++, en voulant amadouer les programmeurs C, ne voulurent pas ajouter de nouvelles fonctionnalités au langage qui puissent impacter la vitesse ou défavoriser l'utilisation du C++ dans des situations où le C se serait révélé acceptable. Cet objectif a été atteint, mais au prix d'une plus grande complexité lorsqu'on programme en C++. Java est plus simple que le C++, mais la contrepartie en est l'efficacité et quelquefois son champ d'applications. Pour un grand nombre de problèmes de programmation cependant, Java constitue le meilleur choix.

Traitement des exceptions : gérer les erreurs

Depuis les débuts des langages de programmation, le traitement des erreurs s'est révélé l'un des problèmes les plus ardues. Parce qu'il est difficile de concevoir un bon mécanisme de gestion des erreurs, beaucoup de langages ignorent ce problème et le délèguent aux concepteurs de bibliothèques qui fournissent des mécanismes qui fonctionnent dans beaucoup de situations mais peuvent être facilement contournés, généralement en les ignorant. L'une des faiblesses de la plupart des mécanismes d'erreur est qu'ils reposent sur la vigilance du programmeur à suivre des conventions non imposées par le langage. Si le programmeur n'est pas assez vigilant - ce qui est souvent le cas s'il est pressé - ces mécanismes peuvent facilement être oubliés.

Le système des exceptions pour gérer les erreurs se situe au niveau du langage de programmation et parfois même au niveau du système d'exploitation. Une exception est un objet qui est « émis » depuis l'endroit où l'erreur est apparue et peut être intercepté par un gestionnaire d'exception conçu pour gérer ce type particulier d'erreur. C'est comme si la gestion des exceptions était un chemin d'exécution parallèle à suivre quand les choses se gâtent. Et parce qu'elle utilise un chemin d'exécution séparé, elle n'interfère pas avec le code s'exécutant normalement. Cela rend le code plus simple à écrire car on n'a pas à vérifier constamment si des erreurs sont survenues. De plus, une exception émise n'est pas comme une valeur de retour d'une fonction signalant une erreur ou un drapeau positionné par une fonction pour indiquer une erreur - ils peuvent être ignorés. Une exception ne peut pas être ignorée, on a donc l'assurance qu'elle sera traitée quelque part. Enfin, les exceptions permettent de revenir d'une mauvaise situation assez facilement. Plutôt que terminer un programme, il est souvent possible de remettre les choses en place et de restaurer son exécution, ce qui produit des programmes plus robustes.

Le traitement des exceptions de Java se distingue parmi les langages de programmes, car en Java le traitement des exceptions a été intégré depuis le début et on est forcé de l'utiliser. Si le code produit ne gère pas correctement les exceptions, le compilateur générera des messages d'erreur. Cette consistance rend la gestion des erreurs bien plus aisée.

Il est bon de noter que le traitement des exceptions n'est pas une caractéristique orientée objet, bien que dans les langages OO une exception soit normalement représentée par un objet. Le traitement des exceptions existait avant les langages orientés objet.

Multithreading

L'un des concepts fondamentaux dans la programmation des ordinateurs est l'idée de traiter plus d'une tâche à la fois. Beaucoup de problèmes requièrent que le programme soit capable de stopper ce qu'il est en train de faire, traite un autre problème puis retourne à sa tâche principale. Le problème a été abordé de beaucoup de manières différentes. Au début, les programmeurs ayant des connaissances sur le fonctionnement de bas niveau de la machine écrivaient des routines d'interruption de service et l'interruption de la tâche principale était initiée par une interruption matérielle. Bien que cela fonctionne correctement, c'était difficile et non portable, et cela rendait le portage d'un programme sur un nouveau type de machine lent et cher.

Quelquefois les interruptions sont nécessaires pour gérer les tâches critiques, mais il existe une large classe de problèmes dans lesquels on tente juste de partitionner le problème en parties séparées et indépendantes afin que le programme soit plus réactif. Dans un programme, ces parties séparées sont appelés threads et le concept général est appelé *multithreading*. Un exemple classique de multithreading est l'interface utilisateur. En utilisant les threads, un utilisateur peut appuyer sur un bouton et obtenir une réponse plus rapide que s'il devait attendre que le programme termine sa tâche courante.

Généralement, les threads sont juste une façon d'allouer le temps d'un seul processeur. Mais si le système d'exploitation supporte les multi-processeurs, chaque thread peut être assigné à un processeur différent et ils peuvent réellement s'exécuter en parallèle. L'une des caractéristiques intéressantes du multithreading au niveau du langage est que le programmeur n'a pas besoin de se préoccuper du nombre de processeurs. Le programme est divisé logiquement en threads et si la machine dispose de plusieurs processeurs, le programme tourne plus vite, sans aucun ajustement.

Tout ceci pourrait faire croire que le multithreading est simple. Il y a toutefois un point d'achoppement : les ressources partagées. Un problème se pose si plus d'un thread s'exécutant veulent accéder à la même ressource. Par exemple, deux tâches ne peuvent envoyer simultanément de l'information à une imprimante. Pour résoudre ce problème, les ressources pouvant être partagées, comme l'imprimante, doivent être verrouillées avant d'être utilisées. Un thread verrouille donc une ressource, accomplit ce qu'il a à faire, et ensuite relâche le verrou posé afin que quelqu'un d'autre puisse utiliser cette ressource.

Le multithreading Java est intégré au langage, ce qui simplifie grandement ce sujet compliqué. Le multithreading est supporté au niveau de l'objet, donc un thread d'exécution est représenté par un objet. Java fournit aussi des mécanismes limités de verrouillage de ressources. Il peut verrouiller la mémoire de n'importe quel objet (qui n'est, après tout, qu'un type de ressource partagée) de manière à ce qu'un seul thread puisse l'utiliser à un moment donné. Ceci est réalisé grâce au mot clef **synchronized**. D'autres types de ressources doivent être verrouillés explicitement par le programmeur, typiquement en créant un objet qui représente un verrou que tous les threads doivent vérifier avant d'accéder à cette ressource.

Persistence

Quand un objet est créé, il existe aussi longtemps qu'on en a besoin, mais en aucun cas il continue d'exister après que le programme se termine. Bien que cela semble logique de prime abord, il existe des situations où il serait très pratique si un objet pouvait continuer d'exister et garder son information même quand le programme ne tourne plus. A l'exécution suivante du programme, l'objet serait toujours là et il aurait la même information dont il disposait dans la session précédente. Bien sûr, on peut obtenir ce comportement en écrivant l'information dans un fichier ou une base de données, mais dans l'esprit du tout objet, il serait pratique d'être capable de déclarer un objet persistant et que tous les détails soient pris en charge.

Java fournit un support pour la « persistance légère », qui signifie qu'on peut facilement stocker des objets sur

disque et les récupérer plus tard. La raison pour laquelle on parle de « persistance légère » est qu'on est toujours obligé de faire des appels explicites pour le stockage et la récupération. De plus, JavaSpaces (décrit au Chapitre 15) fournit un type de stockage persistant des objets. Un support plus complet de la persistance peut être amené à apparaître dans de futures versions.

Java et l'Internet

Java n'étant qu'un autre langage de programmation, on peut se demander pourquoi il est si important et pourquoi il est présenté comme une étape révolutionnaire de la programmation. La réponse n'est pas immédiate si on se place du point de vue de la programmation classique. Bien que Java soit très pratique pour résoudre des problèmes traditionnels, il est aussi important car il permet de résoudre des problèmes liés au web.

Qu'est-ce que le Web ?

Le Web peut sembler un peu mystérieux au début, avec tout ce vocabulaire de « surf », « page personnelles », etc... Il y a même eu une réaction importante contre cette « Internet-mania », se questionnant sur la valeur économique et les revenus d'un tel engouement. Il peut être utile de revenir en arrière et voir de quoi il retourne, mais auparavant il faut comprendre le modèle client/serveur, un autre aspect de l'informatique relativement confus.

Le concept Client/Serveur

L'idée primitive d'un système client/serveur est qu'on dispose d'un dépôt centralisé de l'information - souvent dans une base de données - qu'on veut distribuer à un ensemble de personnes ou de machines. L'un des concepts majeurs du client/serveur est que le dépôt d'informations est centralisé afin qu'elles puissent être changées facilement et que ces changements se propagent jusqu'aux consommateurs de ces informations. Pris ensemble, le dépôt d'informations, le programme qui distribue l'information et la (les) machine(s) où résident informations et programme sont appelés le serveur. Le programme qui réside sur la machine distante, communique avec le serveur, récupère l'information, la traite et l'affiche sur la machine distante est appelé le *client*.

Le concept de base du client/serveur n'est donc pas si compliqué. Les problèmes surviennent car un seul serveur doit servir de nombreux clients à la fois. Généralement, un système de gestion de base de données est impliqué afin que le concepteur répartisse les informations dans des tables pour une utilisation optimale. De plus, les systèmes permettent généralement aux utilisateurs d'ajouter de nouvelles informations au serveur. Cela veut dire qu'ils doivent s'assurer que les nouvelles données d'un client ne contredisent pas les nouvelles données d'un autre client, ou que ces nouvelles données ne soient pas perdues pendant leur inscription dans la base de données (cela fait appel aux transactions). Comme les programmes d'application client changent, ils doivent être créés, débogués, et installés sur les machines clientes, ce qui se révèle plus compliqué et onéreux que ce à quoi on pourrait s'attendre. C'est particulièrement problématique dans les environnements hétérogènes comprenant différents types de matériels et de systèmes d'exploitation. Enfin, se pose toujours le problème des performances : on peut se retrouver avec des centaines de clients exécutant des requêtes en même temps, et donc un petit délai multiplié par le nombre de clients peut être particulièrement pénalisant. Pour minimiser les temps de latence, les programmeurs travaillent dur pour réduire la charge de travail des tâches incriminées, parfois jusque sur les machines clientes, mais aussi sur d'autres machines du site serveur, utilisant ce qu'on appelle le *middleware* (le middleware est aussi utilisé pour améliorer la maintenabilité).

L'idée relativement simple de distribuer de l'information aux gens a de si nombreux niveaux de complexité dans son implémentation que le problème peut sembler désespérément insoluble. Et pourtant il est crucial : le client/serveur compte pour environ la moitié des activités de programmation. On le retrouve pour tout ce qui va

des transactions de cartes de crédit à la distribution de n'importe quel type de données - économique, scientifique, gouvernementale, il suffit de choisir. Dans le passé, on en est arrivé à des solutions particulières aux problèmes particuliers, obligeant à réinventer une solution à chaque fois. Elles étaient difficiles à créer et à utiliser, et l'utilisateur devait apprendre une nouvelle interface pour chacune. Le problème devait être repris dans son ensemble.

Le Web en tant que serveur géant

Le Web est en fait un système client/serveur géant. C'est encore pire que ça, puisque tous les serveurs et les clients coexistent en même temps sur un seul réseau. Vous n'avez pas besoin de le savoir d'ailleurs, car tout ce dont vous vous souciez est de vous connecter et d'interagir avec un serveur à la fois (même si vous devez parcourir le monde pour trouver le bon serveur).

Initialement, c'était un processus à une étape. On faisait une requête à un serveur qui renvoyait un fichier, que le logiciel (ie, le client) interprétait en le formatant sur la machine locale. Mais les gens en voulurent plus : afficher des pages d'un serveur ne leur suffisait plus. Ils voulaient bénéficier de toutes les fonctionnalités du client/serveur afin que le client puisse renvoyer des informations au serveur, par exemple pour faire des requêtes précises dans la base de données, ajouter de nouvelles informations au serveur, ou passer des commandes (ce qui nécessitait plus de sécurité que ce que le système primitif offrait). Nous assistons à ces changements dans le développement du Web.

Le browser Web fut un grand bond en avant en avançant le concept qu'une information pouvait être affichée indifféremment sur n'importe quel ordinateur. Cependant, les browsers étaient relativement primitifs et ne remplissaient pas toutes les demandes des utilisateurs. Ils n'étaient pas particulièrement interactifs, et avaient tendance à saturer le serveur et l'Internet car à chaque fois qu'on avait besoin de faire quelque chose qui requerrait un traitement il fallait renvoyer les informations au serveur pour que celui-ci les traite. Cela pouvait prendre plusieurs secondes ou minutes pour finalement se rendre compte qu'on avait fait une faute de frappe dans la requête. Comme le browser n'était qu'un afficheur, il ne pouvait pas effectuer le moindre traitement (d'un autre côté, cela était beaucoup plus sûr, car il ne pouvait pas de ce fait exécuter sur la machine locale de programmes contenant des bugs ou des virus).

Pour résoudre ce problème, différentes solutions ont été approchées. En commençant par les standards graphiques qui ont été améliorés pour mettre des animations et de la vidéo dans les browsers. Le reste du problème ne peut être solutionné qu'en incorporant la possibilité d'exécuter des programmes sur le client, dans le browser. C'est ce qu'on appelle la programmation côté client.

La programmation côté client

Le concept initial du Web (serveur-browser) fournissait un contenu interactif, mais l'interactivité était uniquement le fait du serveur. Le serveur produisait des pages statiques pour le browser client, qui ne faisait que les interpréter et les afficher. Le HTML de base contient des mécanismes simples pour récupérer de l'information : des boîtes de texte, des cases à cocher, des listes à options et des listes de choix, ainsi qu'un bouton permettant de réinitialiser le contenu du formulaire ou d'envoyer les données du formulaire au serveur. Cette soumission de données passe par CGI (Common Gateway Interface), un mécanisme fourni par tous les serveurs web. Le texte de la soumission indique à CGI quoi faire avec ces données. L'action la plus commune consiste à exécuter un programme situé sur le serveur dans un répertoire typiquement appelé « cgi-bin » (si vous regardez l'adresse en haut de votre browser quand vous appuyez sur un bouton dans une page web, vous verrez certainement « cgi-bin » quelque part au milieu de tous les caractères). Ces programmes peuvent être écrits dans la plupart des langages. Perl est souvent préféré car il a été conçu pour la manipulation de texte et est interprété, et peut donc être installé sur n'importe quel serveur sans tenir compte du matériel ou du système d'exploitation.

Beaucoup de sites Web populaires sont aujourd'hui bâtis strictement sur CGI, et de fait on peut faire ce qu'on

veut avec. Cependant, les sites web bâtis sur des programmes CGI peuvent rapidement devenir ingérables et trop compliqués à maintenir, et se pose de plus le problème du temps de réponse. La réponse d'un programme CGI dépend de la quantité de données à envoyer, ainsi que de la charge du serveur et de l'Internet (de plus, le démarrage d'un programme CGI a tendance à être long). Les concepteurs du Web n'avaient pas prévu que la bande passante serait trop petite pour le type d'applications que les gens développeront. Par exemple, tout type de graphisme dynamique est impossible à réaliser correctement car un fichier GIF doit être créé et transféré du serveur au client pour chaque version de ce graphe. Et vous avez certainement déjà fait l'expérience de quelque chose d'aussi simple que la validation de données sur un formulaire. Vous appuyez sur le bouton « submit » d'une page, les données sont envoyées sur le serveur, le serveur démarre le programme CGI qui y découvre une erreur, formate une page HTML vous informant de votre erreur et vous la renvoie ; vous devez alors revenir en arrière d'une page et réessayer. Non seulement c'est lent et frustrant, mais c'est inélégant.

La solution est la programmation côté client. La plupart des machines qui font tourner des browsers Web sont des ordinateurs puissants capables de beaucoup de choses, et avec l'approche originale du HTML statique, elles ne font qu'attendre que le serveur leur envoie parcimonieusement la page suivante. La programmation côté client implique que le browser Web prenne en charge la partie du travail qu'il est capable de traiter, avec comme résultat pour l'utilisateur une interactivité et une vitesse inégalées.

Les problèmes de la programmation côté client ne sont guère différents des discussions de la programmation en général. Les paramètres sont quasiment les mêmes, mais la plateforme est différente : un browser Web est comme un système d'exploitation limité. Au final, on est toujours obligé de programmer, et ceci explique le nombre de problèmes rencontrés dans la programmation côté client. Le reste de cette section présente les différentes approches de la programmation côté client.

Les plug-ins

L'une des plus importantes avancées en terme de programmation orientée client a été le développement du plug-in. C'est une façon pour le programmeur d'ajouter de nouvelles fonctionnalités au browser en téléchargeant une partie de logiciel qui s'enfiche à l'endroit adéquat dans le browser. Il indique au browser : « à partir de maintenant tu peux réaliser cette nouvelle opération » (on n'a besoin de télécharger le plug-in qu'une seule fois). De nouveaux comportements puissants sont donc ajoutés au browser via les plug-ins, mais écrire un plug-in n'est pas une tâche facile, et ce n'est pas quelque chose qu'on souhaiterait faire juste pour bâtir un site Web particulier. La valeur d'un plug-in pour la programmation côté client est qu'il permet à un programmeur expérimenté de développer un nouveau langage et d'intégrer ce langage au browser sans la permission du distributeur du browser. Ainsi, les plug-ins fournissent une « porte dérobée » qui permet la création de nouveaux langages de programmation côté client (bien que tous les langages ne soient pas implémentés en tant que plug-ins).

Les langages de script

Les plug-ins ont déclenché une explosion de langages de script. Avec un langage de script, du code source est intégré directement dans la page HTML, et le plug-in qui interprète ce langage est automatiquement activé quand la page HTML est affichée. Les langages de script tendent à être relativement aisés à comprendre ; et parce qu'ils sont du texte faisant partie de la page HTML, ils se chargent très rapidement dans la même requête nécessaire pour se procurer la page auprès du serveur. La contrepartie en est que le code est visible pour tout le monde (qui peut donc le voler). Mais généralement on ne fait pas des choses extraordinairement sophistiquées avec les langages de script et ce n'est donc pas trop pénalisant.

Ceci montre que les langages de script utilisés dans les browsers Web sont vraiment spécifiques à certains types de problèmes, principalement la création d'interfaces utilisateurs plus riches et attrayantes. Cependant, un langage de script permet de résoudre 80 pour cent des problèmes rencontrés dans la programmation côté client. La plupart de vos problèmes devraient se trouver parmi ces 80 pour cent, et puisque les langages de script sont

plus faciles à mettre en oeuvre et permettent un développement plus rapide, vous devriez d'abord vérifier si un langage de script ne vous suffirait pas avant de vous lancer dans une solution plus complexe telle que la programmation Java ou ActiveX.

Les langages de script les plus répandus parmi les browsers sont JavaScript (qui n'a rien à voir avec Java ; le nom a été choisi uniquement pour bénéficier de l'engouement marketing pour Java du moment), VBScript (qui ressemble à Visual Basic), et Tcl/Tk, qui vient du fameux langage de construction d'interfaces graphiques. Il y en a d'autres bien sûr, et sans doute encore plus en développement.

JavaScript est certainement le plus commun d'entre eux. Il est présent dès l'origine dans Netscape Navigator et Microsoft Internet Explorer (IE). De plus, il y a certainement plus de livres disponibles sur JavaScript que sur les autres langages, et certains outils créent automatiquement des pages utilisant JavaScript. Cependant, si vous parlez couramment Visual Basic ou Tcl/Tk, vous serez certainement plus productif en utilisant ces langages qu'en en apprenant un nouveau (vous aurez déjà largement de quoi faire avec les problèmes soulevés par le Web).

Java

Si un langage de script peut résoudre 80 pour cent des problèmes de la programmation côté client, qu'en est-il des 20 pour cent restants ? La solution la plus populaire aujourd'hui est Java. C'est non seulement un langage de programmation puissant conçu pour être sûr, interplateformes et international, mais Java est continuellement étendu pour fournir un langage proposant des caractéristiques et des bibliothèques permettant de gérer de manière élégante des problèmes traditionnellement complexes dans les langages de programmation classiques, tels que le multithreading, les accès aux bases de données, la programmation réseau, l'informatique répartie. Java permet la programmation côté client via les *applets*.

Une applet est un mini-programme qui ne tourne que dans un browser Web. L'applet est téléchargée automatiquement en temps que partie d'une page Web (comme, par exemple, une image est automatiquement téléchargée). Quand l'applet est activée elle exécute un programme. Là se trouve la beauté de la chose : cela vous permet de distribuer le logiciel client à partir du serveur au moment où l'utilisateur en a besoin et pas avant. L'utilisateur récupère toujours la dernière version en date du logiciel et sans avoir besoin de le réinstaller. De par la conception même de Java, le programmeur n'a besoin de créer qu'un seul programme, et ce programme tournera automatiquement sur tous les browsers disposant d'un interpréteur interne (ce qui inclut la grande majorité des machines). Puisque Java est un langage de programmation complet, on peut déporter une grande partie des traitements sur le client avant et après avoir envoyé les requêtes au serveur. Par exemple, une requête comportant une erreur dans une date ou utilisant un mauvais paramètre n'a plus besoin d'être envoyée sur Internet pour se voir refusée par le serveur ; un client peut très bien aussi s'occuper de gérer l'affichage d'un nouveau point sur un graphe plutôt que d'attendre que le serveur ne s'en occupe, crée une nouvelle image et l'envoie au client. On gagne ainsi non seulement en vitesse et confort d'utilisation, mais le trafic engendré sur le réseau et la charge résultante sur les serveurs peut être réduite, ce qui est bénéfique pour l'Internet dans son ensemble.

L'un des avantages qu'une applet Java a sur un programme écrit dans un langage de script est qu'il est fourni sous une forme précompilée, et donc le code source n'est pas disponible pour le client. D'un autre côté, une applet Java peut être rétroingéniérée sans trop de problèmes, mais cacher son code n'est pas souvent une priorité absolue. Deux autres facteurs peuvent être importants. Comme vous le verrez plus tard dans ce livre, une applet Java compilée peut comprendre plusieurs modules et nécessiter plusieurs accès au serveur pour être téléchargée (à partir de Java 1.1 cela est minimisé par les archives Java ou fichiers JAR, qui permettent de paqueter tous les modules requis ensemble dans un format compressé pour un téléchargement unique). Un programme scripté sera juste inséré dans le texte de la page Web (et sera généralement plus petit et réduira le nombre d'accès au serveur). Cela peut être important pour votre site Web. Un autre facteur à prendre en compte est la courbe d'apprentissage. Indépendamment de tout ce que vous avez pu entendre, Java n'est pas un langage trivial et

facile à apprendre. Si vous avez l'habitude de programmer en Visual Basic, passer à VBScript sera beaucoup plus rapide et comme cela permet de résoudre la majorité des problèmes typiques du client/serveur, il pourrait être difficile de justifier l'investissement de l'apprentissage de Java. Si vous êtes familier avec un langage de script, vous serez certainement gagnant en vous tournant vers JavaScript ou VBScript avant Java, car ils peuvent certainement combler vos besoins et vous serez plus productif plus tôt.

ActiveX

A un certain degré, le compétiteur de Java est ActiveX de Microsoft, bien qu'il s'appuie sur une approche radicalement différente. ActiveX était au départ une solution disponible uniquement sur Windows, bien qu'il soit actuellement développé par un consortium indépendant pour devenir interplateformes. Le concept d'ActiveX clame : « si votre programme se connecte à son environnement de cette façon, il peut être inséré dans une page Web et exécuté par un browser qui supporte ActiveX » (IE supporte ActiveX en natif et Netscape utilise un plug-in). Ainsi, ActiveX ne vous restreint pas à un langage particulier. Si par exemple vous êtes un programmeur Windows expérimenté utilisant un langage tel que le C++, Visual Basic ou Delphi de Borland, vous pouvez créer des composants ActiveX sans avoir à tout réapprendre. ActiveX fournit aussi un mécanisme pour la réutilisation de code dans les pages Web.

La sécurité

Le téléchargement automatique et l'exécution de programmes sur Internet ressemble au rêve d'un concepteur de virus. ActiveX en particulier pose la question épineuse de la sécurité dans la programmation côté client. Un simple click sur un site Web peut déclencher le téléchargement d'un certain nombre de fichiers en même temps que la page HTML : des images, du code scripté, du code Java compilé, et des composants ActiveX. Certains d'entre eux sont sans importance : les images ne peuvent causer de dommages, et les langages de script sont généralement limités dans les opérations qu'ils sont capables de faire. Java a été conçu pour exécuter ses applets à l'intérieur d'un périmètre de sécurité (appelé bac à sable), qui l'empêche d'aller écrire sur le disque ou d'accéder à la mémoire en dehors de ce périmètre de sécurité.

ActiveX se trouve à l'opposé du spectre. Programmer avec ActiveX est comme programmer sous Windows - on peut faire ce qu'on veut. Donc un composant ActiveX téléchargé en même temps qu'une page Web peut causer bien des dégâts aux fichiers du disque. Bien sûr, les programmes téléchargés et exécutés en dehors du browser présentent les mêmes risques. Les virus téléchargés depuis les BBS ont pendant longtemps été un problème, mais la vitesse obtenue sur Internet accroît encore les difficultés.

La solution semble être les « signatures digitales », où le code est vérifié pour montrer qui est l'auteur. L'idée est basée sur le fait qu'un virus se propage parce que son concepteur peut rester anonyme, et si cet anonymat lui est retiré, l'individu devient responsable de ses actions. Mais si le programme contient un bug fatal bien que non intentionnel cela pose toujours problème. L'approche de Java est de prévenir ces problèmes via le périmètre de sécurité. L'interpréteur Java du browser Web examine l'applet pendant son chargement pour y détecter les instructions interdites. En particulier, les applets ne peuvent écrire sur le disque ou effacer des fichiers (ce que font la plupart des virus). Les applets sont généralement considérées comme sûres ; et comme ceci est essentiel pour bénéficier d'un système client/serveur fiable, les bugs découverts dans Java permettant la création de virus sont très rapidement corrigés (il est bon de noter que le browser renforce ces restrictions de sécurité, et que certains d'entre eux permettent de sélectionner différents niveaux de sécurité afin de permettre différents niveaux d'accès au système).

On pourrait s'interroger sur cette restriction draconienne contre les accès en écriture sur le disque local. Par exemple, on pourrait vouloir construire une base de données locale ou sauvegarder des données pour un usage futur en mode déconnecté. La vision initiale obligeait tout le monde à se connecter pour réaliser la moindre opération, mais on s'est vite rendu compte que cela était impraticable. La solution a été trouvée avec les « applets signées » qui utilisent le chiffrement avec une clef publique pour vérifier qu'une applet provient bien de

là où elle dit venir. Une applet signée peut toujours endommager vos données et votre ordinateur, mais on en revient à la théorie selon laquelle la perte de l'anonymat rend le créateur de l'applet responsable de ses actes, il ne fera donc rien de préjudiciable. Java fournit un environnement pour les signatures digitales afin de permettre à une applet de sortir du périmètre de sécurité si besoin est.

Les signatures digitales ont toutefois oublié un paramètre important, qui est la vitesse à laquelle les gens bougent sur Internet. Si on télécharge un programme buggué et qu'il accomplisse ses méfaits, combien de temps se passera-t-il avant que les dommages ne soient découverts ? Cela peut se compter en jours ou en semaines. Et alors, comment retrouver le programme qui les a causés ? Et en quoi cela vous aidera à ce point ?

Internet vs. intranet

Le Web est la solution la plus générique au problème du client/serveur, il est donc logique de vouloir appliquer la même technologie pour résoudre ceux liés aux client/serveur à *l'intérieur* d'une entreprise. Avec l'approche traditionnelle du client/serveur se pose le problème de l'hétérogénéité du parc de clients, de même que les difficultés pour installer les nouveaux logiciels clients. Ces problèmes sont résolus par les browsers Web et la programmation côté client. Quand la technologie du Web est utilisée dans le système d'information d'une entreprise, on s'y réfère en tant qu'intranet. Les intranets fournissent une plus grande sécurité qu'Internet puisqu'on peut contrôler les accès physiques aux serveurs à l'intérieur de l'entreprise. En terme d'apprentissage, il semblerait qu'une fois que les utilisateurs se sont familiarisés avec le concept d'un browser il leur est plus facile de gérer les différences de conception entre les pages et applets, et le coût d'apprentissage de nouveaux systèmes semble se réduire.

Le problème de la sécurité nous mène à l'une des divisions qui semble se former automatiquement dans le monde de la programmation côté client. Si un programme est distribué sur Internet, on ne sait pas sur quelle plateforme il va tourner et il faut être particulièrement vigilant pour ne pas diffuser du code buggué. Il faut une solution multiplateformes et sûre, comme un langage de script ou Java.

Dans le cas d'un intranet, les contraintes peuvent être complètement différentes. Le temps passé à installer les nouvelles versions est la raison principale qui pousse à passer aux browsers, car les mises à jours sont invisibles et automatiques. Il n'est pas rare que tous les clients soient des plateformes Wintel (Intel / Windows). Dans un intranet, on est responsable de la qualité du code produit et on peut corriger les bugs quand ils sont découverts. De plus, on dispose du code utilisé dans une approche plus traditionnelle du client/serveur où il fallait installer physiquement le client à chaque mise à jour du logiciel. Dans le cas d'un tel intranet, l'approche la plus raisonnable consiste à choisir la solution qui permettra de réutiliser le code existant plutôt que de repartir de zéro dans un nouveau langage.

Quand on est confronté au problème de la programmation côté client, la meilleure chose à faire est une analyse coûts-bénéfices. Il faut prendre en compte les contraintes du problème, et la solution qui serait la plus rapide à mettre en oeuvre. En effet, comme la programmation côté client reste de la programmation, c'est toujours une bonne idée de choisir l'approche permettant un développement rapide.

La programmation côté serveur

Toute cette discussion a passé sous silence la programmation côté serveur. Que se passe-t-il lorsqu'un serveur reçoit une requête ? La plupart du temps cette requête est simplement « envoie-moi ce fichier ». Le browser interprète alors ce fichier d'une manière particulière : comme une page HTML, une image graphique, une applet Java, un programme script, etc... Une requête plus compliquée à un serveur implique généralement une transaction vers une base de données. Un scénario classique consiste en une requête complexe de recherche d'enregistrements dans une base de données que le serveur formate dans une page HTML et renvoie comme résultat (bien sûr, si le client est « intelligent » via Java ou un langage de script, la phase de formatage peut être réalisée par le client et le serveur n'aura qu'à envoyer les données brutes, ce qui allégera la charge et le trafic

engendré). Ou alors un client peut vouloir s'enregistrer dans une base de données quand il joint un groupe ou envoie une commande, ce qui implique des changements dans la base de données. Ces requêtes doivent être traitées par du code s'exécutant sur le serveur, c'est ce qu'on appelle la programmation côté serveur. Traditionnellement, la programmation côté serveur s'est appuyée sur Perl et les scripts CGI, mais des systèmes plus sophistiqués sont en train de voir le jour. Cela inclut des serveurs Web écrits en Java qui permettent de réaliser toute la programmation côté serveur en Java en écrivant ce qu'on appelle des servlets. Les servlets et leur progéniture, les JSPs, sont les deux raisons principales qui poussent les entreprises qui développent un site Web à se tourner vers Java, en particulier parce qu'ils éliminent les problèmes liés aux différences de capacité des browsers.

Une scène séparée : les applications

La plupart de la publicité faite autour de Java se référait aux applets. Mais Java est aussi un langage de programmation à vocation plus générale qui peut résoudre n'importe quel type de problème - du moins en théorie. Et comme précisé plus haut, il peut y avoir des moyens plus efficaces de résoudre la plupart des problèmes client/serveur. Quand on quitte la scène des applets (et les restrictions qui y sont liées telles que les accès en écriture sur le disque), on entre dans le monde des applications qui s'exécutent sans le soutien d'un browser, comme n'importe quel programme. Les atouts de Java sont là encore non seulement sa portabilité, mais aussi sa facilité de programmation. Comme vous allez le voir tout au long de ce livre, Java possède beaucoup de fonctionnalités qui permettent de créer des programmes robustes dans un laps de temps plus court qu'avec d'autres langages de programmation.

Mais il faut bien être conscient qu'il s'agit d'un compromis. Le prix à payer pour ces améliorations est une vitesse d'exécution réduite (bien que des efforts significatifs soient mis en oeuvre dans ce domaine - JDK 1.3, en particulier, introduit le concept d'amélioration de performances « hotspot »). Comme tout langage de programmation, Java a des limitations internes qui peuvent le rendre inadéquat pour résoudre certains types de problèmes. Java évolue rapidement cependant, et à chaque nouvelle version il permet d'adresser un spectre de plus en plus large de problèmes.

Analyse et conception

Le paradigme de la POO constitue une approche nouvelle et différente de la programmation. Beaucoup de personnes rencontrent des difficultés pour appréhender leur premier projet orienté objet. Une fois compris que tout est supposé être un objet, et au fur et à mesure qu'on se met à penser dans un style plus orienté objet, on commence à créer de « bonnes » conceptions qui s'appuient sur tous les avantages que la POO offre.

Une *méthode* (ou méthodologie) est un ensemble de processus et d'heuristiques utilisés pour réduire la complexité d'un problème. Beaucoup de méthodes orientées objet ont été formulées depuis l'apparition de la POO. Cette section vous donne un aperçu de ce que vous essayez d'accomplir en utilisant une méthode.

Une méthodologie s'appuie sur un certain nombre d'expériences, il est donc important de comprendre quel problème la méthode tente de résoudre avant d'en adopter une. Ceci est particulièrement vrai avec Java, qui a été conçu pour réduire la complexité (comparé au C) dans l'expression d'un programme. Cette philosophie supprime le besoin de méthodologies toujours plus complexes. Des méthodologies simples peuvent se révéler tout à fait suffisantes avec Java pour une classe de problèmes plus large que ce qu'elles ne pourraient traiter avec des langages procéduraux.

Il est important de réaliser que le terme «> méthodologie » est trompeur et promet trop de choses. Tout ce qui est mis en oeuvre quand on conçoit et réalise un programme est une méthode. Ca peut être une méthode personnelle, et on peut ne pas en être conscient, mais c'est une démarche qu'on suit au fur et à mesure de l'avancement du projet. Si cette méthode est efficace, elle ne nécessitera sans doute que quelques petites

adaptations pour fonctionner avec Java. Si vous n'êtes pas satisfait de votre productivité ou du résultat obtenu, vous serez peut-être tenté d'adopter une méthode plus formelle, ou d'en composer une à partir de plusieurs méthodes formelles.

Au fur et à mesure que le projet avance, le plus important est de ne pas se perdre, ce qui est malheureusement très facile. La plupart des méthodes d'analyse et de conception sont conçues pour résoudre même les problèmes les plus gros. Il faut donc bien être conscient que la plupart des projets ne rentrant pas dans cette catégorie, on peut arriver à une bonne analyse et conception avec juste une petite partie de ce qu'une méthode recommande [8]. Une méthode de conception, même limitée, met sur la voie bien mieux que si on commence à coder directement.

Il est aussi facile de rester coincé et tomber dans la « paralysie analytique » où on se dit qu'on ne peut passer à la phase suivante car on n'a pas traqué le moindre petit détail de la phase courante. Il faut bien se dire que quelle que soit la profondeur de l'analyse, certains aspects d'un problème ne se révéleront qu'en phase de conception, et d'autres en phase de réalisation, voire même pas avant que le programme ne soit achevé et exécuté. A cause de ceci, il est crucial d'avancer relativement rapidement dans l'analyse et la conception, et d'implémenter un test du système proposé.

Il est bon de développer un peu ce point. A cause des déboires rencontrés avec les langages procéduraux, il est louable qu'une équipe veuille avancer avec précautions et comprendre tous les détails avant de passer à la conception et l'implémentation. Il est certain que lors de la création d'une base de données, il est capital de comprendre à fond les besoins du client. Mais la conception d'une base de données fait partie d'une classe de problèmes bien définie et bien comprise ; dans ce genre de programmes, la structure de la base de données *est* le problème à résoudre. Les problèmes traités dans ce chapitre font partie de la classe de problèmes « joker » (invention personnelle), dans laquelle la solution n'est pas une simple reformulation d'une solution déjà éprouvée de nombreuses fois, mais implique un ou plusieurs « facteurs joker » - des éléments pour lesquels il n'existe aucune solution préétablie connue, et qui nécessitent de pousser les recherches [9]. Tenter d'analyser à fond un problème joker avant de passer à la conception et l'implémentation mène à la paralysie analytique parce qu'on ne dispose pas d'assez d'informations pour résoudre ce type de problèmes durant la phase d'analyse. Résoudre ce genre de problèmes requiert de répéter le cycle complet, et cela demande de prendre certains risques (ce qui est sensé, car on essaie de faire quelque chose de nouveau et les revenus potentiels en sont plus élevés). On pourrait croire que le risque est augmenté par cette ruée vers une première implémentation, mais elle peut réduire le risque dans un projet joker car on peut tout de suite se rendre compte si telle approche du problème est viable ou non. Le développement d'un produit s'apparente à de la gestion de risque.

Souvent cela se traduit par « construire un prototype qu'il va falloir jeter ». Avec la POO, on peut encore avoir à en jeter *une partie*, mais comme le code est encapsulé dans des classes, on aura inévitablement produit durant la première passe quelques classes qui valent la peine d'être conservées et développé des idées sur la conception du système. Ainsi, une première passe rapide sur un problème non seulement fournit des informations critiques pour la prochaine passe d'analyse, de conception et d'implémentation, mais elle produit aussi une base du code. Ceci dit, si on cherche une méthode qui contient de nombreux détails et suggère de nombreuses étapes et documents, il est toujours difficile de savoir où s'arrêter. Il faut garder à l'esprit ce qu'on essaye de découvrir :

1. Quels sont les objets ? (Comment partitionner le projet en ses composants élémentaires ?)
2. Quelles en sont leur interface ? (Quels sont les messages qu'on a besoin d'envoyer à chaque objet ?)

Si on arrive à trouver quels sont les objets et leur interface, alors on peut commencer à coder. On pourra avoir besoin d'autres descriptions et documents, mais on ne peut pas faire avec moins que ça.

Le développement peut être décomposé en cinq phases, et une Phase 0 qui est juste l'engagement initial à respecter une structure de base.

Phase 0 : Faire un plan

Il faut d'abord décider quelles étapes on va suivre dans le développement. Cela semble simple (en fait, *tout* semble simple), et malgré tout les gens ne prennent cette décision qu'après avoir commencé à coder. Si le plan se résume à « retrouvons nos manches et codons », alors ça ne pose pas de problèmes (quelquefois c'est une approche valable quand on a affaire à un problème bien connu). Mais il faut néanmoins accepter que c'est le plan.

On peut aussi décider dans cette phase qu'une structure additionnelle est nécessaire. Certains programmeurs aiment travailler en « mode vacances », sans structure imposée sur le processus de développement de leur travail : « Ce sera fait lorsque ce sera fait ». Cela peut être séduisant un moment, mais disposer de quelques jalons aide à se concentrer et focalise les efforts sur ces jalons au lieu d'être obnubilé par le but unique de « finir le projet ». De plus, cela divise le projet en parties plus petites, ce qui le rend moins redoutable (sans compter que les jalons offrent des opportunités de fête).

Quand j'ai commencé à étudier la structure des histoires (afin de pouvoir un jour écrire un roman), j'étais réticent au début à l'idée de structure, trouvant que j'écrivais mieux quand je laissais juste la plume courir sur le papier. Mais j'ai réalisé plus tard que quand j'écris à propos des ordinateurs, la structure est suffisamment claire pour moi pour que je n'ai pas besoin de trop y réfléchir. Mais je structure tout de même mon travail, bien que ce soit inconsciemment dans ma tête. Même si on pense que le plan est juste de commencer à coder, on passe tout de même par les phases successives en se posant certaines questions et en y répondant.

L'exposé de la mission

Tout système qu'on construit, quelle que soit sa complexité, a un but, un besoin fondamental qu'il satisfait. Si on peut voir au delà de l'interface utilisateur, des détails spécifiques au matériel - ou au système -, des algorithmes de codage et des problèmes d'efficacité, on arrive finalement au coeur du problème - simple et nu. Comme le soi-disant *concept fondamental* d'un film hollywoodien, on peut le décrire en une ou deux phrases. Cette description pure est le point de départ.

Le concept fondamental est assez important car il donne le ton du projet ; c'est l'exposé de la mission. Ce ne sera pas nécessairement le premier jet qui sera le bon (on peut être dans une phase ultérieure du projet avant qu'il ne soit complètement clair), mais il faut continuer d'essayer jusqu'à ce que ça sonne bien. Par exemple, dans un système de contrôle de trafic aérien, on peut commencer avec un concept fondamental basé sur le système qu'on construit : « Le programme tour de contrôle garde la trace d'un avion ». Mais cela n'est plus valable quand le système se réduit à un petit aérodrome, avec un seul contrôleur, ou même aucun. Un modèle plus utile ne décrira pas tant la solution qu'on crée que le problème : « Des avions arrivent, déchargent, partent en révision, rechargent et repartent ».

Phase 1 : Que construit-on ?

Dans la génération précédente de conception de programmes (*conception procédurale*), cela s'appelait « l'analyse des besoins et les spécifications du système ». C'étaient des endroits où on se perdait facilement, avec des documents au nom intimidant qui pouvaient occulter le projet. Leurs intentions étaient bonnes, pourtant. L'analyse des besoins consiste à « faire une liste des indicateurs qu'on utilisera pour savoir quand le travail sera terminé et le client satisfait ». Les spécifications du système consistent en « une description de ce que le programme fera (sans se préoccuper du *comment*) pour satisfaire les besoins ». L'analyse des besoins est un contrat entre le développeur et le client (même si le client travaille dans la même entreprise, ou se trouve être un objet ou un autre système). Les spécifications du système sont une exploration générale du problème, et en un sens permettent de savoir s'il peut être résolu et en combien de temps. Comme ils requièrent des consensus entre

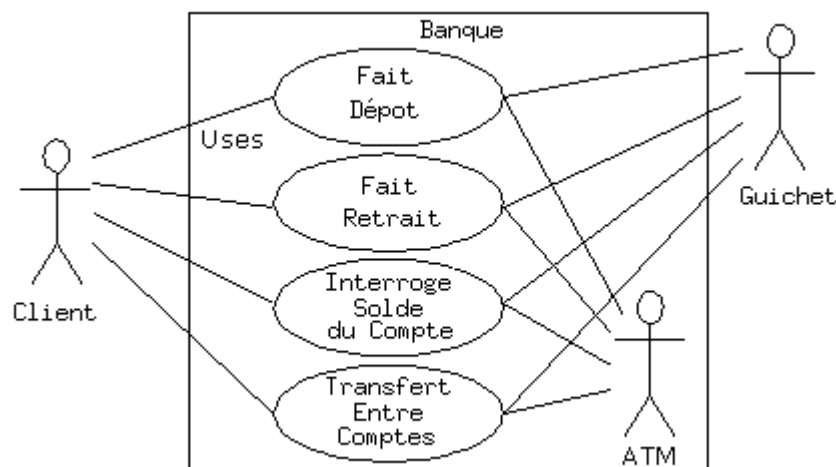
les intervenants sur le projet (et parce qu'ils changent au cours du temps), il vaut mieux les garder aussi bruts que possible - idéalement en tant que listes et diagrammes - pour ne pas perdre de temps. Il peut y avoir d'autres contraintes qui demandent de produire de gros documents, mais en gardant les documents initiaux petits et concis, cela permet de les créer en quelques sessions de brainstorming avec un leader qui affine la description dynamiquement. Cela permet d'impliquer tous les acteurs du projet, et encourage la participation de toute l'équipe. Plus important encore, cela permet de lancer un projet dans l'enthousiasme.

Il est nécessaire de rester concentré sur ce qu'on essaye d'accomplir dans cette phase : déterminer ce que le système est supposé faire. L'outil le plus utile pour cela est une collection de ce qu'on appelle « cas d'utilisation ». Les cas d'utilisation identifient les caractéristiques clefs du système qui vont révéler certaines des classes fondamentales qu'on utilisera. Ce sont essentiellement des réponses descriptives à des questions comme [\[10\]](#) :

- « Qui utilisera le système ? »
- « Que peuvent faire ces personnes avec le système ? »
- « Comment *tel* acteur fait-il *cela* avec le système ? »
- « Comment cela pourrait-il fonctionner si quelqu'un d'autre faisait cela, ou si le même acteur avait un objectif différent ? » (pour trouver les variations)
- « Quels problèmes peuvent apparaître quand on fait cela avec le système ? » (pour trouver les exceptions)

Si on conçoit un guichet automatique, par exemple, le cas d'utilisation pour un aspect particulier des fonctionnalités du système est capable de décrire ce que le guichet fait dans chaque situation possible. Chacune de ces situations est appelée un *scénario*, et un cas d'utilisation peut être considéré comme une collection de scénarios. On peut penser à un scénario comme à une question qui commence par « Qu'est-ce que le système fait si... ? ». Par exemple, « Qu'est-ce que le guichet fait si un client vient de déposer un chèque dans les dernières 24 heures, et qu'il n'y a pas assez dans le compte sans que le chèque soit encaissé pour fournir le retrait demandé ? ».

Les diagrammes de cas d'utilisations sont voulus simples pour ne pas se perdre prématurément dans les détails de l'implémentation du système :



Chaque bonhomme représente un « acteur », typiquement une personne ou une autre sorte d'agent (cela peut même être un autre système informatique, comme c'est le cas avec « ATM »). La boîte représente les limites du système. Les ellipses représentent les cas d'utilisation, qui sont les descriptions des actions qui peuvent être réalisées avec le système. Les lignes entre les acteurs et les cas d'utilisation représentent les interactions.

Tant que le système est perçu ainsi par l'utilisateur, son implémentation n'est pas importante.

Un cas d'utilisation n'a pas besoin d'être complexe, même si le système sous-jacent l'est. Il est seulement destiné à montrer le système tel qu'il apparaît à l'utilisateur. Par exemple :



Les cas d'utilisation produisent les spécifications des besoins en déterminant toutes les interactions que l'utilisateur peut avoir avec le système. Il faut trouver un ensemble complet de cas d'utilisations du système, et cela terminé on se retrouve avec le coeur de ce que le système est censé faire. La beauté des cas d'utilisation est qu'ils ramènent toujours aux points essentiels et empêchent de se disperser dans des discussions non essentielles à la réalisation du travail à faire. Autrement dit, si on dispose d'un ensemble complet de cas d'utilisation, on peut décrire le système et passer à la phase suivante. Tout ne sera pas parfaitement clair dès le premier jet, mais ça ne fait rien. Tout se décantera avec le temps, et si on cherche à obtenir des spécifications du système parfaites à ce point on se retrouvera coincé.

Si on est bloqué, on peut lancer cette phase en utilisant un outil d'approximation grossier : décrire le système en quelques paragraphes et chercher les noms et les verbes. Les noms suggèrent les acteurs, le contexte des cas d'utilisation ou les objets manipulés dans les cas d'utilisation. Les verbes suggèrent les interactions entre les acteurs et les cas d'utilisation, et spécifient les étapes à l'intérieur des cas d'utilisation. On verra aussi que les noms et les verbes produisent des objets et des messages durant la phase de design (on peut noter que les cas d'utilisation décrivent les interactions entre les sous-systèmes, donc la technique « des noms et des verbes » ne peut être utilisée qu'en tant qu'outil de brainstorming car il ne fournit pas les cas d'utilisation) [11].

La frontière entre un cas d'utilisation et un acteur peut révéler l'existence d'une interface utilisateur, mais ne définit pas cette interface utilisateur. Pour une méthode de définition et de création d'interfaces utilisateur, se référer à *Software for Use* de Larry Constantine et Lucy Lockwood, (Addison-Wesley Longman, 1999) ou sur www.ForUse.com.

Bien que cela tienne plus de l'art obscur, à ce point un calendrier de base est important. On dispose maintenant d'une vue d'ensemble de ce qu'on construit, on peut donc se faire une idée du temps nécessaire à sa réalisation. Un grand nombre de facteurs entre en jeu ici. Si on estime un calendrier trop important, l'entreprise peut décider d'abandonner le projet (et utiliser leurs ressources sur quelque chose de plus raisonnable - ce qui est une *bonne* chose). Ou un directeur peut avoir déjà décidé du temps que le projet devrait prendre et voudra influencer les estimations. Mais il vaut mieux proposer un calendrier honnête et prendre les décisions importantes au début. Beaucoup de techniques pour obtenir des calendriers précis ont été proposées (de même que pour prédire l'évolution de la bourse), mais la meilleure approche est probablement de se baser sur son expérience et son intuition. Proposer une estimation du temps nécessaire pour réaliser le système, puis doubler cette estimation et ajouter 10 pour cent. L'estimation initiale est probablement correcte, on *peut* obtenir un système fonctionnel avec ce temps. Le doublement transforme le délai en quelque chose de décent, et les 10 pour cent permettront de poser le vernis final et de traiter les détails [12]. Peu importe comment on l'explique et les gémissements obtenus quand on révèle un tel planning, il semble juste que ça fonctionne de cette façon.

Phase 2 : Comment allons-nous le construire ?

Dans cette phase on doit fournir une conception qui décrive ce à quoi les classes ressemblent et comment elles interagissent. Un bon outil pour déterminer les classes et les interactions est la méthode des

cartes *Classes-Responsabilités-Collaboration* (CRC). L'un des avantages de cet outil est sa simplicité : on prend des cartes vierges et on écrit dessus au fur et à mesure. Chaque carte représente une classe, et sur la carte on écrit :

1. Le nom de la classe. Il est important que le nom de cette classe reflète l'essence de ce que la classe fait, afin que sa compréhension soit immédiate.
2. Les « responsabilités » de la classe : ce qu'elle doit faire. Typiquement, cela peut être résumé par le nom des fonctions membres (puisque ces noms doivent être explicites dans une bonne conception), mais cela n'empêche pas de compléter par d'autres notes. Pour s'aider, on peut se placer du point de vue d'un programmeur fainéant : quels objets voudrait-on voir apparaître pour résoudre le problème ?
3. Les « collaborations » de la classe : avec quelles classes interagit-elle ?
« Interagir » est intentionnellement évasif, il peut se référer à une agrégation ou indiquer qu'un autre objet existant va travailler pour le compte d'un objet de la classe. Les collaborations doivent aussi prendre en compte l'audience de cette classe. Par exemple, si on crée une classe **Pétard**, qui va l'observer, un **Chimiste** ou un **Spectateur** ? Le premier voudra connaître la composition chimique, tandis que le deuxième sera préoccupé par les couleurs et le bruit produits quand il explose.

On pourrait se dire que les cartes devraient être plus grandes à cause de toutes les informations qu'on aimerait mettre dessus, mais il vaut mieux les garder les plus petites possibles, non seulement pour concevoir de petites classes, mais aussi pour éviter de plonger trop tôt dans les détails. Si on ne peut pas mettre toutes les informations nécessaires à propos d'une classe sur une petite carte, la classe est trop complexe (soit le niveau de détails est trop élevé, soit il faut créer plus d'une classe). La classe idéale doit être comprise en un coup d'oeil. L'objectif des cartes CRC est de fournir un premier jet de la conception afin de saisir le plan général pour pouvoir ensuite affiner cette conception.

L'un des avantages des cartes CRC réside dans la communication. Il vaut mieux les réaliser en groupe, sans ordinateur. Chacun prend le rôle d'une ou plusieurs classes (qui au début n'ont pas de nom ni d'information associée). Il suffit alors de dérouler une simulation impliquant un scénario à la fois, et décider quels messages sont envoyés aux différents objets pour satisfaire chaque scénario. Au fur et à mesure du processus, on découvre quelles sont les classes nécessaires, leurs responsabilités et collaborations, et on peut remplir les cartes. Quand tous les scénarios ont été couverts, on devrait disposer d'une bonne approximation de la conception.

Avant d'utiliser les cartes CRC, la meilleure conception initiale que j'ai fourni sur un projet fut obtenue en dessinant des objets sur un tableau devant une équipe - qui n'avait jamais participé à un projet de POO auparavant. Nous avons discuté de la communication entre ces objets, effacé et remplacé certains d'entre eux par d'autres objets. De fait, je recréais la méthode des cartes CRC au tableau. L'équipe (qui connaissait ce que le projet était censé faire) a effectivement créé la conception ; et de ce fait ils la contrôlaient. Je me contentais de guider le processus en posant les bonnes questions, proposant quelques hypothèses et utilisant les réponses de l'équipe pour modifier ces hypothèses. La beauté de la chose fut que l'équipe a appris à bâtir une conception orientée objet non en potassant des exemples abstraits, mais en travaillant sur la conception qui les intéressait au moment présent : celle de leur projet.

Une fois qu'on dispose d'un ensemble de cartes CRC, on peut vouloir une description plus formelle de la conception en utilisant UML [\[13\]](#). L'utilisation d'UML n'est pas une obligation, mais cela peut être utile, surtout si on veut afficher au mur un diagramme auquel tout le monde puisse se référer, ce qui est une bonne idée. Une alternative à UML est une description textuelle des objets et de leur interface, ou suivant le langage de programmation, le code lui-même [\[14\]](#).

UML fournit aussi une notation pour décrire le modèle dynamique du système. Cela est pratique dans les cas où les états de transition d'un système ou d'un sous-système sont suffisamment importants pour nécessiter leurs propres diagrammes (dans un système de contrôle par exemple). On peut aussi décrire les structures de données, pour les systèmes ou sous-systèmes dans lesquels les données sont le facteur dominant (comme une base de données).

On sait que la Phase 2 est terminée quand on dispose de la description des objets et de leur interface. Ou du moins de la majorité d'entre eux - il y en a toujours quelques-uns qu'on ne découvre qu'en Phase 3, mais cela ne fait rien. La préoccupation principale est de découvrir tous les objets. Il est plus agréable de les découvrir le plus tôt possible, mais la POO est assez souple pour pouvoir s'adapter si on en découvre de nouveaux par la suite. En fait, la conception d'un objet se fait en cinq étapes.

Les cinq étapes de la conception d'un objet

La conception d'un objet n'est pas limitée à la phase de codage du programme. En fait, la conception d'un objet passe par une suite d'étapes. Garder cela à l'esprit permet d'éviter de prétendre à la perfection immédiate. On réalise que la compréhension de ce que fait un objet et de ce à quoi il doit ressembler se fait progressivement. Ceci s'applique d'ailleurs aussi à la conception de nombreux types de programmes ; le modèle d'un type de programme n'émerge qu'après s'être confronté encore et encore au problème (se référer au livre *Thinking in Patterns with Java*, téléchargeable sur www.BruceEckel.com). Les objets aussi ne se révèlent à la compréhension qu'après un long processus.

1. Découverte de l'objet. Cette étape se situe durant l'analyse initiale du programme. Les objets peuvent être découverts en cherchant les facteurs extérieurs et les frontières, la duplication d'éléments dans le système, et les plus petites unités conceptuelles. Certains objets sont évidents si on dispose d'un ensemble de bibliothèques de classes. La ressemblance entre les classes peut suggérer des classes de base et l'héritage peut en être déduit immédiatement, ou plus tard dans la phase de conception.

2. Assemblage des objets. Lors de la construction d'un objet, on peut découvrir le besoin de nouveaux membres qui n'était pas apparu durant l'étape de découverte. Les besoins internes d'un objet peuvent requérir d'autres classes pour les supporter.

3. Construction du système. Une fois de plus, un objet peut révéler des besoins supplémentaires durant cette étape. Au fur et à mesure de l'avancement du projet, les objets évoluent. Les besoins de la communication et de l'interconnexion avec les autres objets du système peuvent changer les besoins des classes ou demander de nouvelles classes. Par exemple, on peut découvrir le besoin de classes d'utilitaires, telles que des listes chaînées, qui contiennent peu ou pas d'information et sont juste là pour aider les autres classes.

4. Extension du système. Si on ajoute de nouvelles fonctionnalités au système, on peut se rendre compte que sa conception ne facilite pas l'extension du système. Avec cette nouvelle information, on peut restructurer certaines parties du système, éventuellement en ajoutant de nouvelles classes ou de nouvelles hiérarchies de classes.

5. Réutilisation des objets. Ceci est le test final pour une classe. Si quelqu'un tente de réutiliser une classe dans une situation entièrement différente, il y découvrira certainement des imperfections. La modification de la classe pour s'adapter à de nouveaux programmes va en révéler les principes généraux, jusqu'à l'obtention d'un

type vraiment réutilisable. Cependant, il ne faut pas s'attendre à ce que tous les objets d'un système soient réutilisables - il est tout à fait légitime que la majorité des objets soient spécifiques au système. Les classes réutilisables sont moins fréquentes, et doivent traiter de problèmes plus génériques pour être réutilisables.

Indications quant au développement des objets

Ces étapes suggèrent quelques règles de base concernant le développement des classes :

1. Quand un problème spécifique génère une classe, la laisser grandir et mûrir durant la résolution d'autres problèmes.
2. Se rappeler que la conception du système consiste principalement à découvrir les classes dont on a besoin (et leurs interfaces). Si on dispose déjà de ces classes, le projet ne devrait pas être compliqué.
3. Ne pas vouloir tout savoir dès le début ; compléter ses connaissances au fur et à mesure de l'avancement du projet. La connaissance viendra de toutes façons tôt ou tard.
4. Commencer à programmer ; obtenir un prototype qui marche afin de pouvoir approuver la conception ou au contraire la dénoncer. Ne pas avoir peur de se retrouver avec du code-spaghetti à la procédurale - les classes partitionnent le problème et aident à contrôler l'anarchie. Les mauvaises classes n'affectent pas les classes bien conçues.
5. Toujours rester le plus simple possible. De petits objets propres avec une utilité apparente sont toujours mieux conçus que ceux disposant de grosses interfaces compliquées. Quand une décision doit être prise, utiliser l'approche d'Occam's Razor : choisir la solution la plus simple, car les classes simples sont presque toujours les meilleures. Commencer petit et simple, et étendre l'interface de la classe quand on la comprend mieux. Il est toujours plus difficile d'enlever des éléments d'une classe.

Phase 3 : Construire le coeur du système

Ceci est la conversion initiale de la conception brute en portion de code compilable et exécutable qui peut être testée, et surtout qui va permettre d'approuver ou d'invalidier l'architecture retenue. Ce n'est pas un processus qui se fait en une passe, mais plutôt le début d'une série d'étapes qui vont construire le système au fur et à mesure comme le montre la Phase 4.

Le but ici est de trouver le coeur de l'architecture du système qui a besoin d'être implémenté afin de générer un système fonctionnel, sans se soucier de l'état de complétion du système dans cette passe initiale. Il s'agit ici de créer un cadre sur lequel on va pouvoir s'appuyer pour les itérations suivantes. On réalise aussi la première des nombreuses intégrations et phases de tests, et on donne les premiers retours aux clients sur ce à quoi leur système ressemblera et son état d'avancement. Idéalement, on découvre quelques-uns des risques critiques. Des changements ou des améliorations sur l'architecture originelle seront probablement découverts - des choses qu'on n'aurait pas découvert avant l'implémentation du système.

Une partie de la construction du système consiste à confronter le système avec l'analyse des besoins et les spécifications du système (quelle que soit la forme sous laquelle ils existent). Il faut s'assurer en effet que les tests vérifient les besoins et les cas d'utilisations. Quand le coeur du système est stable, on peut passer à la suite et ajouter des fonctionnalités supplémentaires.

Phase 4 : Itérer sur les cas d'utilisation

Une fois que le cadre de base fonctionne, chaque fonctionnalité ajoutée est un petit projet en elle-même. On ajoute une fonctionnalité durant une *itération*, période relativement courte du développement.

Combien de temps dure une itération ? Idéalement, chaque itération dure entre une et trois semaines (ceci peut varier suivant le langage d'implémentation choisi). A la fin de cette période, on dispose d'un système intégré et testé avec plus de fonctionnalités que celles dont il disposait auparavant. Mais ce qu'il est intéressant de noter, c'est qu'un simple cas d'utilisation constitue la base d'une itération. Chaque cas d'utilisation est un ensemble de fonctionnalités qu'on ajoute au système toutes en même temps, durant une itération. Non seulement cela permet de se faire une meilleure idée de ce que recouvre ce cas d'utilisation, mais cela permet de le valider, puisqu'il n'est pas abandonné après l'analyse et la conception, mais sert au contraire tout au long du processus de création.

Les itérations s'arrêtent quand on dispose d'un système comportant toutes les fonctionnalités souhaitées ou qu'une date limite arrive et que le client se contente de la version courante (se rappeler que les commanditaires dirigent l'industrie du logiciel). Puisque le processus est itératif, on dispose de nombreuses opportunités pour délivrer une version intermédiaire plutôt qu'un produit final ; les projets open-source travaillent uniquement dans un environnement itératif avec de nombreux retours, ce qui précisément les rend si productifs.

Un processus de développement itératif est intéressant pour de nombreuses raisons. Cela permet de révéler et de résoudre des risques critiques très tôt, les clients ont de nombreuses opportunités pour changer d'avis, la satisfaction des programmeurs est plus élevée, et le projet peut être piloté avec plus de précision. Mais un bénéfice additionnel particulièrement important est le retour aux commanditaires du projet, qui peuvent voir grâce à l'état courant du produit où le projet en est. Ceci peut réduire ou éliminer le besoin de réunions soporifiques sur le projet, et améliore la confiance et le support des commanditaires.

Phase 5 : Evolution

Cette phase du cycle de développement a traditionnellement été appelée « maintenance », un terme fourre-tout qui peut tout vouloir dire depuis « faire marcher le produit comme il était supposé le faire dès le début » à « ajouter de nouvelles fonctionnalités que le client a oublié de mentionner » au plus traditionnel « corriger les bugs qui apparaissent » et « ajouter de nouvelles fonctionnalités quand le besoin s'en fait sentir ». Le terme « maintenance » a été la cause de si nombreux malentendus qu'il en est arrivé à prendre un sens péjoratif, en partie parce qu'il suggère qu'on a fourni un programme parfait et que tout ce qu'on a besoin de faire est d'en changer quelques parties, le graisser et l'empêcher de rouiller. Il existe peut-être un meilleur terme pour décrire ce qu'il en est réellement.

J'utiliserai plutôt le terme *évolution* [\[15\]](#). C'est à dire, « Tout ne sera pas parfait dès le premier jet, il faut se laisser la latitude d'apprendre et de revenir en arrière pour faire des modifications ». De nombreux changements seront peut-être nécessaires au fur et à mesure que l'appréhension et la compréhension du problème augmentent. Si on continue d'évoluer ainsi jusqu'au bout, l'élégance obtenue sera payante, à la fois à court et long terme. L'évolution permet de passer d'un bon à un excellent programme, et clarifie les points restés obscurs durant la première passe. C'est aussi dans cette phase que les classes passent d'un statut d'utilité limitée au système à ressource réutilisable.

Ici, « jusqu'au bout » ne veut pas simplement dire que le programme fonctionne suivant les exigences et les cas d'utilisation. Cela veut aussi dire que la structure interne du code présente une logique d'organisation et semble bien s'assembler, sans abus de syntaxe, d'objets surdimensionnés ou de code inutilement exposé. De plus, il faut s'assurer que la structure du programme puisse s'adapter aux changements qui vont inévitablement arriver pendant sa durée vie, et que ces changements puissent se faire aisément et proprement. Ceci n'est pas une petite caractéristique. Il faut comprendre non seulement ce qu'on construit, mais aussi comment le programme va évoluer (ce que j'appelle le *vecteur changement*). Heureusement, les langages de programmation orientés objet sont particulièrement adaptés à ce genre de modifications continues - les frontières créées par les objets sont

ce qui empêchent la structure du programme de s'effondrer. Ils permettent aussi de faire des changements - même ceux qui seraient considérés comme sévères dans un programme procédural - sans causer de ravages dans l'ensemble du code. En fait le support de l'évolution pourrait bien être le bénéfice le plus important de la programmation orientée objet.

Avec l'évolution, on crée quelque chose qui approche ce qu'on croit avoir construit, on le compare avec les exigences et on repère les endroits où cela coince. On peut alors revenir en arrière et corriger cela en remodelisant et réimplémentant les portions du programme qui ne fonctionnaient pas correctement [16]. De fait, on peut avoir besoin de résoudre le problème ou un de ses aspects un certain nombre de fois avant de trouver la bonne solution (une étude de *Design Patterns* s'avère généralement utile ici). On pourra trouver plus d'informations dans *Thinking in Patterns with Java*, téléchargeable à www.BruceEckel.com).

Il faut aussi évoluer quand on construit un système, qu'on voit qu'il remplit les exigences et qu'on découvre finalement que ce n'était pas ce qu'on voulait. Quand on se rend compte après avoir vu le système en action qu'on essayait de résoudre un autre problème. Si on pense que ce genre d'évolution est à prendre en considération, alors on se doit de construire une première version aussi rapidement que possible afin de déterminer au plus tôt si c'est réellement ce qu'on veut.

La chose la plus importante à retenir est que par défaut - par définition, plutôt - si on modifie une classe, ses classes parentes et dérivées continueront de fonctionner. Il ne faut pas craindre les modifications (surtout si on dispose d'un ensemble de tests qui permettent de vérifier les modifications apportées). Les modifications ne vont pas nécessairement casser le programme, et tout changement apporté sera limité aux sous-classes et / ou aux collaborateurs spécifiques de la classe qu'on change.

Les plans sont payants

Bien sûr on ne bâtirait pas une maison sans une multitude de plans dessinés avec attention. Si on construit un pont ou une niche, les plans ne seront pas aussi élaborés, mais on démarre avec quelques esquisses pour se guider. Le développement de logiciels a connu les extrêmes. Longtemps les gens ont travaillé sans structure, mais on a commencé à assister à l'effondrement de gros projets. En réaction, on en est arrivé à des méthodologies comprenant un luxe de structure et de détails, destinées justement à ces gros projets. Ces méthodologies étaient trop intimidantes pour qu'on les utilise - on avait l'impression de passer son temps à écrire des documents et aucun moment à coder (ce qui était souvent le cas). J'espère que ce que je vous ai montré ici suggère un juste milieu. Utilisez une approche qui corresponde à vos besoins (et votre personnalité). Même s'il est minimal, la présence d'un plan vous apportera beaucoup dans la gestion de votre projet. Rappelez-vous que selon la plupart des estimations, plus de 50 pour cent des projets échouent (certaines estimations vont jusqu'à 70 pour cent).

En suivant un plan - de préférence un qui soit simple et concis - et en produisant une modélisation de la structure avant de commencer à coder, vous découvrirez que les choses s'arrangent bien mieux que si on se lance comme ça dans l'écriture. Vous en retirerez aussi une plus grande satisfaction. Suivant mon expérience, arriver à une solution élégante procure une satisfaction à un niveau entièrement différent ; cela ressemble plus à de l'art qu'à de la technologie. Et l'élégance est toujours payante, ce n'est pas une vaine poursuite. Non seulement on obtient un programme plus facile à construire et débbuguer, mais qui est aussi plus facile à comprendre et maintenir, et c'est là que sa valeur financière réside.

Programmation Extrême

J'ai étudié à différentes reprises les techniques d'analyse et de conception depuis que je suis sorti de l'école. Le concept de *Programmation Extrême* (XP) est le plus radical et divertissant que j'ai vu. Il est rapporté dans *Extreme Programming Explained* de Kent Beck (Addison-Wesley, 2000) et sur le web à

www.xprogramming.com.

XP est à la fois une philosophie à propos de la programmation et un ensemble de règles de bases. Certaines de ces règles sont reprises dans d'autres méthodologies récentes, mais les deux contributions les plus importantes et novatrices, sont à mon sens « commencer par écrire les tests » et « programmation en binôme ». Bien qu'il soutienne et argumente l'ensemble de la théorie, Beck insiste sur le fait que l'adoption de ces deux pratiques améliore grandement la productivité et la fiabilité.

Commencer par écrire les tests

Les tests ont traditionnellement été relégués à la dernière partie d'un projet, une fois que « tout marche, mais c'est juste pour s'en assurer ». Ils ne sont généralement pas prioritaires et les gens qui se spécialisent dedans ne sont pas reconnus à leur juste valeur et se sont souvent vus cantonnés dans un sous-sol, loin des « véritables programmeurs ». Les équipes de test ont réagi en conséquence, allant jusqu'à porter des vêtements de deuil et glosser joyeusement quand ils trouvaient des erreurs (pour être honnête, j'ai eu moi aussi ce genre de sentiments lorsque je mettais des compilateurs en faute).

XP révolutionne complètement le concept du test en lui donnant une priorité aussi importante (ou même plus forte) que le code. En fait, les tests sont écrits avant le code qui sera testé, et les tests restent tout le temps avec le code. Ces tests doivent être exécutés avec succès à chaque nouvelle intégration dans le projet (ce qui peut arriver plus d'une fois par jour).

Ecrire les tests d'abord a deux effets extrêmement importants.

Premièrement, cela nécessite une définition claire de l'interface d'une classe. J'ai souvent suggéré que les gens « imaginent la classe parfaite qui résolve un problème particulier » comme outil pour concevoir le système. La stratégie de test de XP va plus loin - elle spécifie exactement ce à quoi la classe doit ressembler pour le client de cette classe, et comment la classe doit se comporter. Dans des termes non ambigus. On peut écrire tout ce qu'on veut, ou créer tous les diagrammes qu'on veut, décrivant comment une classe devrait se comporter et ce à quoi elle ressemble, mais rien n'est aussi réel qu'une batterie de tests. Le premier est une liste de vœux, mais les tests sont un contrat certifié par un compilateur et un programme qui marche. Il est dur d'imaginer une description plus concrète d'une classe que des tests.

En créant les tests, on est forcé de penser complètement la classe et souvent on découvre des fonctionnalités nécessaires qui ont pu être manquées lors de l'utilisation des diagrammes UML, des cartes CRC, des cas d'utilisation, etc...

Le deuxième effet important dans l'écriture des tests en premier vient du fait qu'on peut lancer les tests à chaque nouvelle version du logiciel. Cela permet d'obtenir l'autre moitié des tests réalisés par le compilateur. Si on regarde l'évolution des langages de programmation de ce point de vue, on se rend compte que les améliorations réelles dans la technologie ont en fait tourné autour du test. Les langages assembleur vérifiaient uniquement la syntaxe, puis le C a imposé des restrictions sémantiques, et cela permettait d'éviter certains types d'erreurs. Les langages orientés objet imposent encore plus de restrictions sémantiques, qui sont quand on y pense des formes de test. « Est-ce que ce type de données est utilisé correctement ? » et « Est-ce que cette fonction est appelée correctement ? » sont le genre de tests effectués par le compilateur ou le système d'exécution. On a pu voir le résultat d'avoir ces tests dans le langage même : les gens ont été capables de construire des systèmes plus complexes, et de les faire marcher, et ce en moins de temps et d'efforts. Je me suis souvent demandé pourquoi, mais maintenant je réalise que c'est grâce aux tests : si on fait quelque chose de faux, le filet de sécurité des tests intégré au langage prévient qu'il y a un problème et montre même où il réside.

Mais les tests intégrés permis par la conception du langage ne peuvent aller plus loin. A partir d'un certain point, il est de notre responsabilité de produire une suite complète de tests (en coopération avec le compilateur et le

système d'exécution) qui vérifie tout le programme. Et, de même qu'il est agréable d'avoir un compilateur qui vérifie ce qu'on code, ne serait-il pas préférable que ces tests soient présents depuis le début ? C'est pourquoi on les écrit en premier, et qu'on les exécute automatiquement à chaque nouvelle version du système. Les tests deviennent une extension du filet de sécurité fourni par le langage.

L'utilisation de langages de programmation de plus en plus puissants m'a permis de tenter plus de choses audacieuses, parce que je sais que le langage m'empêchera de perdre mon temps à chasser les bugs. La stratégie de tests de XP réalise la même chose pour l'ensemble du projet. Et parce qu'on sait que les tests vont révéler tous les problèmes introduits (et on ajoute de nouveaux tests au fur et à mesure qu'on les imagine), on peut faire de gros changements sans se soucier de mettre le projet complet en déroute. Ceci est une approche particulièrement puissante.

Programmation en binôme

La programmation en binôme va à l'encontre de l'individualisme farouche endoctriné, depuis l'école (où on réussit ou échoue suivant nos mérites personnels, et où travailler avec ses voisins est considéré comme « tricher »), et jusqu'aux médias, en particulier les films hollywoodiens dans lequel le héros se bat contre la conformité [\[17\]](#). Les programmeurs aussi sont considérés comme des parangons d'individualisme - des « codeurs cowboy » comme aime à le dire Larry Constantine. Et XP, qui se bat lui-même contre la pensée conventionnelle, soutient que le code devrait être écrit avec deux personnes par station de travail. Et cela devrait être fait dans un endroit regroupant plusieurs stations de travail, sans les barrières dont raffolent les gens des Moyens Généraux. En fait, Beck dit que la première tâche nécessaire pour implémenter XP est de venir avec des tournevis et d'enlever tout ce qui se trouve dans le passage [\[18\]](#) (cela nécessite un responsable qui puisse absorber la colère du département des Moyens Généraux).

Dans la programmation en binôme, une personne produit le code tandis que l'autre y réfléchit. Le penseur garde la conception générale à l'esprit - la description du problème, mais aussi les règles de XP à portée de main. Si deux personnes travaillent, il y a moins de chance que l'une d'entre elles s'en aille en disant « Je ne veux pas commencer en écrivant les tests », par exemple. Et si le codeur reste bloqué, ils peuvent changer leurs places. Si les deux restent bloqués, leurs songeries peuvent être remarquées par quelqu'un d'autre dans la zone de travail qui peut venir les aider. Travailler en binôme permet de garder une bonne productivité et de rester sur la bonne pente. Probablement plus important, cela rend la programmation beaucoup plus sociable et amusante.

J'ai commencé à utiliser la programmation en binôme durant les périodes d'exercice dans certains de mes séminaires et il semblerait que cela enrichisse l'expérience personnelle de chacun.

Les raisons du succès de Java

Java est si populaire actuellement car son but est de résoudre beaucoup de problèmes auxquels les programmeurs doivent faire face aujourd'hui. Le but de Java est d'améliorer la productivité. La productivité vient de différents horizons, mais le langage a été conçu pour aider un maximum tout en entravant le moins possible avec des règles arbitraires ou l'obligation d'utiliser un ensemble particulier de caractéristiques. Java a été conçu pour être pratique ; les décisions concernant la conception de Java furent prises afin que le programmeur en retire le maximum de bénéfices.

Les systèmes sont plus faciles à exprimer et comprendre

Les classes conçues pour résoudre le problème sont plus faciles à exprimer. La solution est mise en oeuvre dans le code avec des termes de l'espace problème (« Mets le fichier à la poubelle ») plutôt qu'en termes machine, l'espace solution (« Positionne le bit à 1 ce qui veut dire que le relais va se fermer »). On traite de concepts de plus haut niveau et une ligne de code est plus porteuse de sens.

L'autre aspect bénéfique de cette facilité d'expression se retrouve dans la maintenance, qui représente (s'il faut en croire les rapports) une grosse part du coût d'un programme. Si un programme est plus facile à comprendre, il est forcément plus facile à maintenir. Cela réduit aussi le coût de création et de maintenance de la documentation.

Puissance maximale grâce aux bibliothèques

La façon la plus rapide de créer un programme est d'utiliser du code déjà écrit : une bibliothèque. Un des buts fondamentaux de Java est de faciliter l'emploi des bibliothèques. Ce but est obtenu en convertissant ces bibliothèques en nouveaux types de données (classes), et utiliser une bibliothèque revient à ajouter de nouveaux types au langage. Comme le compilateur Java s'occupe de l'interfaçage avec la bibliothèque - garantissant une initialisation et un nettoyage propres, et s'assurant que les fonctions sont appelées correctement - on peut se concentrer sur ce qu'on attend de la bibliothèque, et non sur les moyens de le faire.

Traitement des erreurs

L'une des difficultés du C est la gestion des erreurs, problème connu et largement ignoré - le croisement de doigts est souvent impliqué. Si on construit un programme gros et complexe, il n'y a rien de pire que de trouver une erreur enfouie quelque part sans qu'on sache d'où elle vienne. Le *traitement des exceptions* de Java est une façon de garantir qu'une erreur a été remarquée, et que quelque chose est mis en oeuvre pour la traiter.

Mise en oeuvre de gros projets

Beaucoup de langages traditionnels imposent des limitations internes sur la taille des programmes et leur complexité. BASIC, par exemple, peut s'avérer intéressant pour mettre en oeuvre rapidement des solutions pour certains types de problèmes ; mais si le programme dépasse quelques pages de long, ou s'aventure en dehors du domaine du langage, cela revient à tenter de nager dans un liquide encore plus visqueux. Aucune limite ne prévient que le cadre du langage est dépassé, et même s'il en existait, elle serait probablement ignorée. On devrait se dire : « Mon programme BASIC devient trop gros, je vais le réécrire en C ! », mais à la place on tente de glisser quelques lignes supplémentaires pour implémenter cette nouvelle fonctionnalité. Le coût total continue donc à augmenter.

Java a été conçu pour pouvoir mettre en oeuvre toutes les tailles de projets - c'est à dire, pour supprimer ces frontières liées à la complexité croissante d'un gros projet. L'utilisation de la programmation orientée objet n'est certainement pas nécessaire pour écrire un programme utilitaire du style « Hello world ! », mais les fonctionnalités sont présentes quand on en a besoin. Et le compilateur ne relâche pas son attention pour dénicher les bugs - produisant indifféremment des erreurs pour les petits comme pour les gros programmes.

Stratégies de transition

Si on décide d'investir dans la POO, la question suivante est généralement : « Comment convaincre mes directeur / collègues / département / collaborateurs d'utiliser les objets ? ». Il faut se mettre à leur place et réfléchir comment on ferait s'il fallait apprendre un nouveau langage et un nouveau paradigme de programmation. Ce cas s'est sûrement déjà présenté de par le passé. Il faut commencer par des cours et des exemples ; puis lancer un petit projet d'essai pour inculquer les bases sans les perdre dans les détails. Ensuite passer à un projet « réel » qui réalise quelque chose d'utile. Au cours du premier projet, continuer l'apprentissage par la lecture, poser des questions à des experts et échanger des astuces avec des amis. Cette approche est celle recommandée par de nombreux programmeurs expérimentés pour passer à Java. Si toute l'entreprise décide de passer à Java, cela entraînera bien sûr une dynamique de groupe, mais cela aide de se rappeler à chaque étape

comment une personne seule opérerait.

Règles de base

Voici quelques indications à prendre en compte durant la transition vers la programmation orientée objet et Java.

1. Cours

La première étape comprend une formation. Il ne faut pas jeter la confusion pendant six ou neuf mois dans l'entreprise, le temps que tout le monde comprenne comment fonctionnent les interfaces. Il vaut mieux se contenter d'un petit groupe d'endocinement, composé de préférence de personnes curieuses, s'entendant bien entre elles et suffisamment indépendantes pour créer leur propre réseau de support concernant l'apprentissage de Java.

Une approche alternative suggérée quelquefois comprend la formation de tous les niveaux de l'entreprise à la fois, en incluant à la fois des cours de présentation pour les responsables et des cours de conception et de programmation pour les développeurs. Ceci est particulièrement valable pour les petites entreprises qui n'hésitent pas à introduire des changements radicaux dans leurs manières de faire, ou au niveau division des grosses entreprises. Cependant, comme le coût résultant est plus important, certains choisiront de commencer avec des cours, de réaliser un projet pilote (si possible avec un mentor extérieur), et de laisser l'équipe du projet devenir les professeurs pour le reste de l'entreprise.

2. Projet à faible risque

Tester sur un projet à faible risque qui autorise les erreurs. Une fois gagnée une certaine expérience, on peut soit répartir les membres de cette première équipe sur les autres projets ou utiliser cette équipe comme support technique de la POO. Le premier projet peut ne pas fonctionner correctement dès le début, il ne faut donc pas qu'il soit critique pour l'entreprise. Il doit être simple, indépendant et instructif : cela veut dire qu'il doit impliquer la création de classes qui soient compréhensibles aux autres programmeurs quand arrivera leur tour d'apprendre Java.

3. S'inspirer de bonnes conceptions

Rechercher des exemples de bonnes conceptions orientées objet avant de partir de zéro. Il y a de fortes chances que quelqu'un ait déjà résolu le problème, et s'ils ne l'ont pas résolu exactement modifier le système existant (grâce à l'abstraction, cf plus haut) pour qu'il remplisse nos besoins. C'est le concept général des *patrons génériques*, couverts dans *Thinking in Patterns with Java*, téléchargeable à www.BruceEckel.com.

4. Utiliser les bibliothèques de classes existantes

La motivation économique première pour passer à la POO est la facilité de réutilisation de code existant sous la forme de bibliothèques de classes (en particulier, les Bibliothèques Standard Java, couvertes tout au long de ce livre). On arrive au cycle de développement le plus court quand on crée et utilise des objets directement tirés des bibliothèques. Cependant, de nouveaux programmeurs ne saisissent pas ce point, ne connaissent pas l'existence de telles bibliothèques ou, fascinés par le langage, veulent réécrire des classes qui existent déjà. Le succès de la transition vers Java et la POO passe par des efforts pour encourager les gens à réutiliser le plus possible le code des autres.

5. Ne pas traduire du code existant en Java

Ce n'est généralement pas le meilleur usage de son temps qu'on puisse trouver que de prendre du code existant et fonctionnel et de le traduire en Java (si on doit le transformer en objets, on peut s'interfacer avec du code C ou C++ en utilisant l'Interface Native Java - JNI - décrite dans l'annexe B). Il y a bien sûr des avantages à le faire, particulièrement si ce code est destiné à être réutilisé. Mais il y a de fortes chances pour qu'on passe à côté de la spectaculaire amélioration de productivité espérée sauf si ce premier projet en est un nouveau. Java et la POO sont mis en valeur quand on suit un projet de la conception à la mise en oeuvre.

Les obstacles au niveau du management

Si on se situe du côté des responsables, le travail consiste à acquérir des ressources pour l'équipe, surmonter les obstacles pouvant gêner l'équipe, et en général tenter de fournir l'environnement le plus productif et agréable afin que l'équipe puisse accomplir ces miracles qu'on vous demande toujours de réaliser. Le passage à Java entre dans ces trois catégories, et ce serait merveilleux si de plus cela ne coûtait rien. Bien que le passage à Java puisse être moins onéreux - suivant les contraintes - que d'autres alternatives de POO pour une équipe de programmeurs C (et probablement pour les développeurs dans d'autres langages), ce coût n'est pas nul, et il y a des obstacles dont il faut être conscient avant de promouvoir le passage à Java à l'intérieur de l'entreprise et de s'embarquer dans la transition.

Coûts de mise en oeuvre

Le coût du passage à Java recouvre plus que l'acquisition d'un compilateur Java (le compilateur Java fourni par Sun est gratuit, cela n'est donc pas un obstacle). Les coûts à moyen et long terme seront minimisés si on investit dans la formation (et éventuellement dans la participation d'un consultant pour le premier projet), de même que si on identifie et achète des bibliothèques de classes qui résolvent les problèmes plutôt que de tenter de réécrire ces bibliothèques soi-même. Ce sont des investissements lourds qui doivent être relativisés dans une proposition raisonnable. De plus, il existe des coûts cachés dans la baisse de productivité liée à l'apprentissage d'un nouveau langage et éventuellement d'un nouvel environnement de développement. La formation et l'encadrement peuvent certainement les réduire, mais les membres de l'équipe devront mener leur propre combat pour maîtriser la nouvelle technologie. Durant cette phase ils feront plus d'erreurs (c'est prouvé, et les erreurs comprises constituent le meilleur moyen de progresser) et seront moins productifs. Cependant, avec certains types de problèmes, les bonnes classes et le bon environnement de développement, il est possible d'être plus productif pendant l'apprentissage de Java (même en considérant qu'on fait plus d'erreurs et qu'on écrit moins de lignes de code par jour) que si on en était resté au C.

Problèmes de performances

Une question qui revient souvent est : « Est-ce que la POO ne rend pas obligatoirement mes programmes plus gros et plus lents ? ». La réponse est « Ca dépend. ». Les fonctionnalités de vérification introduites dans Java ont prélevé leur tribut sur la performance comparé à un langage comme le C++. Des technologies telles que le « hotspot » et les techniques de compilation ont significativement amélioré la vitesse dans la plupart des cas, et les efforts pour des performances accrues continuent.

Quand on se concentre sur le prototypage rapide, on assemble des composants aussi vite que possible en ignorant les problèmes liés à l'efficacité. Si on utilise des bibliothèques extérieures, celles-ci ont généralement été optimisées par leurs distributeurs ; de toutes façons ce n'est pas la préoccupation première quand on est en phase de développement rapide. Quand on dispose d'un système qui nous satisfait et s'il est suffisamment petit et rapide, alors le tour est joué. Sinon, on tente de mettre au point avec un outil de profilage, en cherchant en premier des améliorations qui peuvent être faites en réécrivant de petites portions de code. Si cela ne suffit pas, il faut chercher si on peut apporter quelques changements à l'implémentation sous-jacente afin qu'aucune classe particulière n'ait besoin de modifier son code. Il ne faut toucher à la conception que si rien d'autre ne résout le problème. Le fait que les performances soient si critiques dans cette phase de la conception est un indicateur que ce doit être un des critères essentiels de la conception. On a le bénéfice de s'en rendre compte relativement tôt

en utilisant le prototypage rapide.

Si on trouve une fonction qui soit un goulot d'étranglement, on peut la réécrire en C ou en C++ en utilisant les *méthodes natives* de Java, sujet de l'annexe B.

Erreurs classiques de conception

Quand l'équipe commencera la programmation orientée objet et Java, les programmeurs vont typiquement passer par une série d'erreurs classiques de conception. Ceci est souvent dû à des retours insuffisants de la part d'experts durant la conception et l'implémentation des premiers projets, puisqu'aucun expert n'existe encore au sein de l'entreprise et qu'il peut y avoir des réticences à engager des consultants. Il est facile de se dire trop tôt qu'on maîtrise la POO et partir sur de mauvaises bases. Quelque chose d'évident pour une personne expérimentée peut être le sujet d'un débat interne intense pour un novice. La plupart de ces difficultés peuvent être évitées en utilisant un expert extérieur pour la formation.

Java vs. C++ ?

Java ressemble beaucoup au C++, et il semblerait naturel que le C++ soit remplacé par Java. Mais je commence à m'interroger sur cette logique. D'une part, C++ dispose toujours de fonctionnalités que Java n'a pas, et bien que de nombreuses promesses aient été faites sur le fait que Java soit un jour aussi rapide, voire même plus, que le C++, on a vu de grosses améliorations mais pas de révolutions spectaculaires. De plus, il semblerait que le C++ intéresse une large communauté passionnée, et je ne pense donc pas que ce langage puisse disparaître prochainement (les langages semblent résister au cours du temps. Allen Hollub a affirmé durant l'un de mes « Séminaires Java Intermédiaire/Avancé » que les deux langages les plus utilisés étaient Rexx et COBOL, dans cet ordre).

Je commence à croire que la force de Java réside dans une optique différente de celle du C++. C++ est un langage qui n'essaie pas de se fondre dans un moule. Il a déjà été adapté un certain nombre de fois pour résoudre des problèmes particuliers. Certains des outils du C++ combinent des bibliothèques, des modèles de composants et des outils de génération de code pour résoudre les problèmes concernant le développement d'applications fenêtrées (pour Microsoft Windows). Et pourtant, la vaste majorité des développeurs Windows utilisent Microsoft Visual Basic (VB). Et ceci malgré le fait que VB produise le genre de code qui devient ingérable quand le programme fait plus de quelques pages de long (sans compter que la syntaxe peut être profondément mystérieuse). Aussi populaire que soit VB, ce n'est pas un très bon exemple de conception de langage. Il serait agréable de pouvoir disposer des facilités et de la puissance fournies par VB sans se retrouver avec ce code ingérable. Et c'est là où je pense que Java va pouvoir briller : comme le « VB du futur ». On peut frissonner en entendant ceci, mais Java est conçu pour aider le développeur à résoudre des problèmes comme les applications réseaux ou interfaces utilisateurs multiplateformes, et la conception du langage permet la création de portions de code très importantes mais néanmoins flexibles. Ajoutons à ceci le fait que Java dispose des systèmes de vérifications de types et de gestion des erreurs les plus robustes que j'ai jamais rencontré dans un langage et on se retrouve avec les éléments constitutifs d'un bond significatif dans l'amélioration de la productivité dans la programmation.

Faut-il utiliser Java en lieu et place du C++ dans les projets ? En dehors des applets Web, il y a deux points à considérer. Premièrement, si on veut réutiliser un certain nombre de bibliothèques C++ (et on y gagnera certainement en productivité), ou si on dispose d'une base existante en C ou C++, Java peut ralentir le développement plutôt que l'accélérer.

Si on développe tout le code en partant de zéro, alors la simplicité de Java comparée au C++ réduira significativement le temps de développement - des anecdotes (selon des équipes C++ à qui j'ai parlé après qu'ils eurent changé pour Java) suggèrent un doublement de la vitesse de développement comparé au C++. Si les

performances moindres de Java ne rentrent pas en ligne de compte ou qu'on peut les compenser, les contraintes de temps font qu'il est difficile de choisir le C++ aux dépens de Java.

Le point le plus important est la performance. Java interprété est lent, environ 20 à 50 fois plus lent que le C dans les interpréteurs Java originaux. Ceci a été grandement amélioré avec le temps, mais il restera toujours un important facteur de différence. Les ordinateurs existent de par leur rapidité ; si ce n'était pas considérablement plus rapide de réaliser une tâche sur ordinateur, on la ferait à la main. J'ai même entendu suggérer de démarrer avec Java, pour gagner sur le temps de développement plus court, et ensuite utiliser un outil et des bibliothèques de support pour traduire le code en C++ si on a un besoin de vitesse d'exécution plus rapide.

La clef pour rendre Java viable dans la plupart des projets consiste en des améliorations de vitesse d'exécution, grâce à des compilateurs « juste à temps » (« just in time », JIT), la technologie « hotspot » de Sun, et même des compilateurs de code natif. Bien sûr, les compilateurs de code natif éliminent les possibilités d'exécution interplateformes du programme compilé, mais la vitesse des exécutables produits se rapprochera de celle du C et du C++. Et réaliser un programme multiplateformes en Java devrait être beaucoup plus facile qu'en C ou C++ (en théorie, il suffit de recompiler, mais cette promesse a déjà été faite pour les autres langages).

Vous trouverez des comparaisons entre Java et C++ et des observations sur Java dans les annexes de la première édition de ce livre (disponible sur le CD ROM accompagnant ce livre, et à www.BruceEckel.com).

Résumé

Ce chapitre tente de vous donner un aperçu des sujets couverts par la programmation orientée objet et Java (les raisons qui font que la POO est particulière, de même que Java), les concepts des méthodologies de la POO, et finalement le genre de problèmes que vous rencontrerez quand vous migrerez dans votre entreprise à la programmation orientée objet et Java.

La POO et Java ne sont pas forcément destinés à tout le monde. Il est important d'évaluer ses besoins et décider si Java satisfera au mieux ces besoins, ou si un autre système de programmation ne conviendrait pas mieux (celui qu'on utilise actuellement y compris). Si on connaît ses besoins futurs et qu'ils impliquent des contraintes spécifiques non satisfaites par Java, alors on se doit d'étudier les alternatives existantes [19]. Et même si finalement Java est retenu, on saura au moins quelles étaient les options et les raisons de ce choix.

On sait à quoi ressemble un programme procédural : des définitions de données et des appels de fonctions. Pour trouver le sens d'un tel programme il faut se plonger dans la chaîne des appels de fonctions et des concepts de bas niveau pour se représenter le modèle du programme. C'est la raison pour laquelle on a besoin de représentations intermédiaires quand on conçoit des programmes procéduraux - par nature, ces programmes tendent à être confus car le code utilise des termes plus orientés vers la machine que vers le problème qu'on tente de résoudre.

Parce que Java introduit de nombreux nouveaux concepts par rapport à ceux qu'on trouve dans un langage procédural, on pourrait se dire que la fonction **main()** dans un programme Java sera bien plus compliquée que son équivalent dans un programme C. On sera agréablement surpris de constater qu'un programme Java bien écrit est généralement beaucoup plus simple et facile à comprendre que son équivalent en C. On n'y voit que les définitions des objets qui représentent les concepts de l'espace problème (plutôt que leur représentation dans l'espace machine) et les messages envoyés à ces objets pour représenter les activités dans cet espace. L'un des avantages de la POO est qu'avec un programme bien conçu, il est facile de comprendre le code en le lisant. De plus, il y a généralement moins de code, car beaucoup de problèmes sont résolus en réutilisant du code existant dans des bibliothèques.

[2] Voir *Multiparadigm Programming in Leda* de Timothy Budd (Addison-Wesley 1995).

[3] Certaines personnes établissent une distinction, en disant qu'un type détermine une interface tandis qu'une classe est une implémentation particulière de cette interface.

[4] Je suis reconnaissant envers mon ami Scott Meyers pour cette expression.

[5] Cela suffit généralement pour la plupart des diagrammes, et on n'a pas besoin de préciser si on utilise un agrégat ou une composition.

[6] Invention personnelle.

[7] Les types primitifs, que vous rencontrerez plus loin, sont un cas spécial.

[8] Un très bon exemple en est *UML Distilled*, 2nd édition, de Martin Fowler (Addison-Wesley 2000), qui réduit la méthode UML - parfois écrasante - à un sous-ensemble facilement gérable.

[9] Ma règle pour estimer de tels projets : s'il y a plus d'un facteur joker, n'essayez même pas de planifier combien de temps cela va prendre ou d'estimer le coût avant d'avoir créé un prototype fonctionnel. Il y a trop de degrés de liberté.

[10] Merci à James H Jarrett pour son aide.

[11] D'autres informations sur les cas d'utilisation peuvent être trouvées dans *Applying Use Cases* de Schneider & Winters (Addison-Wesley 1998) et *Use Case Driven Object Modeling with UML* de Rosenberg (Addison-Wesley 1999).

[12] Mon avis personnel sur tout cela a changé récemment. Doubler et ajouter 10 pour cent donnera une estimation raisonnablement précise (en assumant qu'il n'y ait pas trop de facteurs joker), mais il faudra tout de même ne pas traîner en cours de route pour respecter ce délai. Si on veut réellement du temps pour rendre le système élégant et ne pas être continuellement sous pression, le multiplicateur correct serait plus trois ou quatre fois le temps prévu, je pense..

[13] Pour les débutants, je recommande *UML Distilled*, 2nd édition, déjà mentionné plus haut.

[14] Python (www.python.org) est souvent utilisé en tant que « pseudocode exécutable ».

[15] Au moins un aspect de l'évolution est couvert dans le livre de Martin Fowler *Refactoring : improving the design of existing code* (Addison-Wesley 1999), qui utilise exclusivement des exemples Java.

[16] Cela peut ressembler au « prototypage rapide » où on est supposé fournir une version rapide-et-sale afin de pouvoir appréhender correctement le problème, puis jeter ce prototype et construire une version finale acceptable. L'inconvénient du prototypage rapide est que les gens ne jettent pas le prototype, mais s'en servent de base pour le développement. Combiné au manque de structure de la programmation procédurale, cela mène à des systèmes compliqués et embrouillés, difficiles et onéreux à maintenir.

[17] Bien que ceci soit plus une perception américaine, les productions hollywoodiennes touchent tout le monde.

[18] En particulier le système d'annonces sonores. J'ai travaillé une fois dans une entreprise qui insistait pour diffuser tous les appels téléphoniques à tous les responsables, et cela interrompait constamment notre productivité (mais les responsables n'imaginaient même pas qu'il soit concevable de vouloir couper un service aussi important que le système d'annonces sonores). Finalement, j'ai coupé les fils des haut-parleurs quand personne ne regardait.

[19] En particulier, je recommande de regarder dans la direction de Python (www.Python.org).