Conception et programmation orientées objet

Bertrand Meyer

Traduit de l'anglais par Pierre Jouvelot

 ISBN : 978-2-212-12270-1

EYROLLES

22

Comment trouver les classes

Puisque la structure du logiciel OO est fondée sur une décomposition en classes, la méthodologie orientée objet devrait commencer par indiquer la manière de trouver ces classes. Telle est l'ambition des pages qui suivent. (Dans une frange de la littérature, vous trouverez ce problème référencé sous le titre de "*trouver les objets*", mais nous savons qu'il ne s'agit pas que de cela : ce ne sont pas les objets individuels qui sont en jeu dans nos architectures logicielles, mais les types des objets — les classes.)

Avant tout, il ne faut pas s'attendre à des miracles. Trouver des classes est une décision primordiale quand on construit un système logiciel orienté objet ; comme dans toute activité créatrice, prendre de bonnes décisions demande talent et expérience, sans oublier la chance. Espérer trouver des recettes infaillibles pour découvrir les classes est aussi irréaliste que le serait, pour un mathématicien, de trouver des recettes d'invention de théories intéressantes et prouver ses théorèmes. Bien que ces deux activités — construction logicielle et construction théorique — puissent bénéficier de conseils généraux et d'exemples d'illustres prédécesseurs, toutes deux demandent également une créativité qui ne peut être complètement décrite par des règles mécaniques. Si (comme souvent dans l'industrie) comparer un développeur logiciel et un mathématicien vous paraît difficile, pensez simplement à d'autres formes de conception d'ingénierie : bien qu'il soit possible de fournir des règles de conduite de base, aucune règle opérant par étapes ne peut garantir une bonne conception de bâtiments ou d'avions.

Dans le logiciel aussi, aucun conseil livresque ne peut remplacer le savoir-faire et l'ingéniosité. Le rôle principal d'une étude méthodologique est de suggérer quelques bonnes idées, d'attirer l'attention sur certains précédents formateurs et d'avertir de la présence de pièges connus.

En fait, cela serait le cas pour toute méthode de conception logicielle autre que celle évoquée ici. Pour la technologie objet, cette constatation s'avère moins limitée qu'il n'y paraît, car la réutilisabilité nous fournira de bonnes surprises. Puisqu'une grande partie de ce dont nous avons besoin lors des développements logiciels objet a peut-être déjà été inventé, nous pouvons nous inspirer des réalisations des autres.

Et ce n'est pas tout. Avec des objectifs initiaux modestes, mais en étudiant soigneusement ce qui marche et ce qui ne marche pas, nous arriverons, petit à petit et contre toute attente, à concevoir ce qui, au bout du compte, méritera le nom de *méthode* pour trouver les classes. Une des étapes clés sera de réaliser qu'une technique de sélection est définie, comme toujours en conception, par deux composantes : ce qu'il faut garder, et ce qu'il faut rejeter.

22.1 ÉTUDIER UN DOCUMENT D'EXIGENCES

Pour comprendre le problème que pose la recherche des classes, commençons par étudier une approche largement diffusée.

Les noms et les verbes

Voir les notes bibliographiques.

Un certain nombre de publications suggèrent d'utiliser une règle simple pour obtenir les classes : commencer avec le document d'exigences (en supposant, bien sûr, qu'il en existe un, mais c'est une autre histoire) ; dans une conception orientée fonction, vous vous concentrez sur les verbes, qui correspondent aux actions ("faites ceci") ; dans une conception orientée objet, vous soulignez les noms, qui décrivent les objets. Ainsi, selon ce point de vue, une phrase telle que :

L'ascenseur fermera sa porte avant de se déplacer vers un autre étage.

conduit un concepteur orienté fonction à détecter qu'il faut une fonction "déplacer" ; alors qu'un concepteur orienté objet va voir trois types d'objets, `ELEVATOR`, `DOOR` et `FLOOR`, d'où les classes correspondantes. Et voilà !

Ne serait-ce pas merveilleux si la vie était aussi simple ? Vous apporteriez votre document d'exigences le soir, et joueriez à *La poursuite de l'objet* autour de la table de la salle à manger. Une bonne manière d'éloigner les enfants du poste de télévision et de leur faire réviser leurs leçons de grammaire pendant qu'ils aident Maman et Papa dans leurs devoirs de génie logiciel.

Mais une technique aussi simpliste ne peut pas nous mener bien loin. Le langage humain, utilisé pour exprimer les exigences d'un système, est si sujet à nuances, interprétations personnelles et ambiguïtés qu'il est dangereux de prendre une décision importante sur la base d'un document qui peut tout autant être influencé par le style personnel de l'auteur que par les propriétés réelles du système logiciel envisagé.

La méthode "souligner les noms" nous conduira, de toutes façons, à des résultats triviaux. Toute conception OO d'un système de commande d'un ascenseur contiendra, bien évidemment, une classe `ELEVATOR`. Obtenir de telles classes n'est pas difficile. Pour reprendre une expression utilisée précédemment, elles sont prêtes à être cueillies. En revanche, pour les classes non évidentes, un critère syntaxique — comme prendre les noms et non les verbes d'un document qui, par essence, est susceptible d'avoir plusieurs variantes stylistiques — n'a qu'une valeur minime.

Bien que l'idée de "souligner les noms" ne mérite, en soi, guère plus de considération, nous pouvons la pousser plus loin, pas pour son mérite mais pour l'utiliser comme repoussoir ; comprendre ses limites nous aidera à déterminer ce qui est vraiment nécessaire pour trouver les classes, et en quoi le document d'exigences peut nous aider en cela.

Éviter les classes inutiles

Les noms d'un document d'exigences couvriront certaines classes de la conception finale, mais constitueront souvent de "fausses alarmes" : des concepts qui ne devraient *pas* correspondre à des classes.

Dans l'exemple de l'ascenseur, la *porte* est un nom. Avons-nous cependant besoin d'une classe `DOOR` ? Peut-être oui, peut-être non. Il est possible que la seule propriété pertinente des portes d'un ascenseur soit d'être ouvertes ou fermées. Pour exprimer les propriétés utiles des portes, il suffit d'introduire dans la classe `ELEVATOR` la requête et les commandes :

```
door_open: BOOLEAN;
close_door is
        ...
        ensure
                not door_open
        end;
open_door is
        ...
        ensure
                door_open
        end
```

Dans une autre variante du système, toutefois, la notion de porte peut être suffisamment importante pour justifier une classe séparée. Le seul fondement solide est ici la théorie des types abstraits de données, et la seule question pertinente est :

> Une "porte" est-elle un type de données distinct possédant ses propres opérations clairement identifiées, ou toutes les opérations concernant les portes sont-elles déjà gérées par des opérations concernant d'autres types de données comme *ELEVATOR* ?

Seules votre intuition et votre expérience de concepteur vous donneront la réponse. Le document d'exigences pourra vous y aider, mais ne vous attendez pas que les critères grammaticaux soient autre chose qu'une aide superficielle. La théorie des ADT vous sera d'un plus grand secours pour poser à vos clients ou futurs utilisateurs les bonnes questions.

Chapitre 21.

Nous avons rencontré un cas similaire lors de la conception du mécanisme défaire-refaire. L'étude a conduit à distinguer les *commandes*, comme la commande d'insertion de ligne d'un éditeur de texte, de la notion plus générale d'*opération*, qui comprend les commandes, mais aussi les requêtes spéciales comme défaire. Ces deux mots apparaissaient très explicitement dans l'énoncé du problème ; pourtant, seule *COMMAND* a conduit à une abstraction de données (une des classes principales de la conception), alors qu'aucune classe de la solution ne reflétait directement la notion d'opération. Aucune analyse d'un document d'exigences ne peut conduire à une telle différence de traitement.

Est-ce qu'une nouvelle classe est nécessaire ?

Dans l'exemple de l'ascenseur, la notion d'*étage* est un autre exemple de nom qui peut, ou non, amener à introduire une classe. La question, ici (par opposition aux cas des *portes* et des *opérations*), n'est pas de savoir si le concept est un ADT pertinent : les étages constituent une abstraction de données importante d'un système pour ascenseurs. Mais cela ne veut pas nécessairement dire que nous devrions avoir une classe d'étage *FLOOR*.

De nombreux hôtels n'ont pas d'étage 13, et l'arithmétique peut devenir un peu plus complexe.

Les propriétés des étages peuvent, tout simplement, être entièrement décrites, en ce qui concerne les missions d'un système pour ascenseurs, par celles des entiers. Chaque étage a un numéro ; si un étage (tel que le considère un système pour ascenseurs) n'a d'autres caractéristiques que celles associées à son numéro d'étage, une classe *FLOOR* distincte est inutile. Ainsi, la distance entre deux étages ne sera que la simple différence de leurs numéros d'étage.

En revanche, si les étages ont des propriétés autres que celles de leurs numéros — c'est-à-dire, selon les principes des types abstraits de données et de la construction logicielle orientée objet,

des *opérations* significatives qui ne sont pas prises en compte par celles des entiers — une classe `FLOOR` sera alors appropriée. Par exemple, certains étages peuvent posséder des droits d'accès particuliers pour les personnes autorisées à s'y arrêter ; la classe `FLOOR` pourrait alors contenir une caractéristique comme :

```
rights: SET [ AUTHORIZATION]
```

et les procédures associées. Mais, cela n'est même pas certain : nous pourrions nous en sortir en introduisant, dans une autre classe, un tableau :

```
floor_rights: ARRAY [ SET [ AUTHORIZATION] ]
```

qui associerait un ensemble de valeurs d'`AUTHORIZATION` à chaque étage, identifié par son numéro.

Voir exercice E22.1, page 722.

Un autre argument en faveur d'une classe spécifique `FLOOR` serait de limiter les opérations disponibles : soustraire deux étages et les comparer (via la fonction **infix** "<") ont un sens, alors que les ajouter ou les multiplier n'en a pas. Une telle classe peut être écrite comme héritant de `INTEGER`. Le concepteur doit se demander, cependant, si cette motivation justifie vraiment d'ajouter une nouvelle classe.

L'analyse nous ramène une fois de plus à la théorie des types abstraits de données. Une classe ne correspond pas simplement aux "objets" physiques, au sens naïf du terme. Elle décrit un type abstrait de données — un ensemble d'objets logiciels caractérisés par des opérations bien définies, et les propriétés formelles de ces opérations. Un type d'objets du monde réel peut, ou non, avoir un équivalent dans le logiciel, sous la forme d'un type d'objets logiciels — une classe. Quand vous cherchez à évaluer si une certaine notion devrait, ou non, conduire à une classe, seule la vision ADT peut fournir le bon critère : est-ce que les objets du système étudié présentent suffisamment d'opérations et de propriétés spécifiques liées au système et non traitées par les classes existantes ?

"AU-DELÀ DU LOGICIEL", 6.6, page 151.

La qualification "liées au système" est cruciale. L'objectif de l'analyse des systèmes n'est pas de "modéliser le monde". Cela peut correspondre à la tâche des philosophes, mais les constructeurs de systèmes logiciels n'en ont cure, tout au moins dans leur activité professionnelle. La tâche d'analyse consiste à modéliser la partie du monde pertinente pour le logiciel en cours d'étude ou de construction. Ce principe est renforcé par l'approche ADT (c'est-à-dire la méthode orientée objet), qui prétend que *les objets ne sont définis que par ce qu'on leur fait* — ce que nous avons appelé, lors du traitement des types abstraits de données, le principe d'égoïsme. Si une opération ou propriété d'un objet n'est pas pertinente par rapport aux objectifs du système, elle ne devrait pas être introduite dans votre analyse — aussi intéressante qu'elle puisse être par ailleurs. Pour un système de traitement du recensement, la notion de `PERSON` peut avoir les caractéristiques `mother` et `father` ; mais, pour un système de traitement de la paie qui n'a pas besoin d'informations concernant les parents, toute `PERSON` est un orphelin.

Si toutes les opérations et propriétés que vous pouvez identifier dans un type d'objets ne sont pas pertinentes que dans ce sens-là, ou sont déjà traitées par les opérations et les propriétés d'une classe identifiée précédemment, la conclusion à tirer est que le type d'objets lui-même est non pertinent : il ne doit pas conduire à une classe.

Ceci explique pourquoi il n'est pas forcément nécessaire d'introduire, dans un système pour ascenseurs, de classe `FLOOR`, puisque (comme on l'a noté ci-dessus), du point de vue du système, les étages n'ont aucune propriété pertinente autre que leurs numéros entiers associés, alors qu'un système de conception assistée par ordinateur pour architectes offrira une classe `FLOOR` — puisque, dans ce cas, l'étage a de nombreux attributs et routines spécifiques.

Oublier des classes importantes

Une analyse fondée sur les noms peut, on l'a vu, conduire à privilégier des notions qui ne devraient pas correspondre à des classes. Mais elle peut aussi laisser de côté des notions qui devraient, sans aucun doute, correspondre à des classes. Il y a au moins trois sources possibles d'oubli de ce genre.

> N'oubliez pas, comme nous l'avons noté, que le but de cette étude n'est plus de nous convaincre des limitations de l'approche consistant à "souligner les noms", limitations qui sont, dorénavant, tellement évidentes que l'exercice ne serait pas très productif. En revanche, nous analysons ces limitations en vue d'améliorer notre compréhension du processus de découverte de ces classes.

La première cause d'oubli provient de la flexibilité et de l'ambiguïté du langage humain — qualités mêmes qui le rendent adapté à un si large éventail d'applications, des conférences aux romans en passant par les lettres d'amour, mais qui en font un medium peu fiable pour les documents techniques. Supposez que le document d'exigences de notre exemple d'ascenseur contienne la phrase :

> Un enregistrement de la base de données doit être créé chaque fois que l'ascenseur se déplace d'un étage à un autre.

La présence du nom "enregistrement" suggère une classe *DATABASE_RECORD* ; mais il se peut que nous passions totalement à côté d'une abstraction plus importante : la notion d'un déplacement *move* entre deux étages. Si cette phrase se trouve dans le document d'exigences, vous aurez très certainement besoin d'une classe *MOVE*, qui pourrait être de la forme :

```
class MOVE feature
      initial, final: FLOOR;          -- Ou INTEGER, s'il n'y a pas de classe FLOOR
      record (d: DATABASE) is ...
      ... Autres caractéristiques ...
end -- class MOVE
```

Il s'agit là d'une classe importante, qu'une méthode fondée sur la grammaire éluderait du fait de l'énoncé. En revanche, si la phrase était écrite sous la forme :

> Un enregistrement de la base de données doit être créé pour chaque déplacement de l'ascenseur d'un étage à un autre.

Alors "déplacement" serait comptabilisé comme nom et conduirait donc à une classe ! Nous voyons, une fois de plus, les dangers auxquels conduit une confiance trop marquée dans le document en langue naturelle, et l'absurdité qui consiste à rendre dépendante de telles variations de style et de sentiment toute propriété importante d'une conception d'un système.

Système à écrans multiples : chapitre 20. Défaire-refaire : chapitre 21.

La deuxième raison qui conduit à oublier des classes est que certaines abstractions cruciales peuvent ne pas être déduites directement des exigences. Ces cas sont fréquents dans ce livre. Les exigences d'un système à écrans multiples ne mentionnent pas forcément de façon explicite les notions d'état et d'application ; ce sont pourtant des abstractions clés, qui conditionnent la conception dans son ensemble. On a signalé précédemment que certains types d'objets externes peuvent ne pas avoir d'équivalent parmi les classes du logiciel ; nous voyons ici l'inverse : des classes du logiciel qui ne correspondent à aucun objet du monde externe. De même, si l'auteur des exigences d'un éditeur de texte offrant un mécanisme défaire-refaire a écrit "*le système doit*

permettre l'insertion et l'effacement de ligne", nous avons de la chance, car nous pouvons distinguer les noms *insertion* et *effacement* ; mais de telles capacités sont tout aussi nécessaires lorsque la phrase se présente ainsi :

> L'éditeur doit permettre à l'utilisateur d'insérer ou d'effacer une ligne à la position courante.

et conduit le concepteur naïf à consacrer son attention aux notions de "curseur" et de "position", et à manquer les abstractions de commande (insertion et effacement de ligne).

La troisième cause majeure d'oubli de classes, commune à toutes les méthodes qui utilisent un document d'exigences comme base d'analyse, est la non-considération de la réutilisation. Il est surprenant de remarquer qu'une grande partie de la littérature d'analyse orientée objet considère comme acquise la vision traditionnelle du développement logiciel : débuter avec un document d'exigences et concevoir une solution au problème spécifique qu'il décrit. Une des leçons importantes de la technologie objet est l'absence d'une distinction claire et précise entre le problème et la solution. Le logiciel existant peut et doit influencer les nouveaux développements.

Voir "LA NATURE CHANGEANTE DE L'ANALYSE", 27.2, page 876.

Lors de la conception d'un nouveau projet logiciel, le développeur orienté objet n'accepte pas le document d'exigences comme la référence absolue, mais le combine à la connaissance tirée des développements précédents et des bibliothèques logicielles disponibles. Si nécessaire, il critiquera le document d'exigences et proposera des mises à jour et des adaptations qui faciliteront la construction du système ; un changement mineur ou l'élimination d'un service d'intérêt limité aux utilisateurs finaux conduiront, parfois, à une nette simplification, en permettant de réutiliser un vaste ensemble de logiciels existants et, en conséquence, de diminuer le temps de développement de plusieurs mois. Les abstractions correspondantes se trouveront, très probablement, dans le logiciel existant, et non dans le document d'exigences d'un nouveau projet.

Les classes `COMMAND` et `HISTORY_LOG` de l'exemple défaire-refaire en sont un bon exemple. Pour trouver les bonnes abstractions de ce problème, il ne suffit pas de lire le document d'exigences d'un éditeur de texte : soit vous tombez dessus via un processus de découverte intellectuelle (un "eurêka", pour lequel aucune recette infaillible n'existe) ; soit, si quelqu'un d'autre a déjà trouvé la solution, vous réutilisez ses abstractions. Il se peut, bien sûr, que vous puissiez également réutiliser l'implémentation correspondante, si elle est disponible dans une bibliothèque ; c'est même mieux, puisque tout le travail d'analyse-conception-implémentation a déjà été fait.

Découverte et rejet

Outre ses enseignements simples, cette étude nous a conduits à envisager d'autres retombées plus subtiles.

Les enseignements simples ont été rencontrés à plusieurs reprises : n'ayez pas trop confiance dans un document d'exigences ; n'ayez *aucune* confiance dans les critères grammaticaux.

L'étude des "fausses alarmes" a mis en lumière que nous avons autant besoin de critères pour trouver des classes que pour en **rejeter** — des concepts qui semblaient, de prime abord, intéressants, mais qui se sont avérés indignes de correspondre à une classe. Les études de conception de ce livre fournissent de nombreux cas de ce genre.

> Pour ne citer qu'un exemple : une étude prochaine concernant la meilleure manière de fournir un générateur de nombres pseudo-aléatoires commence, naturellement, par considérer la notion de nombre aléatoire pour, finalement, la rejeter, car elle ne correspond pas à l'abstraction de données appropriée.

"Générateurs de nombres pseudo-aléatoires : un exercice de conception", page 730.

Les livres d'analyse et de conception OO que j'ai lus évoquent peu cette tâche. C'est surprenant, car, de par ma pratique de conseil en projets OO, j'ai remarqué, en particulier avec des équipes relativement novices, qu'éliminer les mauvaises idées est tout aussi important que de trouver les bonnes.

C'est même peut-être plus important. Joignez-vous à un groupe d'utilisateurs, de développeurs et de dirigeants tentant de s'initier à la technologie objet à l'occasion d'un nouveau projet, et avec un enthousiasme plus neuf encore. Les idées de classes ne manqueront pas (typiquement présentées comme des "objets"). Le problème consiste à maîtriser ce torrent avant qu'il n'inonde le projet. Bien que certaines idées de classes n'aient probablement pas été entrevues, bien d'autres auront été examinées et rejetées. Comme dans toute enquête criminelle d'envergure, beaucoup d'indices se présentent, spontanément ou non ; il faut trier celles qui sont utiles et celles qui ne mènent à rien.

Il nous faut donc adapter et étendre la question qui a servi de titre à ce chapitre. "Comment trouver les classes" a deux sens : non seulement comment proposer des classes candidates, mais également détecter celles qui ne sont pas adaptées. Ces deux tâches ne sont pas exécutées séquentiellement ; au contraire, elles sont constamment entrelacées. Comme tout jardinier, le concepteur orienté objet doit, à tout moment, soigner les bonnes plantes et arracher les mauvaises :

Principe de sélection de classes

La sélection de classes est un processus double : suggestion de classes, rejet de classes.

Le reste de ce chapitre étudie les deux composantes du processus de sélection de classes.

22.2 LES SIGNAUX DE DANGER

Il est préférable, pour guider notre recherche, de commencer par la partie traitant des rejets. Elle nous fournira une liste de pièges typiques, nous fera toucher du doigt les critères les plus importants et nous aidera à concentrer notre recherche des bonnes classes sur les efforts les plus productifs.

"Une typologie des règles", page 648.

Passons en revue quelques-uns des signes révélant, habituellement, un mauvais choix de classe. Puisque la conception n'est pas une discipline complètement formalisée, vous ne devriez pas considérer ces signes comme des preuves d'une *mauvaise* conception ; dans chaque cas, on peut trouver des situations dans lesquelles la conception d'origine est légitime. Ce que nous verrons donc, ce ne sont pas, selon les termes d'un chapitre précédent, des "négations absolues" (des règles impératives de rejet d'une conception), mais des "conseils de négation" : des signaux de danger qui vous préviennent de la présence d'un schéma suspect, et qui devraient vous inciter à l'analyser davantage. Bien qu'ils doivent vous conduire, dans la plupart des cas, à réviser la conception, il se peut que vous décidiez, occasionnellement, qu'au fond tout est bien ainsi.

La faute majeure

Une grande partie des signaux de danger évoqués ci-dessous sont révélateurs de la faute la plus courante et la plus néfaste : concevoir une classe qui n'en est pas une.

Le principe de construction de logiciel orienté objet est d'élaborer des modules selon les types des objets, et non selon des fonctions. C'est essentiel aux gains de réutilisabilité et d'extensibilité qu'apporte cette approche. Mais les débutants tomberont souvent dans le piège le plus évident : appeler "classe" ce qui, en fait, est une routine. Écrire un module sous la forme **`class`**... **`feature`** ... **`end`** n'en fait pas une vraie classe ; il peut ne s'agir que d'une routine déguisée.

Cette faute majeure est facile à éviter une fois que vous êtes conscient du risque. Le remède est toujours le même : s'assurer que chaque classe correspond à une abstraction de données pertinente.

Ce qui suit est un ensemble d'indices vous avertissant qu'un module se présentant comme classe candidate, et ayant les atours d'une classe, peut n'être qu'un immigrant clandestin ne méritant pas la citoyenneté de la société OO des modules.

Ma classe effectue…

Lors d'une réunion de conception, d'une revue d'architecture ou d'une discussion informelle avec un développeur, vous demandez le rôle d'une classe donnée. La réponse : "*Cette classe imprime les résultats*" ou "*cette classe analyse la syntaxe de l'entrée*", ou une autre variante de "*Cette classe fait…*".

La réponse indique souvent un défaut de conception. Une classe n'est pas supposée *faire*, mais offrir un certain nombre de services (caractéristiques) sur des objets d'un certain type. Si elle ne fait qu'une chose, il s'agit probablement d'un cas de cette faute majeure : concevoir une classe pour ce qui devrait être une simple routine d'une autre classe.

La faute peut ne pas se trouver dans la classe elle-même, mais dans la manière dont elle est décrite, qui utilise une phraséologie trop opérationnelle. De toute façon, il vaut mieux vérifier.

Documentation de NeXT pour Open Step, version 4.0 beta.

> Au cours de ces dernières années, le style "*ma classe fait…*" s'est généralisé. Un document de NeXT décrit des classes de la façon suivante : "*La classe* `NSTextView` *déclare l'interface de programmation des objets affichant un texte mis en page…*" ; "*Un* `NSLayoutManager` *coordonne la mise en page et l'affichage des caractères…*" ; "`NSTextStorage` *est une sous-classe mi-concrète de* `NSMutableAttributedString` *gérant un ensemble de clients* `NSLayoutManagers`*, les prévenant de tout changement…*". Même si (comme c'est très probablement le cas ici) les classes évoquées représentent des abstractions de données valides, il serait préférable de les décrire de manière moins opérationnelle en mettant l'accent sur ces abstractions.

Noms impératifs

Supposez que, lors d'un essai de conception, vous trouviez un nom de classe comme `PARSE` ou `PRINT` — un verbe sous forme impérative ou infinitive. Cela devrait attirer votre attention, signalant à nouveau un cas probable de classe qui "fait une chose", et qui ne devrait pas être une classe.

Il arrive parfois que cette classe soit correcte. C'est alors son nom qui ne l'est pas. C'est une règle "affirmative positive".

Bien que, comme toute autre règle faisant référence au style, celle-ci soit en partie une question de convention, elle renforce l'idée que toute classe représente une abstraction de données.

Règle de nom de classe

Un nom de classe doit être :

- un nom, éventuellement qualifié ;
- (pour les seules classes retardées décrivant une propriété structurelle) un adjectif.

La forme nom recouvre la grande majorité des cas. Un nom peut être utilisé seul, comme dans `TREE`, ou qualifié par un adjectif, comme dans `LINKED_LIST`, ou par un autre nom, comme dans `LINE_DELETION`.

La forme adjectif ne se présente que dans un cas spécifique : les classes décrivant une propriété structurelle abstraite, comme la classe bibliothèque `COMPARABLE` qui regroupe des objets munis d'une certaine relation d'ordre. De telles classes devraient être retardées ; leurs noms (en anglais et français) se termineront souvent en `ABLE`. Elles ont vocation à être utilisées via l'héritage pour indiquer que toutes les instances d'une classe ont une certaine propriété ; par exemple, dans un système qui gère les rangs de joueurs de tennis, la classe `PLAYER` peut hériter de `COMPARABLE`. Dans la taxonomie des genres d'héritage, ce schéma sera classé sous la rubrique *héritage de structure*.

"Héritage de structure", page 802.

Le seul cas qui peut faire penser à une exception à la règle est celui des classes de commande, comme celles introduites dans le schéma de conception défaire-refaire, qui prennent en compte les abstractions d'actions. Mais, même là, vous devriez vous limiter à la règle : appeler des classes de commande d'un éditeur de texte `LINE_DELETION` et `WORD_CHANGE`, et non `DELETE_LINE` ou `REPLACE_WORD`.

Voir chapitre 21.

Contrairement à bon nombre d'autres langages, l'anglais offre plus de flexibilité dans l'application de cette règle, puisque ses catégories grammaticales sont plus souples, les verbes pouvant presque toujours jouer le rôle d'un nom. Si vous écrivez votre logiciel en anglais, vous pourrez utiliser des noms simples et courts : vous pouvez ainsi appeler une classe `IMPORT`, alors que, dans d'autres langages, l'équivalent n'étant qu'un verbe, il faudra utiliser un nom, par exemple `IMPORTATION`. Mais ne trichez pas : la classe `IMPORT` doit correspondre à l'abstraction "objets importés" (nominal) et non, sauf pour une classe de commande, à l'action consistant à importer (verbal).

> Il est intéressant de mettre en rapport la règle de nom de classe avec la discussion sur le conseil "souligner les noms" du début de ce chapitre. "Souligner les noms" consistait à appliquer un critère grammatical formel à un texte en langue naturelle informel ; cela ne peut être que de valeur douteuse. La règle de nom de classe, d'un autre côté, applique le même critère à un texte *formel* — le logiciel.

Classes à routine unique

Un symptôme typique de cette faute majeure de désignation est la présence d'une classe effective ne contenant qu'une routine exportée, appelant éventuellement quelques routines non exportées. La classe est probablement une sous-routine magnifiée — une unité d'une décomposition fonctionnelle et non orientée objet.

Une exception possible se présente pour les objets qui représentent, légitimement, des actions abstraites, par exemple une commande d'un système interactif, et ce qui serait représenté, dans une approche non OO, par une routine passée comme argument d'une autre routine. Mais les exemples présentés dans une étude antérieure montrent suffisamment clairement que, même dans ce cas, il y aura d'ordinaire plusieurs caractéristiques applicables. Nous avons remarqué qu'un objet d'un logiciel mathématique représentant une fonction à intégrer n'aura pas pour seule caractéristique

Voir "Petites classes", page 693.

`item (a: REAL): REAL`, donnant la valeur de la fonction au point `a` : on pourra aussi trouver le domaine de définition, le minimum et le maximum sur un intervalle donné, la dérivée. Même si une classe n'a pas encore toutes ces caractéristiques, vérifier qu'il sera possible de les ajouter par la suite renforcera votre conviction que vous êtes en train de parler d'une authentique abstraction d'objet.

Voir "TAXOMANIE", 24.4, page 791.

En appliquant la règle de la routine unique, vous devriez considérer toutes les caractéristiques d'une classe : celles introduites dans la classe même, ainsi que celles héritées de ses parents. Un texte de classe ne déclarant qu'une routine exportée n'est pas nécessairement une erreur, s'il s'agit d'un simple ajout à une abstraction valide définie par ses ancêtres. Cela peut, cependant, signaler un cas de *taxomanie*, une maladie liée à l'héritage qui sera étudiée dans la partie consacrée à la méthodologie de l'héritage.

Classification prématurée

La mention de la taxomanie fait penser à une autre faute fréquente commise par les novices : commencer à songer à la hiérarchie d'héritage trop en amont dans le processus de conception.

Si l'héritage est au centre de la méthode orientée objet, une bonne structure d'héritage — plus précisément, une bonne structure modulaire contenant à la fois relation d'héritage et relation client — est essentielle à la qualité d'une conception. Mais l'héritage n'a de sens qu'en tant que relation entre abstractions bien comprises. Tant que nous en sommes encore à chercher des abstractions, il est trop tôt pour concevoir la hiérarchie d'héritage.

La seule exception manifeste se présente quand vous devez traiter un domaine d'application pour lequel préexiste une taxonomie largement acceptée, comme dans certaines branches des sciences. Les abstractions correspondantes émergeront alors en même temps que la structure d'héritage. (Avant d'accepter cette taxonomie comme base de la structure de votre logiciel, vérifiez qu'elle est effectivement stable et largement reconnue, et non le point de vue d'une seule personne.)

Dans les autres cas, vous ne devriez concevoir la hiérarchie d'héritage qu'après avoir eu au moins une première idée des abstractions. (L'effort de classification peut, naturellement, vous conduire à réviser votre choix d'abstractions, initiant un processus itératif dans lequel les tâches de sélection de classes et de conception de structure d'héritage interagissent.) Si, au début du processus de conception, les participants se concentrent trop sur les questions de classification alors que les classes ne sont pas encore bien comprises, ils mettent probablement la charrue avant les boeufs.

Il peut s'agir d'une variante de la confusion entre objet et classe commise par un novice. Certains ont commencé avec les hiérarchies d'héritage du type "`SAN_FRANCISCO` et `HOUSTON` héritent de `CITY`" — pour simplement modéliser une situation dans laquelle une classe unique, `CITY`, avait plusieurs instances au moment de l'exécution.

Les classes sans commande

Vous trouverez parfois une classe n'ayant aucune routine, ou qui ne fournit que des requêtes (des moyens d'accéder à des objets), mais pas de commandes (des procédures pour modifier des objets). Une telle classe est l'équivalent des enregistrements en Pascal ou des structures en Cobol ou C. Il peut s'agir d'une erreur de conception, mais comme elle peut être de deux sortes, vous devrez poursuivre l'analyse.

Examinons d'abord trois cas dans lesquels une telle classe n'est *pas* signe de conception erronée :

- elle peut représenter des objets venant du monde extérieur, et que le logiciel orienté objet ne peut changer. Il pourrait s'agir de données provenant d'un capteur d'un système de contrôle de processus, de paquets d'un réseau à commutation de paquets, ou de structures C que le système OO n'a pas à modifier ;
- certaines classes n'ont pas vocation à être instanciées, mais à encapsuler des services comme des constantes, utilisés par d'autres classes par héritage. Un tel *héritage de services* sera étudié lors du traitement de la méthodologie de l'héritage ;

"HÉRITAGE DE SERVICE", 24.9, page 817.

- enfin, une classe peut être *applicative*, c'est-à-dire décrivant des objets non modifiables ; à la place de commandes permettant de modifier un objet, elle fournira des fonctions pour produire de nouveaux objets, habituellement du même type. Par exemple, l'opération d'addition des classes `INTEGER`, `REAL` et `DOUBLE` suit le chemin indiqué par les mathématiques : elle ne modifie aucune valeur, mais, étant donné deux nombres `x` et `y`, elle en produit un troisième, `x + y`. Dans la spécification du type abstrait de données, de telles fonctions seront caractérisées, comme celles qui fournissent des commandes, par des fonctions de commande.

Les fonctions de commande ont été définies dans "Catégories de fonctions", page 139.

Dans ces cas précis, les abstractions sont faciles à reconnaître, et vous n'aurez donc pas de difficulté à identifier les deux cas qui peuvent effectivement être signe d'une déficience au niveau de la conception.

Passons maintenant aux cas suspects. Dans le premier, la classe est justifiée et aurait besoin de commandes ; le concepteur a simplement oublié de fournir les mécanismes pour modifier les objets correspondants. Une simple technique de check-list présentée lors de l'étude de la conception de classes permettra d'éviter de telles erreurs.

Voir "Une check-list", page 744.

Dans le second cas, plus lié au sujet de cette étude, la classe n'était pas justifiée. Ce n'est pas une vraie abstraction de données, mais simplement un morceau d'information passive qui aurait pu être représenté par une structure comme une liste ou un tableau, ou en ajoutant simplement plus d'attributs à une autre classe. Cela se produit parfois quand les développeurs écrivent une classe pour ce qui n'aurait été qu'un simple type enregistrement (structure) en Pascal, Ada ou C. Tous les types enregistrements ne correspondent pas à des abstractions de données distinctes.

Ce cas mérite une analyse soignée pour déterminer s'il est légitime d'introduire une classe, maintenant ou dans le futur. Si la réponse n'est pas claire, il vaut mieux garder la classe, même si elle risque d'être superflue. Introduire une classe peut conduire à une légère perte de performance si cela conduit à traiter de nombreux objets de petite taille, créés dynamiquement un à un et occupant plus d'espace que s'il s'agissait de simples éléments de tableaux ; mais, si vous avez besoin d'une classe et si vous ne l'avez pas introduite suffisamment tôt, l'adaptation peut vous coûter cher.

> Nous avons connu un tel ratage lors du développement du compilateur d'ISE. Un compilateur pour un langage OO a besoin d'un moyen interne pour identifier chacune des classes du système qu'il est en train de traiter ; cette identificateur était, auparavant, un entier. Cela a fonctionné pendant de nombreuses années, mais, au bout d'un certain temps, nous avons eu besoin d'un schéma d'identification de classe plus élaboré, nous permettant, en particulier, de *renuméroter* les classes lors des fusions de plusieurs systèmes. La solution a consisté à introduire une classe `CLASS_IDENTIFIER`, et à remplacer les entiers précédents par des instances de cette classe. L'effort de conversion s'est avéré plus important que nous l'aurions souhaité. Au départ, `INTEGER` était une abstraction suffisante, puisqu'aucune commande n'était applicable aux identificateurs de classes ; le besoin de

caractéristiques plus sophistiquées, en particulier les commandes de renumérotation, a conduit à l'identification d'une abstraction distincte.

Abstractions mixtes

Une classe qui possède des caractéristiques faisant référence à plusieurs abstractions est également le signe d'une conception imparfaite.

Dans une ancienne version de la bibliothèque de NeXT, la classe de texte fournissait également l'ensemble des moyens d'édition visuelle de texte. Les utilisateurs se sont plaints que la classe, quoique utile, était trop grosse. La taille importante de la classe était le symptôme ; le problème venait, en fait, de la fusion de deux abstractions (les chaînes de caractères, et le texte qui peut être édité interactivement) ; la solution a consisté à séparer les deux abstractions, avec une classe `NSAttributedString` pour le mécanisme de base de manipulation des chaînes, et d'autres comme `NSTextView` pour gérer les aspects liés à l'interface utilisateur.

[Page-Jones 1995].

Meilir Page-Jones utilise le terme de *co-naissance* (*connascence*, défini dans les dictionnaires anglo-saxons comme la propriété d'être né et d'avoir été élevé ensemble) pour décrire la relation qui existe entre deux caractéristiques fortement liées selon un critère de changement simultané : un changement à l'une impliquera un changement à l'autre. Comme le remarque l'auteur, vous devriez minimiser la conaissance entre les bibliothèques de classes ; les caractéristiques qui apparaissent dans une classe donnée devraient toutes être reliées à la même abstraction clairement identifiée.

Cette règle universelle de conduite mérite d'être exprimée par une règle méthodologique (présentée sous une forme "positive", bien qu'elle découle d'une analyse des erreurs possibles) :

Principe de cohérence de classe

Toutes les caractéristiques d'une classe doivent appartenir à une abstraction unique et bien identifiée.

La classe idéale

Cet énoncé des erreurs possibles met en évidence, a contrario, ce à quoi ressemblera une classe idéale. Voici quelques propriétés typiques :

- il y a une abstraction clairement associée, qui peut être décrite comme une abstraction de données (ou une machine abstraite) ;
- le nom de la classe est un nom ou un adjectif caractérisant l'abstraction de manière adéquate.
- la classe représente un ensemble d'objets possibles à l'exécution, ses instances (certaines classes ont vocation à n'avoir qu'une instance lors d'une exécution ; c'est aussi acceptable) ;
- plusieurs requêtes sont disponibles pour déterminer les propriétés d'une instance ;
- plusieurs commandes sont disponibles pour modifier l'état d'une instance (dans certains cas, il n'y a pas de commande, mais, à la place, des fonctions produisant d'autres objets du même type, comme avec les opérations sur les entiers ; c'est acceptable, également) ;
- des propriétés abstraites peuvent être énoncées, informellement ou (de préférence) formellement, décrivant : la façon dont les résultats des diverses requêtes sont liées les unes aux autres (ceci conduira à l'invariant) ; dans quelles conditions les caractéristiques sont

applicables (préconditions) ; la façon dont l'exécution des commandes peut affecter les résultats des requêtes (postconditions).

Cette liste décrit un ensemble d'objectifs informels, et non une règle stricte. Une classe légitime peut n'avoir qu'une partie des propriétés listées ci-dessus. La plupart des exemples qui jouent un rôle important dans ce livre — de *LIST* et *QUEUE* à *BUFFER*, *ACCOUNT*, *COMMAND*, *STATE*, *INTEGER*, *FIGURE*, *POLYGON* et bien d'autres — les possèdent toutes.

22.3 HEURISTIQUES GÉNÉRALES POUR TROUVER LES CLASSES

Passons maintenant à la partie positive de notre discussion : les heuristiques pratiques pour trouver des classes.

Catégories de classes

Tout d'abord, nous pouvons remarquer qu'il y a trois grandes catégories de classes : les classes d'analyse, les classes de conception et les classes d'implémentation. La division n'est ni absolue ni rigoureuse (par exemple, on pourrait trouver des arguments pour ranger une classe retardée *LIST* dans l'une quelconque de ces trois catégories), mais il s'agit là d'une règle de conduite générale pratique.

Une classe d'analyse décrit une abstraction de données directement tirée du modèle du système externe. *PLANE* dans un système de contrôle aérien, *PARAGRAPH* dans un système de traitement de documents, *PART* dans un système de contrôle d'inventaire en sont des exemples typiques.

Une classe d'implémentation décrit une abstraction de données introduite pour les besoins internes des algorithmes du logiciel, comme *LINKED_LIST* ou *ARRAY*.

Entre les deux, une classe de conception décrit un choix architectural. On a, par exemple, utilisé *COMMAND* dans la solution du problème défaire-refaire et *STATE* dans la solution du problème des systèmes à écrans multiples. Comme les classes d'implémentation, les classes de conception appartiennent à l'espace des *solutions*, alors que les classes d'analyse appartiennent à l'espace des *problèmes*. Mais, comme les classes d'analyse et contrairement aux classes d'implémentation, elles décrivent des concepts de haut niveau.

Lors de l'étude concernant la manière d'obtenir des classes dans ces trois catégories, nous découvrirons que les classes de conception sont les plus difficiles à identifier, car elles requièrent le genre d'intime perception architecturale qui distingue le concepteur doué. (Le fait qu'elles soient les plus difficiles à trouver ne veut pas dire qu'elles sont les plus difficiles à *construire*, ce privilège revenant habituellement aux classes d'implémentation, sauf, bien sûr, si vous trouvez une bibliothèque d'implémentation prête à être réutilisée.)

Objets externes : trouver les classes d'analyse

Débutons avec les classes d'analyse, modélisées à partir des objets externes.

Nous utilisons le logiciel pour trouver des réponses à certaines questions sur le monde (comme dans un programme qui calcule la solution d'un problème spécifique), pour interagir avec le

monde (comme dans un système de contrôle de processus) ou pour ajouter des choses au monde (comme dans un traitement de texte). Dans chaque cas, le logiciel doit être fondé sur un modèle des aspects du monde qui concernent l'application, comme les lois de la physique ou de la biologie dans un programme scientifique, la syntaxe et la sémantique d'un langage informatique dans un compilateur, les échelles de salaire dans un système de paie, et les lois fiscales dans un logiciel de traitement des impôts.

Voir "La réalité : un cousin doublement éloigné", page 231.

> Pour parler du monde modélisé, nous devrions éviter d'utiliser le terme "monde réel", qui est trompeur, d'une part parce que le logiciel n'est pas moins "réel" qu'autre chose et, d'autre part, parce que de nombreux "mondes" non logiciels sont artificiels, comme dans le cas d'un programme mathématique traitant d'équations et de graphes. (Nous avons évoqué cette question en détail lors d'un chapitre précédent.) Nous devrions parler du *monde externe*, distinct du monde interne que traite le logiciel.

Tout système logiciel est fondé sur un **modèle opérationnel** d'un aspect particulier du monde externe. Opérationnel, car il est utilisé pour générer des résultats pratiques et, parfois, pour réintroduire ces résultats dans le monde ; modèle parce que tout système utile doit découler d'une certaine interprétation d'un phénomène du monde.

Voir "SIMULA", 35.1, page 1081.

Le domaine de la *simulation* est, peut-être, l'endroit où cette vision du logiciel est la plus pertinente. Ce n'est pas un hasard si le premier langage orienté objet, Simula 67, est une évolution de Simula 1, un langage permettant d'écrire des simulations par événements discrets. Bien que Simula 67 soit, lui, un langage de programmation d'intérêt général, il a gardé le nom de son prédécesseur et contient un ensemble de puissantes primitives de simulation. Jusqu'au milieu des années soixante-dix, la simulation est restée le principal domaine d'applications de la technologie objet (comme suffit à le montrer la lecture des actes des conférences annuelles de l'Association des utilisateurs de Simula). Cette tendance naturelle à appliquer les concepts OO pour la simulation est facile à comprendre : pour concevoir la structure d'un système logiciel simulant le comportement d'un ensemble d'objets externes, qu'y a-t-il de mieux que d'utiliser des composants logiciels qui représentent directement ces objets ?

D'une manière générale, bien sûr, tout logiciel est une simulation. S'appuyant sur cette vision du logiciel comme modélisation opérationnelle, la construction logicielle orientée objet utilise, comme premières abstractions, certains types déduits de l'analyse des principaux types d'objets, au sens non logiciel du terme, du monde externe : capteurs, périphériques, avions, employés, feuilles de paie, impôts, paragraphes, fonctions intégrables.

> En fait, ces exemples ne présentent qu'une partie du sujet. Comme le notent Waldén et Nerson dans leur présentation de la méthode B.O.N. :
>
> Une classe représentant une voiture n'est pas plus tangible que celle qui modélise la satisfaction professionnelle des employés. Ce qui compte, c'est l'importance de ces concepts pour l'entreprise, et ce que vous pouvez en faire.
>
> Gardez ce commentaire à l'esprit quand vous cherchez les classes externes : elles peuvent être très abstraites. Les `REGLE_DE_L_AGE` dans un système de vote parlementaire et `TENDANCE_DU_MARCHÉ` dans un système de courtage peuvent être tout aussi réelles que `SENATEUR` et `BOURSE`. Le sourire du chat de Cheshire est tout autant un objet que le chat lui-même.

[Waldén 1995], *pages 182-183.*

Qu'elles soient matérielles ou abstraites, les classes externes représentent les abstractions que les spécialistes du monde extérieur, qu'ils soient ingénieurs aéronautiques, comptables ou mathématiciens, utilisent constamment pour penser et présenter leur domaine. Il est toujours fortement probable — bien que ce ne soit pas une certitude — qu'un tel type d'objets conduira toujours à une classe utile, car, en règle générale, les experts du domaine y associeront des opérations et des propriétés significatives.

Le mot-clé, comme toujours, est *abstraction*. Bien qu'il soit préférable que les classes d'analyse se rapprochent autant que possible des concepts du domaine du problème, ce n'est pas cela qui rend forcément intéressante une classe candidate. La première version de notre système à écrans multiples en a montré la raison de manière saisissante : nous étions en présence d'un modèle directement inspiré de certaines propriétés du système externe, mais qui s'est avéré terriblement inadapté d'un point de vue génie logiciel, car les propriétés choisies étaient de bas niveau et susceptibles d'être modifiées. Une bonne classe externe sera fondée sur des concepts abstraits du domaine du problème, présentant (selon l'école ADT) des caractéristiques externes choisies pour leur valeur durable.

Pour un développeur orienté objet, de telles abstractions préexistantes sont précieuses : elles fournissent certaines classes fondamentales du système et, comme nous pouvons le remarquer une fois de plus, les objets sont prêts à être cueillis.

Trouver les classes d'implémentation

Les classes d'implémentation décrivent les structures que les développeurs logiciels utilisent pour faire exécuter leurs systèmes par un ordinateur. Bien que la mode, dans la littérature du génie logiciel, ait consisté, dans les quinze dernières années, à minimiser le rôle de l'implémentation, les développeurs connaissent le truisme selon lequel l'implémentation occupe une large part de l'effort de construction d'un système, et requiert une bonne partie de l'intelligence qu'exige celle-ci.

La mauvaise nouvelle est qu'implémenter est difficile. La bonne est que les classes d'implémentation, bien que souvent difficiles à *construire* en l'absence de bonnes bibliothèques réutilisables, ne sont pas les plus difficiles à *sélectionner*, grâce à l'abondance de la littérature sur le sujet. Puisque "Structures de données et algorithmes", parfois appelé (NdT : aux États-Unis) "CS 2", est un composant obligatoire de tout enseignement en informatique, de nombreux livres d'étude évoquent le large catalogue des structures de données utiles et identifiées au cours des années. Encore mieux, bien que la majorité des livres existants n'utilisent pas explicitement une approche orientée objet, nombreux sont ceux qui adoptent un style inspiré des types abstraits de données, même s'ils n'utilisent pas cette terminologie, pour présenter les structures de données ; par exemple, pour introduire les différentes formes de tables comme les arbres binaires de recherche et les tables de hachage, vous devez d'abord indiquer les diverses opérations (insérer un élément avec sa clé, chercher un élément de clé donnée, et ainsi de suite) et leurs propriétés. La transition aux classes est alors relativement facile.

Récemment, certains livres d'étude ont commencé à aller plus loin, en appliquant tout au long une approche orientée objet aux sujets traditionnels abordés dans CS 2.

Qu'il ait ou non suivi un cours de structures de données et d'algorithmes à l'école, tout ingénieur logiciel devrait garder à portée de main un bon livre sur le sujet et s'y référer souvent. Il est bien trop facile de perdre du temps à inventer à nouveau des concepts bien connus, à implémenter un algorithme sous-optimal ou à choisir une représentation qui n'est pas appropriée à l'utilisation logicielle d'une structure de données — par exemple, une liste simplement chaînée pour une structure séquentielle, alors que les algorithmes doivent régulièrement traverser celle-ci d'avant en arrière, ou un tableau pour une structure qui croît et s'amenuise de manière imprévisible. Remarquez que règne, ici également, l'approche ADT : la structure de données et sa représentation découlent du service offert aux clients.

Outre les livres et l'expérience, le meilleur espoir pour les classes d'implémentation se trouve dans les bibliothèques réutilisables, comme nous le verrons à la fin de ce chapitre.

Classes retardées d'implémentation

Les livres traitant des structures de données traditionnelles mettent naturellement l'accent sur les classes effectives (complètement implémentées). En pratique, l'essentiel de la valeur d'un ensemble de classes d'implémentation réside, en particulier si elles ont vocation à être réutilisables, dans la taxonomie sous-jacente, telle qu'elle apparaît dans une structure d'héritage contenant des classes retardées. Par exemple, diverses implémentations de files seront des descendants d'une classe retardée `QUEUE` décrivant le concept abstrait de liste séquentielle.

"Une classe retardée d'implémentation" n'est donc pas une contradiction. Les classes comme `QUEUE`, bien qu'abstraites, facilitent la construction des taxonomies grâce auxquelles nous pouvons préserver la cohérence et l'organisation des différentes variétés de structures d'implémentation, en affectant à chaque classe une place précise dans le plan d'ensemble.

Dans un autre livre [M 1994a], j'ai décrit une taxonomie "à la Linné" des structures fondamentales de l'informatique, qui utilise des classes retardées pour classifier les genres des principales structures de données utilisées dans le développement logiciel.

Trouver les classes de conception

À propos des itérateurs et de MVC, voir les notes bibliographiques.

Les classes de conception représentent les abstractions architecturales qui facilitent la production de structures logicielles élégantes et extensibles. `STATE`, `APPLICATION`, `COMMAND`, `HISTORY_LIST`, les classes d'itérateurs, les classes "*controller*" dans le modèle MVC de Smalltalk sont des bons exemples de classes de conception. Nous verrons d'autres idées clés dans les prochains chapitres, comme les structures de données actives et les handles pour les bibliothèques portables et adaptables aux plates-formes.

Bien que, comme on l'a vu, il n'existe pas de moyen infaillible pour trouver des classes de conception, il peut être utile de noter quelques règles :

[M 1993].

- de nombreuses classes de conception ont déjà été conçues par d'autres. En lisant des livres et des articles décrivant des solutions précises à des problèmes de conception, vous glanerez de nombreuses idées utiles. Par exemple, le livre *Object-Oriented Applications* contient des chapitres écrits par des concepteurs de divers projets industriels, fournissant un guide précieux à ceux qui doivent traiter des problèmes similaires en télécommunication, conception assistée par ordinateur, intelligence artificielle et autres domaines d'application ;

[Gamma 1995].

- le livre sur les "schémas de conception" (*design patterns*) de Gamma *et al.* a initié un projet tendant à capturer les solutions de conception éprouvées, et est maintenant suivi par d'autres ;
- de nombreuses classes de conception utiles décrivent des abstractions qu'il est plus facile de considérer comme des machines plutôt que des "objets" dans le sens usuel (non logiciel) du terme ;
- comme pour les classes d'implémentation, la réutilisation est préférable à l'invention. On peut espérer que, parmi les nombreux "schémas" en cours d'étude, plusieurs d'entre eux cesseront bientôt d'être de simples idées pour devenir, à la place, des classes bibliothèques directement utilisables.

22.4 AUTRES SOURCES DE CLASSES

Un certain nombre d'heuristiques se sont avérées utiles dans cette recherche des bonnes abstractions.

Développement précédents

Le conseil de regarder d'abord ce qui est disponible ne s'applique pas uniquement aux classes bibliothèques. Au fur et à mesure que vous écrirez des applications, vous accumulerez des classes qui, si elles sont conçues correctement, devraient faciliter les développements ultérieurs.

Tout logiciel réutilisable n'est pas forcément réutilisable d'origine. Souvent, une première version d'une classe est produite pour répondre à une exigence immédiate plutôt que pour la postérité. Si on se soucie de réutilisabilité, il est souvent profitable de consacrer un certain temps, après le développement, à rendre une classe plus générale et plus robuste, en améliorant sa documentation, en ajoutant des assertions. Ce type de construction de logiciels, bien que différent de la construction de logiciels réutilisables dès le début, n'en est pas moins fructueux. En effet, étant issues de composants de systèmes réels, les classes résultantes ont passé le premier test de réutilisabilité, c'est-à-dire l'*utilisabilité* : elles ont au moins une utilisation réelle.

Voir "GÉNÉRALISATION", 28.5, page 898.

Adaptation par héritage

En découvrant l'existence d'une classe potentiellement utile, vous vous rendrez parfois compte qu'elle ne correspond pas exactement à votre besoin actuel : une certaine adaptation peut être nécessaire.

Sauf si l'adaptation corrige un défaut qui devrait être corrigé également dans la classe d'origine, il est généralement préférable de laisser la classe telle quelle, en préservant ses clients selon le principe ouvert-fermé. Vous pouvez, à la place, utiliser l'héritage et la redéfinition pour façonner la classe selon votre nouveau besoin.

Cette technique, que notre future taxonomie des utilisations de l'héritage abordera en détail sous le nom d'*héritage de variation*, fait l'hypothèse que la nouvelle classe décrit une variante de la même abstraction que celle d'origine. Si elle est utilisée à bon escient (selon les règles de conduite de l'étude prochaine), c'est l'une des contributions les plus remarquables de la méthode, vous permettant de résoudre le dilemme *réutiliser-refaire* : combiner réutilisabilité et extensibilité.

Voir "Héritage de variation", page 799.

Évaluer les décompositions candidates

On dit que la critique est plus aisée que l'art ; une bonne manière d'apprendre la conception consiste à analyser les conceptions existantes. En particulier, quand un certain ensemble de classes a été proposé pour résoudre un certain problème, vous devriez l'étudier en fonction des critères et principes de modularité donnés dans le chapitre 3 : constituent-elles des modules cohérents et autonomes, ayant des canaux de communication fortement contrôlés ? Souvent, découvrir que deux modules sont trop fortement couplés, qu'un module communique avec un trop grand nombre de modules, qu'une liste d'arguments est trop longue, est caractéristique d'erreurs de conception, et conduit à une meilleure solution.

Chapitre 20.

Un critère important a été exploré dans l'exemple du système à écrans multiples : le flot de données. Nous avons vu combien il est important d'étudier, dans une structure de classe candidate, le flot d'objets passés comme arguments des appels successifs. Si, comme avec la notion d'état de cet exemple, vous détectez qu'un certain élément d'information est transmis à plusieurs modules, c'est très certainement signe que vous avez omis une importante abstraction de données. Une telle analyse, qui a été appliquée pour obtenir la classe `STATE`, est une source importante d'abstractions.

Il est, bien sûr, préférable de trouver les classes dès le début ; mais, mieux vaut tard que jamais. Après une telle découverte a posteriori, vous devriez prendre du recul pour analyser pourquoi cette abstraction avait été initialement laissée de côté, et pour réfléchir sur la manière de faire mieux la prochaine fois.

Suggestions tirées des autres approches

L'exemple de l'analyse du flot des données dans une structure descendante illustre l'idée générale consistant à dériver des classes à l'aide de concepts provenant de décompositions non OO. Cela sera utile dans deux cas non mutuellement exclusifs :

- un système logiciel non OO qui remplit une partie de la tâche existe peut-être déjà ; il peut être intéressant de l'examiner pour trouver des idées de classes. La même chose s'applique si, à la place d'un système existant, vous pouvez utiliser le résultat d'une analyse ou d'une conception produite avec une autre méthode plus ancienne ;
- certaines personnes chargées du développement peuvent avoir une longue expérience des autres méthodes et, en conséquence, peuvent penser en termes de concepts différents, dont certains peuvent être transformés en classes.

Voici des exemples de ce processus, débutant avec des langages de programmation et se poursuivant avec des techniques d'analyse et de conception.

À propos des blocs communs poubelles, voir "Petites interfaces", page 50.

Les programmes Fortran contiennent souvent un ou plusieurs *blocs communs* — des zones de données partagées par plusieurs routines. Un bloc commun cache souvent une ou plusieurs abstractions utiles. Plus précisément, les bons programmeurs Fortran savent qu'un bloc commun ne devrait contenir que quelques variables ou tableaux correspondant à des concepts fortement liés ; il y a de fortes chances pour qu'un tel bloc corresponde à une classe. Malheureusement, ce n'est pas une pratique universelle, et même les programmeurs qui savent qu'il vaut mieux ne pas utiliser le "bloc commun poubelle" mentionné au début de ce livre ont tendance à mettre trop de choses dans un bloc commun. Dans ce cas, il vous faudra examiner les schémas d'utilisation de chaque bloc pour découvrir la ou les abstractions auxquelles il correspond.

Les programmes Pascal et C utilisent des enregistrements, appelés structures en C. (Pascal n'a que des *types* enregistrements ; en C, vous pouvez aussi bien avoir des types structures que des structures individuelles.) Un type enregistrement correspond souvent à une classe, mais seulement si vous pouvez trouver les opérations agissant spécifiquement sur les instances du type, comprenant souvent (comme nous l'avons vu) des commandes et des requêtes. Sinon, le type peut ne représenter que certains attributs d'une autre classe.

Cobol offre également des structures, et sa Data Division facilite l'identification des types de données importants.

Dans la modélisation entité-relation (ER), les analystes isolent des “entités”, qui peuvent souvent être la source de classes.

> Les personnes ayant une longue pratique de la modélisation ER ont souvent du mal à appliquer efficacement les idées orientées objet, car elles ont l’habitude de traiter entités et relations comme étant de nature différente, et considèrent le comportement “dynamique” du système comme complètement séparé d’eux. Avec la modélisation OO, relations et comportement conduisent à des caractéristiques attachées aux types des objets (entités) ; penser les relations et opérations comme des variantes de la même notion et les attacher aux entités n’est parfois pas chose facile au début.

Dans la conception par flot de données (“analyse et conception structurées”), peu de chose peut être directement utilisé pour une décomposition orientée objet, mais, parfois, les “mémoires de stockage” (abstractions de bases de données ou de fichiers) peuvent suggérer une abstraction.

Fichiers

Le commentaire à propos des mémoires de stockage suggère une idée plus générale, utile si vous venez d’un environnement non OO. Parfois, l’essentiel de l’intelligence d’un système traditionnel réside hors du texte du logiciel, dans la structure des fichiers qu’il manipule.

Toute personne ayant l’expérience d’Unix sait que l’information essentielle réside non dans la description des commandes spécifiques mais dans certains fichiers clés et leur format : `passwd` pour les mots de passe, `printcap` pour les propriétés des imprimantes, `termcap` ou `terminfo` pour les propriétés des terminaux. On pourrait caractériser ces fichiers comme des abstractions de données sans abstraction : bien que documentés à un niveau très concret (“*Chaque entrée du fichier* `printcap` *décrit une imprimante, et est une ligne formée d’un certain nombre de champs séparés par des caractères* :. *La première entrée de chaque imprimante donne les noms sous lesquels est connue l’imprimante, séparés par des caractères* |”, etc.), ils décrivent des types de données importants et accessibles via des primitives bien définies, munis de propriétés associées et de conditions d’utilisation. Dans la transition vers une vision orientée objet, de tels fichiers joueraient un rôle central.

Une observation similaire s’applique à de nombreux programmes, dont les fichiers principaux caractérisent certaines de leurs principales abstractions.

J’ai participé, un jour, à une session de conseil avec le responsable d’un système logiciel qui était convaincu que le système — un ensemble de programmes Fortran — ne pourrait se prêter à une décomposition orientée objet. Lors de la description de ce que faisaient les programmes, il a incidemment mentionné certains fichiers par lesquels communiquaient les programmes. J’ai commencé à poser des questions sur ces fichiers, mais il a d’abord persisté à les ignorer, les considérant comme sans importance, et à retourner aux programmes. J’ai insisté et réalisé, grâce à ses explications, que les fichiers décrivaient des structures complexes de données qui codaient l’information essentielle des programmes. La leçon était claire : dès que la pertinence de ces fichiers a été reconnue, ils ont acquis une place privilégiée dans l’architecture orientée objet ; dans un retournement caractéristique des réorganisations orientées objet, les programmes, qui étaient auparavant les éléments clés de l’architecture, sont devenus de simples caractéristiques des classes résultantes.

Cas d'utilisation

Ivar Jacobson a conseillé de s'appuyer sur des cas d'utilisation pour sélectionner les classes. Un cas d'utilisation, appelé *scénario* par d'autres auteurs en analyse et conception (et *trace* en informatique théorique, en particulier dans l'étude de la concurrence), est une description contenant :

[Jacobson 1992], *page 154. Jacobson utilise le terme "acteur" pour désigner les utilisateurs du futur système.*

une collection complète des événements initiés par un [utilisateur du futur système] *et* [de] *l'interaction entre* [l'utilisateur] *et le système*.

Dans un système de commutation téléphonique, par exemple, le cas d'utilisation "appel initié par un client" contient la séquence des événements suivants : le client soulève le combiné, une identification est envoyée au système, le système envoie un signal de numérotation, et ainsi de suite. D'autres cas d'utilisation du système pourraient décrire l'"installation du service d'identification d'appelant" et la "déconnexion du client".

Les cas d'utilisation ne sont pas un bon outil pour trouver des classes. Compter dessus de façon significative comporte de nombreux risques :

Voir "Ordonnancement et développement OO", page 115 et "Structure et ordre : le développeur logiciel est un incendiaire", page 204.

- les cas d'utilisation mettent l'accent sur l'ordre. ("*Quand un client passe une commande par téléphone, son numéro de carte de crédit est validé. La base de données est alors mise à jour et un numéro de confirmation est généré*", etc.). C'est incompatible avec la technologie objet : cette méthode évite de s'appuyer trop tôt sur les propriétés de séquencement, car elles sont fragiles et sujettes à modification. L'analyste et concepteur OO compétent refuse de se concentrer sur les propriétés de la forme "Le système fait `a`, puis `b`" ; à la place, il pose la question "Quelles sont les opérations disponibles sur les instances de l'abstraction `A` et les contraintes sur ces opérations ?". Les propriétés fondamentales de séquencement émergeront sous la forme de contraintes de haut niveau sur les opérations ; par exemple, au lieu de dire qu'une pile peut être soumise à des séquences d'opérations alternant `push` et `pop` tant qu'il n'y a pas plus de `pop` que de `push`, nous définissons les préconditions attachées à chacune des ces opérations, ce qui implique une propriété d'ordre, mais plus abstraite. Des exigences moins fondamentales d'ordonnancement n'ont tout simplement pas leur place dans le modèle d'analyse, car elles détruisent l'adaptabilité du système et, en conséquence, sa survie. Porter prématurément intérêt aux relations d'ordre est l'une des pires erreurs qui puisse être commise dans un projet OO. Cependant, si votre analyse repose sur des cas d'utilisation, cette erreur est difficile à éviter ;
- se baser sur un scénario indique qu'on se concentre sur la manière dont les utilisateurs voient le fonctionnement du système. Mais le système n'existe pas encore. (Un système précédent peut exister, mais s'il était complètement satisfaisant, on ne vous aurait pas demandé de le modifier ou de le récrire.) Ainsi, la vision du système à laquelle conduisent les cas d'utilisation est fondée sur des processus existants, informatisés ou non. Votre tâche de constructeur de systèmes est d'arriver à des scénarios *neufs* et meilleurs, et non de perpétuer d'anciens modes de fonctionnement. Il y a suffisamment d'exemples de systèmes informatiques qui simulent des procédures obsolètes ;
- les cas d'utilisation favorisent une approche fonctionnelle fondée sur des processus (actions). Cette approche est à l'opposé de la décomposition OO, qui se concentre sur les abstractions de données ; elle court le risque sérieux de retourner, sous la bannière d'un développement orienté objet, aux formes les plus traditionnelles de la conception fonctionnelle. S'il est vrai que vous pouvez compter sur plusieurs scénarios plutôt que sur un unique programme principal, cette approche continue de considérer comme point de départ *ce que fait le système*, alors que la technologie objet s'attache à *ce qu'il fait à qui*. Le conflit est inévitable.

Les conséquences pratiques sont évidentes. Un certain nombre d'équipes ayant adopté les cas d'utilisation se trouvent, sans s'en rendre compte, en train de pratiquer une conception fonctionnelle descendante ("*le système doit faire* `a`, *puis* `b`, …") et de construire des systèmes qui seront obsolètes le jour où ils seront livrés, tout en étant difficiles à modifier, car ils sont liés à une vision spécifique de ce que fait le système. J'ai assisté, en tant que consultant extérieur, à des revues de conception de tels projets et essayé de promouvoir plus d'abstraction. Mais la tâche est difficile, car les concepteurs sont convaincus de pratiquer une conception orientée objet ; ils attendent du consultant quelques suggestions, quelques critiques de détail et sa bénédiction pour l'ensemble du résultat. Les conceptions que j'ai vues n'étaient pas orientées objet du tout et ne pouvaient que conduire à des systèmes mal conçus ; mais, il n'y a pire sourd que celui qui ne veut pas entendre — nous travaillons sur des cas d'utilisation, et tout le monde ne sait-il pas que les cas d'utilisation sont OO ?

Les risques sont peut-être moins élevés si l'équipe de conception orientée objet est très expérimentée — la même équipe qui a élaboré avec succès des systèmes OO de taille importante, contenant des milliers de classes et des centaines de milliers de lignes. Un tel groupe peut considérer les cas d'utilisation comme un complément utile aux autres techniques d'analyse. Mais, pour une équipe débutante ou n'ayant qu'une expérience limitée, les avantages d'analyse qu'apportent les cas d'utilisation ne sont pas aussi sûrs : le risque de détruire la qualité du futur système est si grand qu'il est recommandé d'éviter une telle technique :

Principe du cas d'utilisation

Sauf dans le cas d'une équipe de conception très expérimentée (ayant construit avec succès, dans un langage purement OO, plusieurs systèmes contenants plusieurs milliers de classes chacun), ne comptez pas sur les cas d'utilisation pour vous aider dans l'analyse et la conception orientées objet.

Ne vous méprenez pas : les cas d'utilisation sont toujours un outil potentiellement utile. Mais leur rôle dans la construction de logiciel orienté objet a été mal compris : ils sont plus considérés comme outil de *validation* que comme outil d'analyse. Si (comme vous le devriez) vous disposez d'une équipe distincte d'assurance qualité, elle peut trouver dans les cas d'utilisation un moyen de vérifier si une proposition de modèle d'analyse ou un essai de conception présentent des oublis. L'équipe d'AQ peut ainsi vérifier que le système sera à même de traiter les scénarios typiques identifiés par les utilisateurs. (Dans certains cas de réponse négative, il se peut que le modèle propose un scénario différent aboutissant à des résultats identiques ou meilleurs. C'est, bien sûr, acceptable.)

Une autre application possible des cas d'utilisation concerne les aspects finaux de l'implémentation, lorsqu'on veut s'assurer que le système contient des routines pour les scénarios typiques d'utilisation. De telles routines auront souvent un comportement abstrait, décrivant un schéma général effectif s'appuyant sur des routines retardées que les divers composants du système, et les futurs ajouts à celui-ci, peuvent redéfinir de différentes manières. ([Jacobson 1992] mentionne effectivement une notion de *cas abstrait d'utilisation* qui correspond au concept orienté objet de classe de comportement.)

En tant que mécanisme de validation et guide d'implémentation, les cas d'utilisation peuvent être bénéfiques. Mais, dans la technologie objet, ils ne représentent pas un mécanisme de conception et d'analyse utile. Les analystes et constructeurs de systèmes devraient se concentrer sur les abstractions, et non sur des moyens particuliers d'ordonnancer des opérations sur ces abstractions.

Cartes CRC

K. Beck et W. Cunningham : "A Laboratory for Teaching O-O Thinking", OOPSLA '89, pages 1-6.

Pour être complet, mentionnons le fait que les cartes CRC peuvent être un moyen de trouver des classes. Les cartes CRC (*Classe*, *Responsabilité*, *Collaboration*) sont des cartes en papier, 4 pouces par 6 pouces (10,16 centimètres par 15,24 centimètres), sur lesquelles les concepteurs évoquent les classes potentielles en fonction de leurs responsabilités et de la façon dont elles communiquent. Il faut dire, à son actif, que cette technique ne nécessite qu'un budget d'équipement des plus raisonnables (une boîte de cartes est habituellement moins coûteuse qu'une station de travail munie d'outils CASE) et favorise l'interaction au sein des équipes. Sa contribution technique au processus de conception — pour faciliter la détermination et la caractérisation des abstractions utiles — est, cependant, incertaine.

22.5 RÉUTILISATION

La façon la plus aisée et la plus productive de trouver des classes n'est pas de les inventer vous-même, mais de prendre dans une bibliothèque celles qui sont déjà écrites par d'autres concepteurs et validées par l'expérience d'utilisateurs précédents.

Le composant ascendant

La nature ascendante du développement orienté objet devrait s'appliquer à l'ensemble du processus de développement logiciel, en débutant avec l'analyse. Une approche qui ne se concentre que sur les documents d'exigences et les demandes des utilisateurs (du type, par exemple, des cas d'utilisation) ne peut conduire qu'à un système unique en son genre, qui sera coûteux à construire et qui peut passer à côté de nombreuses idées importantes glanées dans les projets précédents. Parcourir ce qui est déjà disponible et voir comment les classes existantes peuvent faciliter le nouveau développement — même si, dans certains cas, cela veut dire adapter les exigences d'origine — fait partie de la tâche incombant à une équipe de développement, et ce dès la phase d'acquisition des exigences.

Trop souvent, quand on évoque la question de trouver des classes, nous pensons à les *concevoir.* Grâce au développement de la technologie objet, la croissance des bibliothèques de qualité et la pénétration de la notion de réutilisabilité, *trouver* reprendra, de plus en plus, le sens étymologique que lui donne le dictionnaire, *rencontrer.*

La sagesse des classes

Il était une fois un jeune homme, habitant la province d'Ood, qui rêvait de connaître le secret pour trouver des classes. Il avait écouté tous les maîtres des environs, mais aucun d'eux ne le savait.

Ayant assisté à la pénitence publique de Yu-Ton, un ancien abbé de l'Ordre Sacré des Flèches et des Bulles, il se mit à espérer que, peut-être, la fin de sa quête était proche. En entrant dans la cellule de Yu, cependant, il découvrit que ce dernier était toujours en train de chercher à comprendre la différence entre classes et objets. Réalisant qu'il n'avait aucune chance d'obtenir, ici, un éclaircissement, il prit congé sans poser de questions.

Sur le chemin du retour, il surprit la conversation de deux pauvres hères, conduisant leurs carrioles tirées par des ânes, et qui parlaient d'un célèbre ermite dont on disait qu'il connaissait le secret des classes. Le jour suivant, il se mit en route afin de trouver ce grand maître. Après avoir parcouru moult sentiers, grimpé moult collines et traversé moult rivières, il atteignit enfin la retraite du maître. Cela faisait si longtemps qu'il

cherchait que sa jeunesse s'était envolée ; mais, comme les autres pèlerins, il allait lui falloir encore subir un rite purificateur de trente-trois mois avant de pouvoir rencontrer l'objet de sa quête.

Enfin, par une sombre journée d'hiver, alors que la neige frappait sauvagement les pics montagneux environnants, il fut admis dans la salle d'audience du maître. Alors que son coeur battait avec le rythme d'un rocher dévalant le lit d'un torrent asséché, il demanda, timidement : "Maître, comment puis-je trouver les classes ?".

Le vieux sage baissa la tête et répondit, d'une voix calme et lente. "Retourne d'où tu viens. Les classes y sont déjà."

Il était tellement éberlué qu'il lui fallut quelques instants avant de se rendre compte que les aides du maître l'emmenaient déjà au loin. Il eut à peine le temps de se précipiter vers la frêle figure qui commençait à disparaître, pour toujours. "Maître", demanda-t-il encore (en criant presque, cette fois), "Juste une dernière question ! S'il vous plaît ! Dis-moi comment s'appelle cette histoire !".

Le vieux maître tourna lourdement la tête. "Ne devrais-tu pas déjà le savoir ? C'est l'histoire de la réutilisation."

22.6 LA MÉTHODE POUR OBTENIR DES CLASSES

Petit à petit, les idées évoquées dans ce chapitre façonnent ce que nous pouvons appeler, sans trop de prétention (sous réserve que nous nous rappelions qu'une méthode n'est toujours qu'un moyen pour incuber, nourrir, canaliser et développer l'invention, et pas un substitut de celle-ci), la méthode pour obtenir les classes dans la construction logicielle orientée objet.

La méthode suppose que l'identification des classes relève de deux activités fortement liées : obtenir des suggestions de classes et éliminer les moins prometteuses d'entre elles. Les deux tableaux qui suivent résument ce que nous avons appris sur ces deux activités. Seules quelques entrées correspondent à des genres spécifiques de classes, comme les classes d'analyse ; les autres sont applicables à tous les cas.

Voici les sources d'idées de classe :

Sources de classes possibles

Source d'idées	Que rechercher
Bibliothèques existantes	• classes qui correspondent aux besoins de l'application ; • classes qui décrivent des concepts liés à l'application ;
Document d'exigences	• termes qui apparaissent souvent ; • termes pour lesquels le texte fournit une définition explicite ; • termes qui ne sont pas définis précisément mais considérés comme acquis tout au long de la présentation ; • (négligez les catégories grammaticales) ;
Discussions avec les clients et les futurs utilisateurs	• abstractions importantes du domaine d'application ; • jargon spécifique du domaine d'application ; • rappelez-vous que les classes provenant du "monde externe" peuvent décrire aussi bien des objets *conceptuels* que *matériels* ;
Documentation (comme un manuel utilisateur) pour d'autres systèmes (par exemple, des concurrents) du même domaine	• abstractions importantes du domaine d'application ; • jargon spécifique du domaine d'application ; • abstractions utiles de conception ;

Descriptions du système ou de systèmes non OO	• éléments de données passés comme arguments entre divers composants du logiciel, en particulier s'ils vont loin ; • zones de mémoire partagée (bloc `COMMON` en Fortran) ; • fichiers importants ; • unités de `DATA DIVISION` (Cobol) ; • types enregistrements (Pascal), structures et types structures (C, C++) jouant un rôle important dans le logiciel, en particulier s'ils sont utilisés par diverses routines ou modules (fichiers en C) ; • entités dans la modélisation ER ;
Discussions avec les concepteurs expérimentaux	• classes de conception ayant été utilisées avec succès lors de développement précédents de nature similaire ;
Littérature concernant les algorithmes et structures de données	• structures de données connues adaptées aux algorithmes efficaces ;
Littérature de conception OO	• schémas de conception applicables ;

Puis, les critères d'analyse et de rejet éventuel des classes potentielles :

Raisons pour rejeter une classe candidate

Signal de danger	Pourquoi être suspicieux
Classes ayant un nom verbal (infinitif ou impératif)	• peut n'être qu'une simple routine, et non une classe ;
Classe complètement effective n'ayant qu'une routine exportée	• peut n'être qu'une simple routine, et non une classe ;
Classe décrite comme "effectuant" quelque chose	• peut ne pas être une bonne abstraction de données ;
Classe sans routine	• peut n'être qu'un morceau opaque d'information, et non un ADT. Ou peut être un ADT dont les routines ont été oubliées ;
Classe n'introduisant aucune ou seulement quelques caractéristiques (mais qui hérite des caractéristiques des parents)	• peut être un cas de "taxomanie" ;
Classe recouvrant plusieurs abstractions	• devrait être coupée en plusieurs classes, une par abstraction ;

22.7 CONCEPTS CLÉS INTRODUITS DANS CE CHAPITRE

- Identifier les classes est l'une des principales tâches de la construction logicielle orientée objet.
- Identifier les classes est un processus double : suggérer des classes *et* rejeter des classes. Identifier des classes potentielles est tout aussi important que rejeter des idées inadaptées.

- Identifier les classes revient à identifier les abstractions pertinentes du domaine modélisé et de l'espace des solutions.
- "Souligner les noms dans le document d'exigences" n'est pas une technique suffisante pour trouver les classes, car ses résultats sont trop dépendants du style utilisé. Cela peut conduire les concepteurs à omettre des classes utiles tout en incorporant des classes qui ne sont pas nécessaires.
- Une caractérisation grossière des classes fait la distinction entre les classes d'analyse, liées aux concepts du monde externe en cours de modélisation, les classes de conception, qui décrivent des décisions architecturales, et les classes d'implémentation, qui décrivent des structures de données et des algorithmes.
- Les classes de conception sont souvent les plus difficiles à inventer.
- Lors de la conception de classes externes, rappelez-vous que les objets externes correspondent autant à des concepts qu'à des choses matérielles.
- Pour décider si une notion donnée justifie de définir une classe associée, appliquez les critères de l'abstraction de données.
- Les classes d'implémentation sont effectives ou retardées, ces dernières décrivant des catégories abstraites de techniques d'implémentation.
- L'héritage fournit un moyen de réutiliser les conceptions antérieures tout en les adaptant.
- Un moyen d'obtenir des classes consiste à évaluer les conceptions candidates et à rechercher toute abstraction non reconnue comme telle, en particulier en analysant les transmissions de données entre modules.
- Les cas d'utilisation, ou scénarios, peuvent être un outil de validation utile et un guide permettant de finaliser une implémentation, mais ne devraient pas être utilisés comme mécanisme d'analyse et de conception.
- Les bibliothèques réutilisables constituent la meilleure source de classes.

22.8 NOTES BIBLIOGRAPHIQUES

Russell J. Abbott dans Comm. ACM, 26, 11, Nov. 1983, p. 882-894.

Utiliser les noms extraits des exigences comme point de départ pour trouver les types des objets a été rendu populaire par [Booch 1986], qui crédite Abbott de cette idée. D'autres conseils peuvent être trouvés dans [Wirfs-Brock 1990].

Un article sur les spécifications formelles [M 1985a] analyse les problèmes soulevés par les documents d'exigences en langue naturelle. En s'appuyant sur une courte description en langue naturelle d'un problème qui, par ailleurs, a été extensivement utilisé par la communauté de la vérification de programmes, il identifie un grand nombre de défauts et offre une taxonomie de ceux-ci (bruit, ambiguïté, contradiction, remord, surspécification, référence en avant) ; il évoque la manière dont les spécifications formelles peuvent remédier à certains de ces problèmes.

[Waldén 1995] présente des conseils utiles pour identifier les classes.

L'annexe B de [Page-Jones 1995] liste de nombreux "symptômes de problèmes" dans les conceptions orientées objet candidates (par exemple, "*l'interface des classes permet des comportements illégaux ou dangereux*"), illustrant, pour les concepteurs, les signaux de danger du type de ceux qui ont été présentés dans le présent chapitre. La table, tout comme le reste du livre, offre des suggestions pour corriger de tels défauts.

[Ong 1993] décrit un outil pour convertir des programmes non OO (essentiellement Fortran) sous forme orientée objet. La conversion est semi-automatique, c'est-à-dire qu'elle repose sur un certain effort manuel. La description par l'auteur de certaines heuristiques pour identifier des classes potentielles par analyse du code d'origine, en particulier en regardant les blocs `COMMON`, est particulièrement pertinente pour ce chapitre.

Simula 1 (le langage de simulation qui a précédé les versions modernes de Simula) est décrit dans [Dahl 1966]. Voir le chapitre 35 pour plus de références sur Simula.

Parmi les livres sur les structures de données typiques, qui fournissent une source précieuse de classes d'implémentation, on trouve le traité célèbre de Knuth [Knuth 1968] [Knuth 1981] [Knuth 1973] et de nombreux livres universitaires comme [Aho 1974] [Aho 1983].

Un livre récent, [Gore 1996], présente les structures de données et les algorithmes fondamentaux de manière complètement orientée objet.

Parmi les sources de classes de conception, citons [Gamma 1995], qui présente un certain nombre de schémas de conception (*design patterns*) pour C++, et [M 1994a], un recueil de techniques de conception de bibliothèques et de classes réutilisables, évoquant en détail les notions de classe de traitement (*handle class*) et de classe d'itérateurs (*iterator class*). [Krief 1996] présente le modèle MVC de Smalltalk.

EXERCICES

E22.1 Les parties entières vues comme des entiers

Voir "Est-ce qu'une nouvelle classe est nécessaire ?", page 699.

Montrez comment on peut définir une classe `FLOOR` comme héritière de `INTEGER`, restreignant les opérations applicables.

E22.2 Inspecter les objets

Daniel Halbert et Patrick O'Brien évoquent le problème suivant, se produisant lors de la conception d'environnements de développement logiciel :

D'après [Halbert 1987], *légèrement résumé.*

*Considérez la conception d'un service d'***inspection** *utilisé pour afficher, dans une fenêtre de débogage, des informations concernant un objet : le contenu de ses champs et, peut-être, quelques valeurs calculées. Des catégories différentes d'inspecteurs sont nécessaires pour différents types d'objets. Par exemple, toute l'information pertinente concernant un point peut être affichée d'un coup dans un format simple, alors qu'il serait préférable qu'un tableau à deux dimensions de taille importante soit affiché sous forme de matrice pouvant être déroulée horizontalement et verticalement.*

Vous devriez d'abord décider où installer le comportement de l'inspecteur : *dans la* [*classe génératrice*] *de l'objet devant être inspecté ou dans une classe nouvelle et séparée ?*

Répondez à cette question en considérant les avantages et les inconvénients des diverses possibilités. (*Note* : les prochains chapitres traitant de l'héritage peuvent s'avérer utiles.)