Conception et programmation orientées objet

Bertrand Meyer

Traduit de l'anglais par Pierre Jouvelot

 ISBN : 978-2-212-12270-1

EYROLLES

2

Critères d'orientation objet

Nous avons exploré, dans le chapitre précédent, les objectifs de la méthode orientée objet. En préparation des parties B et C, dans lesquelles nous découvrirons les détails techniques de cette méthode, il est utile d'effectuer un survol rapide, mais néanmoins exhaustif, des aspects clés du développement orienté objet. Telle est l'ambition de ce chapitre.

Une conséquence importante sera d'obtenir un descriptif concis des caractéristiques d'un système dit orienté objet. Cette expression est devenue tellement galvaudée que nous avons besoin d'une liste précise de propriétés nous permettant d'évaluer chaque méthode, langage ou outil qui se prétend orienté objet.

Ce chapitre utilise un minimum d'explications et, donc, s'il s'agit de votre première lecture, ne vous attendez pas à comprendre en détail tous les critères évoqués ici ; les expliquer est le but du reste du livre. Considérez cette discussion comme une bande-annonce — pas le film complet, seulement un aperçu.

Attention : *GÂCHIS* !

Ceci mérite d'ailleurs un avertissement car, contrairement à une bande-annonce bien faite, ce chapitre correspond à ce que des fanas de cinéma considéreraient comme du gâchis — il fournit trop d'indications, et trop tôt. Il perturbe ainsi la progression pas à pas du livre, en particulier sa partie B, qui motive la technologie objet en procédant patiemment, cas par cas, avant de déduire et de justifier les solutions proposées. Lisez ce chapitre si vous aimez faire un rapide survol d'un sujet avant de plonger plus en profondeur. Mais si vous préférez conserver le plaisir de voir les problèmes s'évanouir petit à petit et découvrir les solutions une à une, n'hésitez pas à sauter ce chapitre. Vous n'aurez pas besoin de l'avoir lu pour comprendre les chapitres suivants.

2.1 À PROPOS DES CRITÈRES

Commençons par examiner les critères possibles d'évaluation du caractère objet.

Jusqu'où faut-il être dogmatique ?

La liste ci-dessous énonce toutes les caractéristiques que je considère comme essentielles à la production de logiciels de qualité basés sur la méthode orientée objet. Elle est ambitieuse et peut sembler sans compromission, voire dogmatique. Que faut-il en déduire dans le cas d'un

environnement qui satisferait certaines de ces conditions, mais pas toutes ? Devrait-on considérer qu'un tel environnement, à moitié OO, n'aurait aucun mérite ?

Vous seul, lecteur, êtes à même de répondre à cette question en fonction de votre situation propre. Plusieurs raisons peuvent justifier un certain compromis :

- Être "orienté objet" n'est pas une condition booléenne : l'environnement A, bien que n'étant pas 100 % OO, peut être "plus" OO qu'un environnement B ; si des contraintes externes limitent votre choix à A ou B, vous choisirez A, car il s'approche plus d'un environnement pleinement orienté objet.
- Il est rare que l'on ait besoin de toutes les propriétés OO en même temps.
- L'orientation objet peut n'être, dans le choix d'une solution logicielle, qu'un facteur parmi d'autres, et vous devez donc arbitrer entre les critères présentés ici et ces autres considérations.

Ceci ne change rien à ce qui est évident : pour faire un choix pertinent, même en présence de contraintes pratiques imposant des solutions imparfaites, vous devez connaître l'ensemble du contexte, tel qu'il est fourni ci-dessous.

Catégories

L'ensemble des critères a été divisé en trois parties :

- *Méthode et langage* : ces deux aspects presque indifférentiables concernent les méthodes de raisonnement et les notations utilisées pour analyser et produire du logiciel. Notez bien que (particulièrement dans la technologie objet) le terme "langage" fait référence non seulement au langage de programmation proprement dit, mais également aux notations, graphiques ou textuelles, qui sont utilisées lors de l'analyse et de la conception.
- *Implémentation et environnement* : les critères de cette catégorie décrivent les propriétés de base des outils qui permettent aux développeurs d'appliquer les idées orientées objet.
- *Bibliothèques* : la technologie objet est fondée sur la réutilisation de composants logiciels. Les critères de cette catégorie concernent la disponibilité de telles bibliothèques de base, ainsi que les mécanismes requis pour utiliser ces bibliothèques et en créer de nouvelles.

Cette division est pratique mais pas absolue, car certains critères concernent deux ou trois de ces catégories. Par exemple, le critère intitulé "gestion de la mémoire" a été classé dans le cadre méthode et langage, car un langage peut permettre ou interdire le ramasse-miettes automatique, mais il appartient aussi à la catégorie implémentation et environnement ; de même, le critère "assertion" fait aussi référence à des outils permettant de traiter ce concept.

2.2 MÉTHODE ET LANGAGE

Le premier ensemble de critères concerne la méthode et la notation qui l'accompagne.

Intégration

L'approche orientée objet est ambitieuse : elle prend en compte le cycle de vie complet du logiciel. Quand vous examinez diverses solutions orientées objet, vous devriez vérifier que la méthode et le langage, ainsi que les outils correspondants, s'appliquent aussi bien à l'analyse et à la conception qu'à l'implémentation et à la maintenance. Le langage, en particulier, devrait être un vecteur de pensée vous aidant dans toutes les étapes de votre travail.

Il en résulte un processus de développement intégré dans lequel la cohérence des concepts et des notations permet de minimiser les transitions entre étapes successives du cycle de vie.

Ces exigences permettent d'éviter deux situations qu'on rencontre encore fréquemment, bien qu'elles soient aussi inadéquates l'une que l'autre,

- L'utilisation des concepts orientés objet durant les seules phases d'analyse et de conception, la méthode et la notation ne pouvant être utilisées pour écrire du logiciel exécutable.
- L'utilisation d'un langage de programmation orienté objet qui n'est pas adapté à l'analyse ou à la conception.

En résumé :

> Un langage et un environnement orientés objet, couplés à une méthode adaptée, devraient être applicables au cycle de vie complet, de façon à réduire les transitions entre activités.

Classes

La méthode orientée objet est fondée sur la notion de classe. Informellement, une classe est un élément logiciel qui décrit un type abstrait de données et son implémentation totale ou partielle. Un type abstrait de données est un ensemble d'objets définis par la liste des opérations, ou *caractéristiques*, qui s'appliquent à ces objets, ainsi que les propriétés de ces opérations.

> La méthode et le langage devraient utiliser la notion de classe comme concept central.

Assertions

Les caractéristiques d'un type abstrait de données ont des propriétés spécifiées formellement qui devraient être visibles dans les classes auxquelles elles correspondent. Les assertions — préconditions de routine, postconditions de routine et invariants de classe — jouent ce rôle. Elles décrivent l'effet des caractéristiques sur les objets, indépendamment de l'implémentation de ces caractéristiques.

Les assertions ont trois applications majeures : elles facilitent la production de logiciel fiable ; elles fournissent une documentation systématique ; et elles sont un outil clé lors du test et du débogage de logiciel orienté objet.

> Le langage devrait permettre de décorer une classe et ses caractéristiques avec des assertions (préconditions, postconditions et invariants), afin de pouvoir utiliser des outils pour produire la documentation à partir de ces assertions et, éventuellement, de contrôler celles-ci à l'exécution.

Les groupes, étudiés dans le chapitre 28, jouent le rôle des pays et des provinces.

Dans la société des modules logiciels, où les classes jouent le rôle des villes et les instructions (le code effectivement exécutable) celui de l'exécutif du gouvernement, les assertions correspondent à la branche législative. Nous verrons plus loin qui prend en charge le système judiciaire.

Les classes vues comme des modules

L'orientation objet est avant tout une technique architecturale : elle influence principalement la structure modulaire des systèmes logiciels.

Le rôle clé est ici rempli à nouveau par les classes. Une classe décrit non seulement un type d'objets mais aussi une unité modulaire. Dans une approche orientée objet pure :

> Les classes devraient être les seuls modules.

En particulier, il n'y a pas de raison d'introduire une notion de programme principal, et les sous-programmes n'existent pas en tant qu'unités modulaires indépendantes. (Ils ne peuvent apparaître qu'à l'intérieur des classes.) On n'a nul besoin des "paquetages" présents dans les langages comme Ada, quoiqu'il puisse être pratique, pour des questions de gestion, de regrouper plusieurs classes dans des unités administratives appelées *groupes*.

Les classes vues comme des types

La notion de classe est suffisamment puissante pour éliminer le recours à un autre mécanisme de typage :

> Chaque type devrait être basé sur une classe.

Même les types de base comme `INTEGER` et `REAL` peuvent être dérivés à partir de classes ; normalement, de telles classes seront prédéfinies plutôt que redéfinies à chaque fois par chaque développeur.

Le calcul à base de caractéristiques

Dans le calcul orienté objet, il n'y a qu'un mécanisme fondamental de calcul : étant donné un objet qui (du fait de la règle précédente) est toujours une instance d'une certaine classe, appeler une caractéristique de cette classe sur cet objet. Par exemple, pour afficher une fenêtre donnée

sur un écran, vous appelez la caractéristique *display* sur un objet qui représente la fenêtre — une instance de la classe *WINDOW*. Les caractéristiques peuvent aussi avoir des arguments : pour accroître le salaire d'un employé `e` de `n` dollars, à partir de la date `d`, vous appelez la caractéristique *raise* sur `e`, avec `n` et `d` comme arguments.

De la même manière que nous traitons les types de base comme des classes prédéfinies, nous pouvons voir les opérations de base (comme l'addition de deux nombres) comme des cas spéciaux, prédéfinis, d'un appel de caractéristique, celui-ci étant un mécanisme très général de description des calculs.

L'appel de caractéristique devrait être le mécanisme essentiel de calcul.

Une classe qui contient un appel à une caractéristique d'une classe `C` est dite **client** de `C`. L'appel de caractéristique est aussi connu sous le vocable d'**envoi de message** ; dans cette terminologie, un appel comme celui décrit ci-dessus sera considéré comme envoyant à `e` le message "augmenter votre paie", avec les arguments `d` et `n`.

Rétention d'information

Lors de l'écriture d'une classe, il vous faudra parfois introduire une caractéristique dont la classe n'a besoin que de manière interne : une caractéristique qui fait partie de l'implémentation de la classe, mais pas de son interface. D'autres caractéristiques de la classe — éventuellement utilisables par les clients — peuvent appeler cette caractéristique pour leurs besoins propres ; mais il ne devrait pas être possible à un client de l'appeler directement.

Le mécanisme qui empêche certaines caractéristiques d'être appelées par des clients est appelé la rétention d'information. Comme cela sera expliqué dans un prochain chapitre, celui-ci est essentiel à l'évolution en douceur des systèmes logiciels.

En pratique, une bonne rétention d'information ne s'obtient pas en ne fournissant que des caractéristiques exportées (accessibles aux clients) et des caractéristiques secrètes (inaccessibles) ; les concepteurs de classe doivent aussi avoir la possibilité d'exporter une caractéristique de manière sélective à un ensemble déterminé de clients.

Il devrait être possible à l'auteur d'une classe de spécifier qu'une caractéristique est accessible à tous les clients, à aucun client ou à des clients précis.

Une conséquence immédiate de cette règle est que la communication entre classes devrait être strictement limitée. En particulier, un bon langage orienté objet ne devrait pas offrir de notion de variable globale ; les classes échangeront de l'information exclusivement via des appels de caractéristiques, et via le mécanisme d'héritage.

Traitement des exceptions

Des événements anormaux peuvent se produire lors de l'exécution d'un système logiciel. Dans le calcul orienté objet, ils correspondent souvent à des appels qui ne peuvent pas être exécutés

normalement, à cause d'un mauvais fonctionnement matériel, d'une impossibilité inattendue (comme un débordement numérique lors d'une addition) ou d'une bogue dans le logiciel.

Pour produire un logiciel fiable, il faut pouvoir se récupérer dans de telles situations. C'est le but du mécanisme d'exception.

> Le langage devrait fournir un mécanisme permettant de récupérer les situations anormales non prévues.

Dans la société des systèmes logiciels, comme vous avez pu vous en douter, le mécanisme d'exception correspond à la troisième branche du gouvernement, le système judiciaire (et la police, qui en est la main).

Typage statique

Quand l'exécution d'un système logiciel entraîne l'appel d'une caractéristique donnée sur un objet donné, comment pouvons-nous savoir que cet objet sera capable de gérer cet appel ? (En termes de messages : comment savons-nous que l'objet peut traiter le message ?)

Pour garantir une exécution correcte, le langage doit être typé. Ceci veut dire qu'il doit imposer un certain nombre de règles de compatibilité ; en particulier :

- Chaque entité (c'est-à-dire chaque nom utilisé dans le texte du logiciel pour se référer à des objets à l'exécution) est explicitement déclarée comme étant d'un certain type, dérivé d'une classe.
- Chaque appel de caractéristique sur une certaine entité utilise une caractéristique de la classe correspondante (et la caractéristique est accessible, au sens de la rétention d'information, à la classe de l'appelant).
- Les affectations et les passages d'arguments sont soumis à des **règles de conformité**, basées sur l'héritage, qui imposent que le type de la source soit compatible avec le type de la cible.

Dans un langage qui impose une telle politique, il est possible d'écrire un **vérificateur statique de type** qui acceptera ou rejettera les systèmes logiciels, garantissant que les systèmes ne causeront aucune erreur du type "caractéristique de l'objet non accessible" au moment de l'exécution.

> Un système de type bien défini devrait, en imposant un certain nombre de déclarations de type et de règles de conformité, garantir la sûreté à l'exécution des systèmes qu'il accepte.

Généricité

Pour que le typage soit utilisable, il doit être possible de définir des classes, paramétrées sur les types, appelées génériques. Une classe générique `LIST [G]` décrira les listes d'éléments d'un type arbitraire représenté par `G`, le "paramètre générique formel" ; vous pouvez alors déclarer des listes spécifiques par l'intermédiaire de dérivations comme `LIST [INTEGER]` et

LIST [*WINDOW*] , en utilisant les types *INTEGER* et *WINDOW* comme “paramètres génériques réels”. Toutes les dérivations partagent le même texte de classe.

> Il devrait être possible d’écrire des classes ayant des paramètres génériques formels qui représentent des types arbitraires.

Cette forme de paramétrisation de type est appelée généricité **non contrainte**. Une possibilité voisine mentionnée ci-dessous, la généricité contrainte, utilise l’héritage.

Héritage simple

Le développement de logiciel manipule un grand nombre de classes ; beaucoup sont des variantes d’autres. Pour maîtriser la complexité potentielle qui peut en résulter, nous avons besoin d’un mécanisme de classification, appelé héritage. Une classe sera héritière d’une autre si elle incorpore les caractéristiques d’une autre aux siennes. (Un *descendant* est un héritier direct ou indirect ; la notion inverse correspond à un *ancêtre*.)

> Il devrait être possible de définir une classe par héritage d’une autre.

L’héritage est un des concepts centraux des méthodes orientées objet et a des conséquences profondes sur le processus de développement logiciel.

Héritage multiple

Nous aurons souvent besoin de combiner plusieurs abstractions. Une classe peut, par exemple, modéliser la notion de “nourrisson”, que nous pouvons voir à la fois comme une “personne”, avec ses caractéristiques propres, et, plus prosaïquement, comme un “article déductible des impôts”. L’héritage *multiple* est la garantie qu’une classe peut hériter non seulement d’une autre, mais d’autant de classes que cela est conceptuellement justifié.

L’héritage multiple soulève un certain nombre de problèmes techniques, en particulier la résolution des *conflits de noms* (configurations dans lesquelles des caractéristiques différentes, héritées de classes distinctes, ont le même nom). Toute notation permettant l’héritage multiple doit fournir une solution appropriée à ces problèmes.

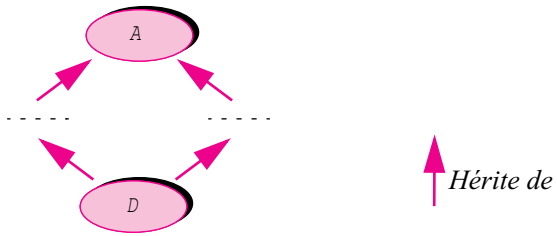
> Il devrait être possible à une classe d’hériter d’autant de classes qu’il est nécessaire, grâce à un mécanisme adéquat permettant de régler les conflits de noms.

La solution développée dans ce livre est basée sur le renommage (*renaming*) des caractéristiques en conflit dans la classe héritière.

Héritage répété

L'héritage multiple introduit la possibilité d'héritage *répété,* correspondant au cas dans lequel une classe hérite d'une autre via deux chemins ou plus, comme indiqué ci-dessous.

Héritage répété



Dans ce cas, le langage doit fournir des règles précisant ce que deviennent les caractéristiques successivement héritées d'un ancêtre commun, `A` dans la figure. Comme on le verra lors de la discussion sur l'héritage répété, il peut être préférable, dans certains cas, qu'une caractéristique de `A` corresponde à une seule caractéristique de `D` (*partage*), tandis que, dans d'autres, elle donne lieu à deux versions (*réplication*). Les développeurs doivent être libres d'imposer l'une ou l'autre de ces politiques séparément pour chaque caractéristique.

> Des règles précises doivent gouverner le sort des caractéristiques soumises à héritage répété, permettant aux développeurs de choisir, séparément pour chaque caractéristique héritée répétitivement, entre partage et réplication.

Généricité contrainte

La combinaison de la généricité et de l'héritage permet d'introduire une technique puissante, la généricité contrainte, grâce à laquelle vous pouvez spécifier qu'un paramètre générique d'une classe ne représente pas un type arbitraire, comme dans la forme précédente de généricité (non-contrainte), mais un type qui est un descendant d'une classe donnée.

Une classe générique `SORTABLE_LIST`, décrivant les listes ayant une caractéristique `sort` qui permet de les réordonner séquentiellement selon une certaine relation d'ordre, a besoin d'un paramètre générique qui représente le type des éléments de la liste. Ce type n'est pas arbitraire : il doit admettre une relation d'ordre. Pour indiquer que n'importe quel paramètre générique réel doit être un descendant de la classe bibliothèque `COMPARABLE`, décrivant les objets munis d'une relation d'ordre, on utilise la généricité contrainte pour déclarer la classe comme `SORTABLE_LIST [G -> COMPARABLE]`.

> Le mécanisme de généricité devrait permettre une forme contrainte de généricité.

Redéfinition

Quand une classe hérite d'une autre, elle peut avoir besoin de changer l'implémentation, ou d'autres propriétés, de certaines des caractéristiques dont elle hérite. Une classe `SESSION` décrivant les sessions des utilisateurs dans un système d'exploitation peut offrir une

caractéristique *terminate* pour prendre en compte les opérations de nettoyage à la fin d'une session ; un héritier peut en être *REMOTE_SESSION*, qui gère des sessions initialisées à partir d'un ordinateur distant sur un réseau. Si la clôture d'une session distante nécessite des actions supplémentaires (comme prévenir l'ordinateur distant), la classe *REMOTE_SESSION* redéfinira la caractéristique *terminate*.

La redéfinition peut influer sur l'implémentation d'une caractéristique, sa signature (type des arguments et du résultat), et sa spécification.

Il devrait être possible de redéfinir la spécification, la signature et l'implémentation d'une caractéristique héritée.

Polymorphisme

Avec l'introduction de l'héritage dans le système, l'exigence de typage statique évoquée plus haut serait trop restrictive si elle se limitait à imposer que chaque entité déclarée de type *C* ne puisse référer qu'à des objets dont le type est exactement *C*. Ceci voudrait dire, par exemple, qu'une entité de type *C* (dans un système de contrôle de la navigation) ne pourrait pas être utilisée pour faire référence à un objet de type *MERCHANT_SHIP* ou *SPORTS_BOAT*, deux types que nous supposons être des classes héritant de *BOAT*.

> Comme indiqué ci-dessus, une "entité" est un nom auquel peuvent être liées plusieurs valeurs au moment de l'exécution. C'est une généralisation de la notion traditionnelle de variable.

Le polymorphisme est la capacité qu'a une entité de pouvoir être attachée à des objets de types variés. Dans un environnement statiquement typé, le polymorphisme ne sera pas arbitraire, mais contrôlé par l'héritage ; par exemple, nous ne devrions pas autoriser l'attachement de notre entité *BOAT* à un objet bouée de type *BUOY*, une classe qui n'hérite pas de *BOAT*.

Il devrait être possible d'attacher à des entités (noms qui représentent, dans le texte des logiciels, des objets à l'exécution) des objets à l'exécution ayant divers types possibles, sous contrôle du système de type basé sur l'héritage.

Liaison dynamique

La combinaison des deux derniers mécanismes que nous venons de mentionner, redéfinition et polymorphisme, en suggère immédiatement un nouveau. Soit un appel dont la cible est une entité polymorphique, par exemple un appel à la caractéristique *turn* sur une entité déclarée de type *BOAT*. Les divers descendants de *BOAT* peuvent avoir redéfini la caractéristique de différentes manières. Un mécanisme automatique devra donc garantir que la version de *turn* utilisée sera toujours celle déduite du type de l'objet réel, indépendamment de la manière dont l'entité a été déclarée. Cette propriété est appelée liaison dynamique.

L'appel d'une caractéristique sur une entité devrait toujours déclencher la caractéristique correspondant au type de l'objet attaché à l'exécution, qui n'est pas nécessairement le même lors des différentes exécutions de l'appel.

La liaison dynamique a une influence majeure sur la structures des applications orientées objet, car elle permet aux développeurs d'écrire des appels simples (c'est-à-dire, par exemple, "appeler la caractéristique *turn* de l'entité *my_boat*") pour indiquer ce qui correspond, en fait, à plusieurs appels possibles en fonction de la situation au moment de l'exécution. Cela permet d'éviter une grande partie des tests répétitifs ("Est-ce un bateau commercial ? Est-ce un bateau de sport ?") qui polluent le logiciel écrit selon des approches plus conventionnelles.

Interrogation de type à l'exécution

Les développeurs de logiciel orienté objet ont rapidement une hantise de bon aloi pour tout style de calcul basé sur des choix explicites entre les divers types d'un objet. Le polymorphisme et la liaison dynamique fournissent une alternative nettement préférable. Dans certains cas, néanmoins, un objet provient de l'extérieur, de telle sorte que l'auteur du logiciel n'a aucun moyen de prédire son type avec certitude. Cela se produit, en particulier, si l'objet provient d'un mécanisme de stockage externe, est reçu lors d'une transmission sur réseau ou est transferé d'un autre système.

Le logiciel a alors besoin d'un mécanisme pour accéder à l'objet de manière fiable, sans violer les contraintes du typage statique. Un tel mécanisme devrait être conçu avec soin de manière à ne pas anéantir les avantages du polymorphisme et de la liaison dynamique.

L'opération de **tentative d'affectation** décrite dans ce livre remplit ces exigences. Une tentative d'affectation est une opération conditionnelle : elle essaie d'attacher un objet à une entité ; si, lors d'une exécution donnée, le type de l'objet est conforme au type déclaré pour l'entité, l'effet est celui d'une affectation normale ; sinon l'entité reçoit la valeur spéciale "void". Ainsi, vous pouvez gérer des objets dont le type ne vous est pas connu avec certitude, sans violer la sûreté du système de type.

Il devrait être possible de déterminer, au moment de l'exécution, si le type d'un objet est conforme à un type statique donné.

Caractéristiques et classes retardées

Dans certains cas où la liaison dynamique fournit une solution élégante en éliminant les tests explicites, il n'y a pas de version initiale de caractéristique à redéfinir. Par exemple, la classe *BOAT* peut être trop générale pour fournir une implémentation par défaut de *turn*. Pourtant, nous désirons être à même d'appeler la caractéristique *turn* d'une entité déclarée de type *BOAT* si nous nous sommes assurés qu'elle sera attachée, au moment de l'exécution, à des objets ayant des types complètement définis comme *MERCHANT_SHIP* et *SPORTS_BOAT*.

Dans ces conditions, *BOAT* peut être déclarée comme classe retardée (une classe qui n'est pas complètement implémentée), avec la caractéristique retardée *turn*. Les caractéristiques et classes retardées peuvent néanmoins posséder des assertions décrivant leurs propriétés abstraites, mais leurs implémentations sont différées aux classes descendantes. Une classe non retardée est dite *effective*.

Il devrait être possible d'écrire qu'une classe ou une caractéristique est retardée, c'est-à-dire spécifiée mais non implémentée.

Les classes retardées (appelées aussi classes abstraites) sont particulièrement importantes dans l'analyse orientée objet et la conception de haut niveau, car elles rendent possible la description des aspects essentiels d'un système tout en en différant les détails.

Gestion de la mémoire et ramasse-miettes

Le dernier élément de notre liste de critères sur les méthodes et les langages peut sembler, au premier abord, appartenir plutôt à la seconde catégorie — implémentation et environnement. En fait, il appartient aux deux. Mais les exigences essentielles s'appliquent au langage ; le reste n'est qu'une question de bonne ingénierie.

Les systèmes orientés objet, plus encore que les programmes traditionnels (excepté Lisp), ont tendance à créer de nombreux objets aux interdépendances parfois complexes. Une politique laissant aux développeurs la charge de gérer la mémoire correspondante, en particulier quand il s'agit de récupérer l'espace occupé par les objets dont l'on n'a plus besoin, nuirait tout à la fois à l'efficacité du processus de développement, car elle compliquerait le logiciel et prendrait une part importante du temps des développeurs, et à la sûreté des systèmes résultants, car elle accroîtrait le risque d'un mauvais recyclage des zones mémoire. Dans un bon environnement orienté objet, la gestion de la mémoire sera automatique, sous le contrôle d'un *ramasse-miettes*, un composant de l'exécutif.

Cette question de langage importe autant que l'exigence d'implémentation, car un langage qui n'a pas été explicitement conçu pour la gestion automatique de la mémoire la rendra souvent impossible. C'est le cas des langages dans lesquels un pointeur vers un objet d'un certain type peut se déguiser (en utilisant des conversion appelées "coercions") en un pointeur d'un autre type ou même en un entier, rendant impossible l'écriture d'un ramasse-miettes sûr.

Le langage devrait rendre possible une gestion automatique de la mémoire et son implémentation devrait fournir un gestionnaire automatique de mémoire prenant en compte le processus de ramasse-miettes.

2.3 IMPLÉMENTATION ET ENVIRONNEMENT

Nous en arrivons maintenant aux éléments essentiels d'un environnement de développement qui permet la construction de logiciel orienté objet.

Mise à jour automatique

Le développement du logiciel est un processus incrémental. Les développeurs écrivent rarement des milliers de lignes d'un coup ; ils procèdent par ajouts et modifications, débutant la plupart du temps avec un système qui est déjà de taille substantielle.

Lors de telles mises à jour, il est essentiel d'avoir la garantie que le système résultant sera cohérent. Par exemple, si vous changez la caractéristique `f` d'une classe `C`, vous devez être

certain que chaque descendant de `C` qui ne redéfinit pas `f` sera mis à jour avec la nouvelle version de `f`, et que chaque appel à `f` dans un client de `C` ou dans un des descendants de `C` déclenchera la nouvelle version.

Les approches classiques de ce problème sont manuelles, forçant les développeurs à enregistrer toutes les dépendances, ainsi que leurs changements, en utilisant des mécanismes spéciaux appelés "fichiers make" et "fichiers include". C'est inacceptable quand on développe du logiciel de manière moderne, particulièrement dans un monde orienté objet où les dépendances entre classes, résultant des relations client et héritage, sont souvent complexes, mais peuvent être déduites par un examen systématique du texte du logiciel.

> La mise à jour d'un système devrait être automatique, l'analyse des dépendances interclasses étant effectuée par des outils et non manuellement par les développeurs.

Il est possible de remplir cette exigence dans un environnement compilé (où le compilateur travaillera de concert avec un outil d'analyse des dépendances), dans un environnement interprété ou dans une combinaison de ces deux techniques d'implémentation.

Mise à jour rapide

En pratique, le mécanisme de mise à jour d'un système devrait être non seulement automatique, mais rapide. Plus précisément, il devrait être proportionnel à la taille des modifications, non à celle du système global. Sans cette propriété, la méthode et l'environnement peuvent être applicables à de petits systèmes, mais pas à ceux plus importants.

> Le temps mis pour prendre en compte un ensemble de changements d'un système, autorisant l'exécution de la version mise à jour, devrait être fonction de la taille des composants modifiés, et non de celle du système global.

Ici également, les environnements interprétés et compilés peuvent aussi bien remplir ce critère, bien que, dans ce dernier cas, le compilateur doive être incrémental. Outre un compilateur incrémental, l'environnement peut bien sûr offrir un compilateur muni d'un optimiseur global travaillant sur l'ensemble du système, sous réserve que ce compilateur ne soit utilisé que pour délivrer un produit final ; le développement utilisera le compilateur incrémental.

Persistance

Beaucoup d'applications auront sans doute besoin de sauvegarder des objets d'une session à l'autre. L'environnement devrait fournir un mécanisme permettant de le faire simplement.

Un objet contiendra souvent des références à d'autres objets ; puisque ceci peut être également vrai de ces derniers, chaque objet peut donc avoir un grand nombre d'objets *dépendants*, formant souvent un graphe de dépendance complexe (qui peut contenir des cycles). Il serait absurde de stocker ou de récupérer un objet sans toutes ses dépendances directes et indirectes.

On dit qu'un mécanisme de persistance qui peut automatiquement stocker un objet et ses dépendances fournit une **fermeture persistante**.

Un mécanisme de stockage persistant offrant la fermeture persistante devrait être disponible pour stocker un objet et tous ses objets dépendants sur des supports externes, et pour les récupérer dans la même session ou une autre.

Pour certaines applications, une persistance simple n'est pas suffisante ; une **base de données** sera alors nécessaire. La notion de bases de données orientées objet est évoquée dans un prochain chapitre, qui explore également les aspects concernant la persistance comme l'*évolution de schéma*, c'est-à-dire la possibilité de récupérer des objets de manière sûre même quand les classes correspondantes ont changé.

Documentation

Les développeurs de classes et de systèmes doivent fournir aux gestionnaires, aux clients et aux autres développeurs des descriptions claires et de haut niveau du logiciel qu'ils produisent. Ils ont besoin d'outils pour les assister dans cet effort ; la documentation devrait, autant que possible, être automatiquement produite à partir des textes des logiciels. Les assertions permettent, comme on l'a vu, de rendre de tels documents, extraits du logiciel, à la fois précis et informatifs.

Des outils automatiques devraient être disponibles pour produire la documentation des classes et des systèmes.

Navigation

Quand vous analysez une classe, vous aurez souvent besoin d'obtenir des informations concernant d'autres classes ; en particulier, les caractéristiques utilisées dans une classe peuvent avoir été introduites, non par la classe elle-même, mais par ses divers ancêtres. L'environnement doit donc fournir aux développeurs des outils avec lesquels ils peuvent examiner le texte d'une classe, trouver ses dépendances vis-à-vis des autres classes et passer rapidement du texte d'une classe à un autre.

Cette tâche est appelée navigation. Parmi les possibilités offertes par les bons outils de navigation, on trouve : rechercher les clients, fournisseurs, descendants et ancêtres d'une classe ; trouver toutes les redéfinitions d'une caractéristique ; trouver la déclaration initiale d'une caractéristique redéfinie.

S est un "fournisseur" de C si C est un client de S. "Client" a été défini page 27.

Des capacités de navigation interactive devraient permettre aux développeurs logiciels de suivre facilement et rapidement les dépendances entre classes et caractéristiques.

2.4 BIBLIOTHÈQUES

Un des aspects les plus caractéristiques du développement de logiciel orienté objet est l'importance donnée aux bibliothèques. Un environnement orienté objet devrait fournir de bonnes bibliothèques, ainsi que des mécanismes permettant d'en écrire de nouvelles.

Bibliothèques de base

Les structures de données fondamentales de l'informatique — ensembles, listes, arbres, piles… — et les algorithmes associés — tri, recherche, parcours, association de modèles — sont omniprésents dans le développement logiciel. Dans les approches conventionnelles, chaque développeur les implémente ou les réimplémente tout le temps ; outre un gaspillage d'efforts, cela nuit également à la qualité du logiciel, car un développeur individuel qui implémente une structure de données, non pour elle-même mais comme simple composant d'une application, atteindra difficilement un optimum de fiabilité et d'efficacité.

Un environnement de développement orienté objet doit fournir des classes réutilisables correspondant à ces besoins, fréquents dans les systèmes logiciels.

> Des classes réutilisables devraient être disponibles pour traiter les structures de données et les algorithmes les plus fréquemment utilisés.

Interfaces graphiques et interfaces utilisateur

De nombreux systèmes logiciels modernes sont interactifs, communiquant avec leurs utilisateurs par des techniques d'interfaces graphiques ou autres artefacts attrayants. C'est l'un des domaines où le modèle orienté objet s'est révélé le plus pratique et impressionnant. Des développeurs devraient pouvoir compter sur des bibliothèques graphiques pour construire rapidement et efficacement des applications interactives.

> Des classes réutilisables devraient être disponibles pour développer des applications qui fournissent à leurs utilisateurs des interfaces graphiques attrayantes.

Mécanismes d'évolution des bibliothèques

Développer des bibliothèques de grande qualité est une tâche longue et pénible. Il est impossible de garantir que la conception d'une bibliothèque sera parfaite du premier coup. Il faudra alors permettre aux développeurs de bibliothèques de mettre à jour et de modifier leurs programmes sans mettre en péril les systèmes existants qui dépendent de ces bibliothèques. Ce critère appartient à la catégorie bibliothèque, mais aussi à la catégorie méthode et langage.

> Des mécanismes devraient être disponibles pour faciliter l'évolution des bibliothèques, et ce en dérangeant le moins possible le logiciel client.

Mécanismes d'indexation des bibliothèques

Un autre problème soulevé par les bibliothèques est celui de l'identification des classes correspondant à un certain besoin. Ce critère touche les trois catégories : bibliothèques, langage (puisqu'il doit exister un moyen permettant d'introduire des informations d'indexation dans le texte de chaque classe) et outils (pour traiter les requêtes concernant les classes qui satisfont à certaines conditions).

> Les classes bibliothèque devraient être munies d'informations d'indexation permettant des recherches en fonction de ces propriétés.

2.5 POUR UNE BANDE-ANNONCE PLUS LONGUE

Bien qu'il soit préférable, pour comprendre les concepts en profondeur, de lire ce livre en séquence, les lecteurs qui voudraient étoffer le survol théorique précédent avec un exemple concret peuvent, dès à présent, se reporter au chapitre 20, qui présente une étude de cas sur un problème pratique de conception dans lequel on compare une solution OO avec les techniques plus traditionnelles.

Cette étude de cas est indépendante, et vous pouvez en comprendre l'essentiel sans avoir lu les chapitres intermédiaires. (Mais si vous allez jeter un coup d'œil là-bas, vous devez promettre de revenir au reste de la présentation séquentielle, qui débute au chapitre 3, dès que vous aurez fini.)

2.6 NOTES BIBLIOGRAPHIQUES ET RESSOURCES OBJET

Cette introduction aux critères de l'orientation objet nous donne l'occasion de présenter un échantillon de livres qui offrent de bonnes introductions générales à la technologie objet.

[Waldén 1995] traite des questions les plus importantes de la technologie objet, se concentrant sur l'analyse et la conception, points sur lesquels il est probablement la meilleure référence.

[Page-Jones 1995] fournit un excellent survol de la méthode.

[Cox 1990] (dont la première édition date de 1986) est basé sur un point de vue un peu différent de la technologie objet et a joué un rôle essentiel dans la diffusion des concepts OO à une audience élargie.

[Henderson-Sellers 1991] (une seconde édition est annoncée) fournit un rapide survol des idées OO. Ciblant les personnes à qui il a été demandé, au sein de leur entreprise, d'"aller voir ce que c'est que cette histoire d'objet", il offre des originaux de transparents prêts à être photocopiés, ce qui peut être précieux dans de telles occasions. Un autre survol est [Eliëns 1995].

Le *Dictionary of Object Technology* [Firesmith 1995] fournit une riche référence sur de nombreux aspects de la méthode.

Tous ces livres visent, à un degré ou un autre, les personnes intéressées par la technique. Il est également utile d'éduquer les gestionnaires. [M 1995] étoffe un chapitre qui devait au départ être utilisé dans ce livre, et qui est devenu une étude approfondie de la technologie objet

destinée aux gestionnaires. Il débute par une présentation technique rapide, élaborée en termes compréhensibles pour les gestionnaires, et se poursuit par l'analyse des questions touchant la gestion (cycle de vie, gestion de projet, politiques de réutilisation). Un autre livre orienté gestion, [Goldberg 1995], fournit une perspective complémentaire sur de nombreux sujets importants. [Baudoin 1996] insiste sur les aspects du cycle de vie et sur l'importance des standards.

Pour revenir aux présentations techniques, trois livres majeurs sur des langages orientés objet, écrits par les concepteurs de ces langages, contiennent des études méthodologiques générales pouvant intéresser même ceux qui n'utilisent pas ces langages ou les critiquent. (L'histoire des langages de programmation montre que les concepteurs ne sont pas toujours les meilleurs défenseurs de leurs propres créations mais, en l'occurrence, ils l'étaient.) Ces livres sont :

- *Simula BEGIN* [Birtwistle 1973]. (Ici deux autres auteurs se sont joints aux concepteurs du langage Nygaard et Dahl.)
- *Smalltalk-80* : *The Language and its Implementation* [Goldberg 1983].
- *The C++ Programming Language*, *second edition* [Stroustrup 1991].

Le chapitre 29 étudie l'enseignement de cette technologie.

Plus récemment, des livres de cours d'introduction à la programmation ont d'emblée fait appel aux idées orientées objet, car il n'y a pas de raison que "l'ontogenèse copie la phylogenèse", c'est-à-dire de balader les pauvres étudiants à travers l'histoire des hésitations et des erreurs rencontrées par leurs prédécesseurs avant d'arriver aux bonnes idées. Le premier livre de ce type a été (à ma connaissance) [Rist 1995]. Un autre bon livre couvrant des besoins similaires est [Wiener 1996]. Au niveau supérieur — livres d'enseignement pour un cours avancé sur la programmation, traitant des structures de données et des algorithmes à l'aide des notations de ce livre — vous trouverez [Gore 1996] et [Wiener 1997] ; [Jézéquel 1996] présente les principes de génie logiciel orienté objet.

Le groupe de discussion *comp.object* sur Usenet, archivé par de nombreux sites sur le Web, est le média naturel de discussion pour toute la technologie objet. Comme toujours dans de tels forums, soyez prêt à absorber un mélange alliant de bon, de mauvais et de détestable. La rubrique Object Technology de *Computer* (IEEE), que j'édite depuis sa sortie en 1995, invite fréquemment dans ses colonnes des experts renommés.

Parmi les magazines consacrés à la Technologie Objet :

- Le *Journal of Object-Oriented Programming* (le premier journal du domaine, se concentrant sur les discussions techniques tout en visant une large audience), *Object Magazine* (de contenu éditorial plus large, avec parfois des articles pour les gestionnaires), *Objekt Spektrum* (Allemand), *Object Currents* (on-line), tous décrits dans *http://www.sigs.com*.
- *Theory and Practice of Object Systems*, un journal de référence.
- *L'OBJET* (Français), décrit dans *http://www.tools.com/lobjet*.

Les principales conférences internationales OO sont OOPSLA (annuel, USA ou Canada, voir *http://www.acm.org*) ; *Object Expo* (fréquence et endroit variables, décrit dans *http://www.sigs.com*) ; et TOOLS (Technology of Object-Oriented Languages and Systems), organisée par ISE avec trois sessions par an (USA, Europe, Pacifique), dont la page principale, dans *http://www.tools.com*, sert également de ressource générale sur la technologie objet et les sujets de ce livre.