**Jean-Pierre Rosen**

# *MÉTHODES DE GÉNIE LOGICIEL AVEC ADA 95*

## *Deuxième édition*

**Licence de Libre Diffusion des Documents -- LLDD version 1**

Ce document peut être librement lu, stocké, reproduit, diffusé, traduit et cité par tous moyens et sur tous supports aux conditions suivantes:

tout lecteur ou utilisateur de ce document reconnaît avoir pris connaissance de ce qu'aucune garantie n'est donnée quant à son contenu, à tout point de vue, notamment véracité, précision et adéquation pour toute utilisation;

il n'est procédé à aucune modification autre que cosmétique, changement de format de représentation, traduction, correction d'une erreur de syntaxe évidente, ou en accord avec les clauses ci-dessous;

des commentaires ou additions peuvent êtres insérés à condition d'apparaître clairement comme tels; les traductions ou fragments doivent faire clairement référence à une copie originale complète, si possible à une copie facilement accessible.

les traductions et les commentaires ou ajouts insérés doivent être datés et leur(s) auteur(s) doi(ven)t être identifiable(s) (éventuellement au travers d'un alias);

cette licence est préservée et s'applique à l'ensemble du document et des modifications et ajouts éventuels (sauf en cas de citation courte), quelqu'en soit le format de représentation;

quel que soit le mode de stockage, reproduction ou diffusion, toute personne ayant accès à une version numérisée de ce document doit pouvoir en faire une copie numérisée dans un format directement utilisable et si possible éditable, suivant les standards publics, et publiquement documentés, en usage.

la transmission de ce document à un tiers se fait avec transmission de cette licence, sans modification, et en particulier sans addition de clause ou contrainte nouvelle, explicite ou implicite, liée ou non à cette transmission. En particulier, en cas d'inclusion dans une base de données ou une collection, le propriétaire ou l'exploitant de la base ou de la collection s'interdit tout droit de regard lié à ce stockage et concernant l'utilisation qui pourrait être faite du document après extraction de la base ou de la collection, seul ou en relation avec d'autres documents.

Toute incompatibilité des clauses ci-dessus avec des dispositions ou contraintes légales, contractuelles ou judiciaires implique une limitation correspondante du droit de lecture, utilisation ou redistribution verbatim ou modifiée du document.

# Table des matières

# Préface à la deuxième édition

En relisant aujourd'hui (2004) la première édition de cet ouvrage, parue en 1995, je fus frappé de ce qu'il était à la fois obsolète et toujours d'actualité. Obsolète, car de nouveaux langages, de nouvelles méthodes sont apparus : on ne comprendrait pas aujourd'hui qu'un livre traitant des rapports entre langages de programmation et méthodes ne traite ni de Java, ni d'UML. Mais toujours d'actualité, car si les modes changent, les problèmes restent les mêmes. Les nouveaux outils permettent-ils vraiment de développer mieux qu'il y a dix ans ? On est en droit d'en douter.

C'est pourquoi j'ai décidé de remettre à jour ce livre. On y retrouvera l'essentiel de la version précédente, les principales modifications portant sur les langages et méthodes apparus postérieurement à l'édition originale. Des méthodes qui ont disparu dans la vague UML ne sont également plus décrites. Il y aurait certainement encore beaucoup d'autres améliorations à apporter, mais j'ai préféré obéir à l'adage qui dit: «mieux vaut quelque chose d'imparfait maintenant que quelque chose de parfait jamais».

L'autre différence par rapport à la première édition est la diffusion de cet ouvrage sous forme de document libre. Si de nombreuses personnes m'ont amicalement poussé à mettre à jour cet ouvrage, il ne semble pas que le marché potentiel soit de nature à intéresser un éditeur. Alors autant le mettre à disposition de tout le monde. Ceci m'a conduit à modifier considérablement la présentation, en particulier pour la mettre au format A4. Ce format n'est pas le plus agréable en terme de lecture, mais il permet de minimiser le nombre de pages, ce qui est important maintenant que chacun est libre de l'imprimer pour soi, pour ses élèves, ou pour toute autre personne intéressée.

Le logiciel libre, qui s'est considérablement développé ces dernières années, est un processus fondamentalement coopératif; chacun peut apporter sa contribution au développement. Si l'analogie n'est pas complète avec un document libre, j'espère que les lecteurs voudront bien me faire part de leurs remarques, de leurs critiques et de leur suggestions d'amélioration. Grâce au mode de diffusion libre, cet ouvrage pourra ainsi devenir un document vivant, évoluant au gré de l'apparition de nouvelles méthodes et de nouveaux langages. Lecteur qui bénéficiez du libre accès à ce livre, il n'appartient qu'à vous qu'il soit également un peu le vôtre!

# Introduction

On pourrait définir le génie logiciel comme la conjonction de méthodes, de règles d'organisation et d'outils. Les outils sont nombreux et abondamment vantés par leurs fabricants. De multiples ouvrages décrivent des méthodes, certaines très célèbres, d'autres plus confidentielles. Et pourtant, *dans la pratique*, il semble que le génie logiciel ne soit pas toujours entré dans les mœurs des informaticiens. D'où provient ce décalage entre théorie et pratique ?

Au-delà des considérations théoriques, le génie logiciel est d'abord un *état d'esprit*, demandant rigueur, méthode... et modestie, puisque toute la difficulté d'une «belle» conception est de faire en sorte que n'importe qui puisse la reprendre après son concepteur initial. Le but de ce livre est de faire sentir au lecteur l'importance de la *démarche* «génie logiciel» et de mener une réflexion sur la nature des méthodes, leurs origines, leurs points communs et leurs différences, et leurs rapports aux langages de programmation.

Dans ce cadre, nous avons naturellement adopté le langage Ada comme moyen d'expression. Pourquoi ? Parce que c'est le seul langage qui ait été conçu dès le départ en fonction d'un *cahier des charges* dicté par les impératifs du génie logiciel. Nous verrons tout au long de ce livre que chacune des fonctionnalités, chacun des contrôles apportés par ce langage trouvent leur origine dans un principe méthodologique. Mieux : l'explication même des mécanismes du langage est souvent un exemple parlant d'un principe de génie logiciel sous-jacent. Cette question du langage de programmation est souvent prétexte à des débats quasi religieux. Soyons clair : la solution aux difficultés du développement logiciel ne saurait provenir *seulement* d'un langage de programmation, quel qu'il soit. Seuls l'utilisation de méthodes de conception rigoureuses et le développement de composants logiciels réutilisables peuvent maîtriser la complexité. Mais certains en ont un peu hâtivement conclu que le langage n'avait *aucune* importance, du moment que l'on adoptait une «bonne» méthode. En fait, son rôle est beaucoup plus important qu'on ne le croit généralement. D'une part, ses possibilités influencent, qu'on le veuille ou non, la conception ; d'autre part, un langage adapté doit poursuivre et vérifier la méthode, pour faciliter le passage de la conception au codage et détecter les incohérences possibles. C'est typiquement ce qu'Ada permet, qu'apprécient ses zélateurs et que lui reprochent ses détracteurs : il ne laisse pas passer les fautes de conception et n'autorise pas la programmation indisciplinée.

Ce livre s'adresse donc aux responsables de projet, responsables qualité et développeurs, mais aussi aux enseignants et étudiants qui souhaitent un panorama des principes du génie logiciel, des méthodes qui l'accompagnent et de la façon de les mettre en œuvre jusqu'au niveau du codage. Ceux qui ne connaissent pas Ada y découvriront qu'un langage de programmation peut être un véritable outil de génie logiciel, et nous espérons qu'ils auront alors à cœur de l'essayer, ce qu'ils peuvent faire aisément maintenant qu'il existe un compilateur libre[1] ; quant à ceux qui le connaissent déjà, ils découvriront les nouvelles possibilités d'Ada 95 ainsi que la façon de l'utiliser au mieux dans un contexte de génie logiciel.

Dans la première partie, nous discuterons des rôles respectifs des méthodes de conception et des langages. Nous verrons comment Ada peut apporter des modifications profondes à la façon d'organiser le développement du logiciel, d'autant plus que les nouvelles fonctionnalités apportées par la révision 95 ont permis de combler les dernières lacunes qu'il comportait vis-à-vis de certaines méthodes. La deuxième partie présente les principales variétés de méthodes de conception utilisées aujourd'hui, et le support *actif* qu'Ada leur apporte. La troisième partie se consacre au problème plus

[1] Disponible sur le site principal, ftp://cs.nyu.edu/pub/gnat, ou son miroir français (université Paris VI) ftp://ftp.lip6.fr/pub/gnat.

spécifique du développement et de l'utilisation de composants logiciels. La quatrième aborde les problèmes de l'organisation générale du développement et des choix qui interviennent à tous les stades du projet. Enfin la cinquième partie présente un cadre méthodologique général qui met en pratique, à partir d'une méthode traditionnelle, les principes exposés dans le reste du livre. Chaque chapitre est accompagné d'exercices, sujets de réflexion ou mise en pratique de la théorie.

Si une certaine connaissance de la programmation et d'au moins un langage est nécessaire, il n'est nullement besoin de connaître Ada pour lire ce livre. Un préambule présente rapidement le langage ; nous y avons séparé les fonctionnalités de la version 83 de celles de la version 95, pour permettre un accès plus facile à ceux qui, connaissant déjà l'ancien langage, ne sont intéressés que par les nouveautés. Un glossaire et un index aideront également les nouveaux venus à mieux appréhender le livre.

> Des hors-texte de ce style donnent des digressions «linguistiques» complémentaires là où elles sont nécessaires à la bonne compréhension des exemples.

Les exemples de programme, ainsi que les références dans le texte à des éléments du langage (instructions, variables...) sont en caractères `courrier` ; lorsqu'il s'agit de mots clés du langage, ils sont en caractères **`gras`**.

Le problème de la langue est toujours présent dans un ouvrage d'informatique, domaine souvent envahi de jargon anglophone. Nous utilisons le français autant que possible, sans aller jusqu'à employer certaines locutions qui, bien que recommandées officiellement, nous ont paru exagérées : disons-le franchement, nous n'avons jamais pu nous habituer à dire «bogue» pour «bug». Les termes purement Ada ont été repris du lexique qui sert à la traduction, en cours à l'heure où nous écrivons ces lignes, de la norme Ada 95 en français.

Si, à la fin de cet ouvrage, le lecteur est convaincu de l'importance des méthodes de conception et de la démarche générale du génie logiciel ; s'il a pris conscience que le développement informatique est une longue suite de choix demandant d'arbitrer des compromis parmi des exigences contradictoires ; si notamment son choix du langage de programmation résulte d'une volonté consciente de prolonger le cadre méthodologique et non du simple hasard des disponibilités de compilateurs sur sa machine ; s'il décide de mettre en place une politique de réutilisation à l'échelle de son entreprise ; alors nous sommes convaincu que le langage choisi sera souvent Ada, et nous penserons que le but de cet ouvrage aura été atteint.

Je terminerai en remerciant, selon la tradition, ceux dont les avis, conseils, critiques et remarques ont contribué à faire de ce livre ce qu'il est. Je dois tout d'abord, et à de multiples titres, une mention spéciale à Michel Gauthier, de l'université de Limoges ; les nombreuses références à ses publications montrent bien ce que je lui dois. Mes remerciements vont ensuite à mon assistante de tous les jours, Catherine Supper, et à ceux qui ont bien voulu consacrer une partie de leur temps toujours précieux à relire les premières versions de cet ouvrage : Alain Detrie, Michael Feldmann, Jean-Alain Hernandez et Patrice Jano. Je dois également une mention spéciale à tous les membres d'Ada-France : nombre des idées de ce livre proviennent des discussions que nous avons eues lors de nos réunions. Je ne saurais oublier non plus tous mes élèves, depuis mes débuts à l'ENST jusqu'aux participants des séminaires Adalog : ils ne se rendront jamais compte à quel point une question apparemment naïve peut obliger un enseignant à faire progresser sa propre réflexion sur un sujet.

Je tiens enfin à remercier mon portable, un petit Compaq Aero, pour m'avoir permis de travailler à ce livre dans les endroits les plus divers, depuis les fauteuils d'avion (sans oublier, hélas ! les salles d'attente) jusque dans le lit conjugal ; je remercie ma femme de m'avoir encouragé durant tout ce temps, ainsi que d'avoir bien voulu tolérer la présence dudit portable dans ledit lit conjugal ; enfin je remercie mes enfants de m'avoir parfois laissé utiliser mon ordinateur malgré la présence sur son disque dur de jeux dont ils voyaient beaucoup plus clairement l'utilité !

# Préambule

# Une présentation rapide du langage Ada

Le langage que nous utilisons dans ce livre est l'Ada d'aujourd'hui, Ada 95. Celui-ci étant relativement récent, nous avons pensé qu'il serait utile de fournir aux anciens utilisateurs des précisions sur les nouveautés apportées par la révision de la norme.

Aussi avons-nous divisé cette présentation en deux parties. La première explique simplement le «noyau de base», minimum nécessaire pour comprendre le reste de ce livre ; presque tous ses éléments figuraient déjà dans Ada 83, et les utilisateurs actuels d'Ada pourront la survoler rapidement. La deuxième partie est plus technique et présente avec plus de détails les grandes lignes des nouveautés importantes. Sa lecture n'est pas nécessaire à la compréhension du reste de l'ouvrage, et ceux qui ne connaissent pas déjà le langage pourront la sauter.

Ce préambule ne cherche pas l'exhaustivité ; pour une étude complète du langage, nous renvoyons le lecteur à la bibliographie en fin d'ouvrage.

# 1
# Origine d'Ada

## 1.1 Historique

En janvier 1975 le ministère américain de la Défense (DoD) a constitué un comité d'experts, le High Order Language Working Group (HOLWG), avec pour mission de trouver une approche systématique aux problèmes de qualité et de coûts des logiciels militaires. La plus grosse part de ces logiciels, ou tout du moins là où résidaient les plus gros frais de maintenance, était dans le domaine des systèmes temps réel embarqués (*embedded*), c'est-à-dire des systèmes informatiques intégrés dans un ensemble électromécanique plus vaste : systèmes d'armes, de radar, de commutation... Développer un langage de programmation universel apparut alors comme une solution à beaucoup de ces problèmes, et le HOLWG produisit une succession de cahiers des charges spécifiant les caractéristiques souhaitables d'un tel langage. Au printemps de 1977 dix-sept organismes répondirent à l'appel d'offres, parmi lesquels quatre furent retenus pour une pré-étude : Softech, Intermetrics, SRI et Cii-Honeywell-Bull. Les propositions furent évaluées de façon anonyme, et en mars 1978, il ne restait plus en lice qu'Intermetrics et Honeywell-Bull, l'équipe française dirigée par Jean Ichbiah remportant l'appel d'offre un an plus tard. Le langage fut baptisé Ada, du nom d'Ada Augusta Byron (cf. encadré).

Le document proposé en 1979 était en fait beaucoup trop vague pour permettre l'établissement d'un standard rigoureux. Une première version révisée fut produite en juillet 1980, et l'on pensait à l'époque que le langage pourrait être normalisé avant la fin de cette même année. Il faut comprendre que la qualité de ce document était au moins aussi bonne que celle de la plupart des normes des autres langages de programmation. Mais pour Ada, il fallait faire beaucoup mieux ! En effet, le document comportait encore nombre de petites imprécisions, de possibilités d'interprétations divergentes ou de dépendances excessives à l'implémentation. L'exigence de portabilité nécessitait une définition beaucoup plus précise, et il fallut attendre 1983 pour obtenir la standardisation par l'ANSI[1]. La standardisation internationale exigeait que la norme fût déposée simultanément en deux des trois langues officielles de l'ISO[2] qui sont l'anglais, le français et le russe. Pour des raisons qui leur appartiennent, les américains préférèrent financer la traduction française plutôt que russe, et la normalisation internationale fut suspendue à la parution de la norme française. La traduction fut effectuée avec un soin extrême : il fallut attendre 1987 pour obtenir une norme satisfaisante [Afn87] et la standardisation ISO [Iso87]. Ada est également une norme européenne (C.E.N. 28652), allemande (DIN 66268), suédoise (SS 63-6115), tchèque, japonaise...

Il était important de définir rigoureusement le langage ; encore fallait-il que les compilateurs respectent la norme ! Pour cela, le DoD a déposé le nom «Ada», et l'autorisation d'utiliser ce nom

---

[1] American National Standard Institute

[2] Organisation Internationale de Normalisation / International Organisation for Standardization / МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ. Noter que le sigle «ISO» n'est l'acronyme de son nom dans aucune de ses langues officielles !

Ada Augusta Byron, comtesse de Lovelace (1815-1851), était la fille du Lord Byron, le fameux poète anglais. Elle était passionnée de mathématiques, chose scandaleuse pour une femme à son époque et qui lui attira bien des inimitiés. Amie et disciple de Charles Babbage, elle eut l'intuition de la possibilité d'utiliser des suites d'instructions codées d'après le modèle des métiers Jacquard. Elle écrivit des programmes pour calculer les nombres de Bernouilli sur la machine de Babbage, mais elle ne put les exécuter puisque la machine ne fut jamais construite. Ses programmes furent recodés bien plus tard en PL/1 et fonctionnèrent, paraît-il, du premier coup. Elle est donc considérée comme le premier programmeur de l'histoire.

pour un compilateur fut subordonnée à la confrontation avec succès du compilateur à une suite de validation. Le premier compilateur Ada ainsi validé fut Ada/ED, réalisé par l'équipe de New York University. De nombreux compilateurs ont été validés depuis[1]. Quelques années plus tard, le DoD a abandonné ses droits sur la marque, considérant que cette contrainte n'était plus nécessaire. Il a en revanche déposé une marque de certification, utilisable uniquement par les compilateurs validés. La notion de validation est maintenant tellement liée au langage Ada qu'il n'existe pas de compilateur non validé.

Il est de règle, cinq ans après la parution d'une norme, soit de confirmer le standard tel quel, soit de décider de mettre en route une révision. On savait dès le début que celle-ci serait nécessaire : certains pays n'avaient voté la norme d'origine que pour ne pas ralentir le processus de standardisation et à la condition qu'à la première occasion, on réglerait certains problèmes restés en suspens, principalement celui du jeu de caractères (Ada 83 utilisait l'ASCII 7 bits, qui est notoirement insuffisant pour les langues européennes[2]). Plusieurs autres facteurs concouraient à la nécessité d'une révision. D'abord, comme toute œuvre humaine, le standard comportait des imprécisions, des omissions et des ambiguïtés. A l'ISO, un comité de maintenance, l'ARG (*Ada Rapporteur Group*), est chargé de régler ces problèmes ; une révision de la norme se devait d'incorporer ses décisions. Ensuite, la mise en œuvre pratique du langage a montré certaines petites faiblesses, ou la nécessité de certains mécanismes dus à l'évolution des modes de programmation (programmation orientée objet notamment). Enfin, il y avait un désir de reformuler certaines règles afin de les rendre plus faciles à interpréter.

La décision fut donc prise en 1988 de démarrer le processus de révision. Le DoD fournit encore une fois le support financier nécessaire, et la révision fut conduite par l'*Ada 9X Project Office*, dirigé par Chris Anderson, qui confia la réalisation technique à une équipe d'Intermetrics conduite par Tucker Taft. Dans la grande tradition d'ouverture d'Ada, un appel fut lancé aux utilisateurs pour connaître leurs desiderata. La réponse fut très importante, et une synthèse en fut tirée, regroupant par thèmes les principales tendances des demandes, et leur attribuant un degré de priorité. De cette synthèse on tira un document, le *Mapping Document*, destiné à décrire les transformations nécessaires pour satisfaire les points identifiés à l'étape précédente. Ce document, encore très touffu, comportait quasiment tout ce qu'il était *envisageable* de modifier, c'est-à-dire nettement plus qu'il n'était *souhaitable*. Sur la base de ce document, une première réunion de l'ISO se teint au printemps 1992 dans la région de Francfort. Cette réunion marathon (50 participants de 12 pays[3] travaillèrent de façon quasi ininterrompue, pendant une semaine, de 9h du matin à 11h du soir) permit d'aboutir à un accord unanime sur les éléments qu'il convenait de changer dans le langage. D'autres réunions internationales, à Salem puis à Boston (USA), permirent de préciser les détails de la révision. Une dernière réunion au mois de mars 1994 à Villars (Suisse) figea définitivement le langage. Les documents ne subirent alors plus que des changements éditoriaux pendant que le processus

1 En septembre 1994, on recensait 806 compilateurs validés par 52 compagnies.

2 En fait, des mesures ont été prises dès 1992 pour permettre d'étendre le jeu de caractères sans attendre la révision de la norme.

3 Allemagne, Belgique, Canada, Grande-Bretagne, Espagne, France, Japon, Pays-Bas, Russie, Suède, Suisse, USA

bureaucratique suivait son chemin, pour aboutir à l'enregistrement officiel par l'ISO le 15 février 1995.

Mais le temps continua à passer... 1995 plus cinq ans, vint le temps de décider de l'opportunité d'une nouvelle révision. En fait, dès novembre 1999 l'ISO prit la décision de ne pas commencer d'effort de révision. Bien entendu, la norme 1995 comportait des imprécisions et des erreurs qu'il fallait tout de même corriger. Il fut donc décidé de simplement faire paraître des corrigendums techniques pour tenir la définition du langage à jour. Un tel corrigendum technique a été officiellement approuvé le 1er juin 2001.

Un corrigendum technique ne peut que corriger des erreurs de définition du langage, il ne peut rien modifier fondamentalement, ni apporter d'améliorations. L'apparition de Java (avec la notion d'interfaces), certains problèmes pratiques apparus à l'usage, le besoin de nouvelles bibliothèques (conteneurs) demandaient des ajouts plus conséquents que ceux autorisés par le corrigendum. Il fut donc décidé début 2001 de préparer un amendement à la norme. Un amendement permet de rajouter de nouvelles fonctionnalités au langage, mais reste d'ampleur très inférieure à une révision. Le texte de l'amendement a été figé à la fin de 2005, ce qui fait que le langage est appelé Ada 2005, même si la parution officielle par l'ISO n'a été proclamée que le 9 mars 2007.

## 1.2 Objectifs du langage

Contrairement à beaucoup de langages de programmation, Ada n'est pas le fruit des idées personnelles de quelque grand nom de l'informatique, mais a été conçu pour répondre à un *cahier des charges* précis, dont l'idée directrice était de diminuer le coût des logiciels, en tenant compte de tous les aspects du cycle de vie. Le langage est donc bâti autour de quelques idées-forces :

*Privilégier la facilité de maintenance sur la facilité d'écriture :* le coût de codage représente environ 6% du coût global d'un logiciel ; la maintenance représente plus de 60%.
*Fournir un contrôle de type extrêmement rigoureux :* plus une erreur est diagnostiquée tôt, moins elle est coûteuse à corriger. Le langage fournit des outils permettant de diagnostiquer beaucoup d'erreurs de cohérence dès la compilation.
*Permettre une programmation intrinsèquement sûre :* des contrôles nombreux à l'exécution permettent de diagnostiquer les erreurs dynamiques. Un programme doit être capable de traiter *toutes* les situations anormales, y compris celles résultant d'erreurs du programme (autovérifications, fonctionnement en mode dégradé).
*Offrir un support aux méthodologies de programmation :* le langage doit faciliter la mise en œuvre des méthodes du génie logiciel, sans exclusive et sans omission.
*Etre portable entre machines d'architectures différentes :* les programmes doivent donner des résultats identiques, ou au moins équivalents, sur des machines différentes, y compris en ce qui concerne la précision des calculs numériques.
*Permettre des implémentations efficaces et donner accès à des interfaces de bas niveau :* exigence indispensable à la réalisation notamment de systèmes «temps réel».
*Offrir un support à une industrie du composant logiciel :* en fournissant des interfaces standard et des garanties de portabilité sur toutes les machines, Ada permet la création d'entreprises spécialisées en composants logiciels, tandis que d'autres réalisent des produits finis (applications) par assemblage de ces composants.

## 1.3 Au-delà du langage

La définition d'un nouveau langage est une condition nécessaire, mais non suffisante, pour entrer dans une ère de production industrielle de logiciel. Aussi ne peut-il exister de compilateur Ada sans un environnement de programmation associé ; en particulier, le langage définit des règles très strictes concernant les dépendances entre unités de compilation, pour garantir la cohérence globale

des différentes unités constituant un programme Ada. Ceci nécessite au moins un outil de gestion de projet associé. En fait, parallèlement à la définition du langage, se poursuivait un effort de définition de l'APSE (*Ada Programming Support Environment*) qui a abouti au rapport STONEMAN. Celui-ci définit les outils logiciels qui doivent faire partie d'un environnement de programmation Ada.

Aujourd'hui, il existe non seulement des compilateurs de qualité industrielle, mais aussi des outils d'environnement de haut niveau : metteurs au point symboliques, outils de gestion de projet, de documentation et de support méthodologique se sont multipliés. Les objectifs d'Ada sont ceux de tout le mouvement du génie logiciel, et le langage n'est qu'un élément qui s'intègre dans un ensemble, notamment méthodologique, plus vaste.

## 1.4 Exercices

1. Etudier comment ont été développés les langages COBOL, C, Pascal et Ada. Quels sont ceux qui avaient un cahier des charges au départ, et comment l'historique explique-t-il les particularités de chacun ?
2. Une particularité d'Ada est d'avoir été normalisé avant l'apparition du premier compilateur. Expliquer en quoi ceci est un facteur important pour la qualité de la définition du langage.

# 2

# Présentation du langage

Ada est un langage algorithmique d'une puissance d'expression considérable, dérivé de Pascal dont il a retenu les structures de contrôle et certains types de données, mais avec en plus des possibilités d'encapsulation de données, de modularité, de modélisation de tâches parallèles, et de traitement des situations exceptionnelles. Ada reste un langage procédural «classique» (contrairement à LISP, SNOBOL, SETL ou Prolog par exemple), mais c'est un langage qui a un spectre sémantique assez vaste, et dont les apports originaux sont surtout du domaine de l'ingénierie du logiciel : fiabilité, maintenabilité, modularité, portabilité, réutilisabilité de composants logiciels.

Ada est-il un langage orienté objet ? Cette question a fait couler beaucoup d'encre. Ada 83 offrait un excellent support aux méthodes objet par composition mais n'avait pas de structure réalisant *directement* l'héritage ; il s'en est suivi une querelle de mots pour savoir si Ada était «orienté objet» ou non, et l'utilisation parfois du terme «basé objet» (?) pour décrire les langages tels qu'Ada qui fournissaient indiscutablement la notion d'objet, tout en se passant fort bien de l'héritage... Ce débat n'a plus lieu d'être : la version 95 a apporté le support de l'héritage et des méthodes par classification. Nous discuterons complètement de ces questions dans la deuxième partie du livre, mais retenons que, maintenant, Ada est indiscutablement un langage orienté objet.

## 2.1 Un cadre général proche de Pascal

Lors de l'appel d'offres qui conduisit à la définition d'Ada, il était spécifié que la proposition de langage devait être fondée sur la base d'un «grand» langage classique (Pascal, PL/1, Algol...). Il est significatif que les quatre langages finalistes avaient tous choisi Pascal comme base de départ.

Le programme ci-dessous est un premier exemple qui imprime «Bonjour», «Bon après-midi» ou «Bonsoir» en fonction de l'heure de la journée.

```
with Text_IO, Calendar;
procedure Bonjour is
   use Text_IO, Calendar;
   Heure : Day_Duration;
begin
   Heure := Seconds(Clock)/3600;
   if Heure < 12.00 then
      Put_Line ("Bonjour");
   elsif Heure < 19.00 then
      Put_Line ("Bon après-midi");
   else
      Put_Line ("Bonsoir");
   end if;
end Bonjour;
```

Il n'y a pas de construction spéciale pour désigner le programme principal : c'est une procédure comme les autres. La clause **`with`** qui est en tête permet d'utiliser deux *paquetages* (**`package`**) qui fournissent respectivement l'accès aux fonctionnalités d'entrées-sorties (`Text_IO`) et à l'utilisation du temps (`Calendar`). De façon générale, toute unité de compilation (structure pouvant être

compilée séparément, comme les sous-programmes et paquetages) doit citer au début les autres unités qu'elle utilise au moyen d'une clause **`with`**. La clause **`use`** est une simplification d'écriture dont nous reparlerons plus tard.

Les instructions de base sont inspirées de celles de Pascal (Fig. 0), tout en tenant compte de ses imperfections reconnues : toutes les instructions se terminent par un point-virgule et les instructions structurées se terminent par un mot clé, éliminant ainsi le besoin de blocs **`begin..end`** (ou des accolades `{..}` de C).

```
if condition then
  instructions
elsif condition then
  instructions
else
  instructions
end if;
```

```
case expression is
   when choix {|choix} =>
      instructions
   when choix {|choix} =>
      instructions
   when others =>
      instructions
end case;
```

```
[Etiquette_de_boucle:]
   [while condition] | [for ident. in [reverse] intervalle]
   loop
      instructions
   end loop;
```

**Figure 0 :** Instructions de base

L'ordre des déclarations n'est pas imposé : il est donc possible de regrouper les types, variables et constantes qui sont logiquement reliés. Signalons aussi la présence de deux originalités fort utiles : les *pragmas* et les *attributs*. Un pragma permet de donner des indications au compilateur ; par exemple, si la performance est plus importante pour une application que l'espace mémoire, on pourra en informer l'optimiseur au moyen du pragma :

```
pragma Optimize (Time);
```

Les attributs permettent d'obtenir des renseignements (pouvant dépendre de l'implémentation) sur des objets, des types, des sous-programmes... Si l'on souhaite par exemple connaître la taille en bits d'une variable (attribut `'Size`), on écrira :

```
Taille := X'Size;
```

Nous ne décrirons pas ici tous les pragmas et attributs, mais nous donnerons des précisions lorsque nous en rencontrerons dans le cours du livre.

## 2.2 Sous-programmes

Les procédures et fonctions (que l'on appelle collectivement des sous-programmes) se présentent de façon similaire à Pascal : un en-tête, suivi de déclarations, puis d'instructions (Fig. 1).

**Procédure**

```
procedure Identificateur [(paramètres)] is
   Déclarations
begin
   Instructions
exception
   Traite-exceptions
end Identificateur;
```

**Fonction**

```
function Identificateur [(paramètres)] return Identificateur_de_type is
   Déclarations
begin
   Instructions
exception
   Traite-exceptions
end Identificateur;
```

**Paramètres**

```
Identificateur : [ in | out | in out ] Identificateur_de_type
```

**Figure 1 :** Sous-programmes

Il n'existe *aucune* limitation au type retourné par une fonction : ce peut être un tableau, un article, une tâche... Le mode de passage des paramètres fait référence à l'usage qui en est fait : les paramètres peuvent ainsi être déclarés **`in`** (lecture seulement), **`out`** (écriture seulement), ou **`in out`** (lecture et écriture). Ceci remplace avantageusement la notion de «passage par valeur» ou de «passage par variable». Bien sûr, tous les sous-programmes sont récursifs (et réentrants, puisque Ada permet le parallélisme).

Plusieurs sous-programmes peuvent porter le même nom s'ils diffèrent par le type de leurs arguments. Cette possibilité s'appelle la *surcharge* et permet de donner des noms identiques à des sous-programmes effectuant la même fonction logique sur des types différents : par exemple, toutes les procédures d'impression s'appellent `Put` quel que soit le type de ce que l'on imprime.

## 2.3 Exceptions

La notion d'exception fournit un moyen commode de traiter tout ce qui peut être considéré comme «anormal» ou «exceptionnel» dans le déroulement d'un programme. Une exception se comporte comme une sorte de déroutement déclenché par programme depuis une séquence d'exécution «normale» vers une séquence chargée de traiter les cas exceptionnels. Une exception peut être déclarée par l'utilisateur ; certaines exceptions sont prédéfinies, comme `Storage_Error` en cas de mémoire insuffisante, ou `Constraint_Error` en cas de valeur incorrecte affectée à une variable.

Une exception est déclenchée soit implicitement (en cas de non-respect d'une règle du langage à l'exécution), soit explicitement au moyen de l'instruction **`raise`**. Un bloc de programme peut déclarer un *traite-exception* auquel le contrôle sera donné si l'exception citée se produit dans le bloc considéré. Voici un bloc dans lequel un traite-exception permet de renvoyer une valeur par défaut dans le cas où il se produit une division par 0 :

```
begin
   Résultat := A/B;
exception
   when Constraint_Error =>
       Résultat := Float'Large;
end;
```

Si une exception n'est pas traitée localement, elle est *propagée* aux unités appelantes, jusqu'à ce que l'on trouve un traite-exception adapté, ou que l'exception sorte du programme principal, ce qui arrête l'exécution. Une clause spéciale, **`when others`,** permet de traiter toutes les exceptions. Voici un exemple de ce que pourrait être un programme principal qui garantirait un arrêt propre, quoi qu'il arrive dans le programme, sans connaissance *a priori* de la fonctionnalité des éléments appelés :

```
procedure Principale is
begin
   Faire_Le_Travail;
exception
   when others => Nettoyage;
end Principale;
```

Il est important de noter que des événements «catastrophiques», comme la saturation de l'espace mémoire, provoquent simplement en Ada la levée d'exceptions prédéfinies, et sont donc traitables par le programme utilisateur.

## 2.4 Le modèle du typage fort

### *2.4 .1 Le typage fort, même pour les types élémentaires*

Toutes les variables doivent être déclarées et munies d'un type. Le typage est extrêmement strict (beaucoup plus qu'en Pascal – ne parlons pas de C ou de C++), et ceci constitue un atout majeur du langage. Nous y reviendrons souvent.

Un type définit un ensemble de *valeurs* muni d'un ensemble d'*opérations* portant sur ces valeurs. En fait, un type Ada représente des entités de plus haut niveau que les types d'autres langages : il représente en effet une entité *du domaine de problème*, et non une entité *machine*. Le programmeur exprime les exigences de son problème ; si par exemple il doit représenter des temps avec une précision absolue de 1 ms et des longueurs avec une précision relative de $10^{-5}$, il déclarera :

```
type Temps    is delta 0.001 range 0.0 .. 86400.0*366;
type Longueur is digits 5    range 1.0E-16..1.0E25;
```

C'est le compilateur qui choisira, parmi les types machine disponibles, le plus performant satisfaisant *au moins* les exigences ainsi exprimées. Le langage offre une vaste palette de types numériques : types entiers normaux ou modulaires (ces derniers – nouveaux en Ada 95 – sont non signés et munis d'une arithmétique modulaire), types flottants, fixes et décimaux (ces derniers également apportés par Ada 95). Par exemple, la première déclaration ci-dessus correspond à une déclaration d'un nombre *point fixe*, qui approxime les nombres réels avec une erreur *absolue* constante, alors que la seconde correspond à une déclaration de nombre *point flottant* qui approxime les nombres réels avec une erreur *relative* constante. Ces derniers correspondent aux nombres «réels» des autres langages. Noter la possibilité de limiter l'intervalle de valeurs en plus du type et de l'étendue de la précision[1]. Puisqu'il s'agit d'entités de nature différente, deux variables de types différents sont absolument incompatibles entre elles (on ne peut additionner une longueur et un temps !), *même lorsqu'il s'agit de types numériques*. Etant donné :

```
type Age   is range  0 .. 120;  -- Soyons optimistes
type Etage is range -3 .. 10;
  A : Age;
  E : Etage;
```

on peut écrire :

```
A := 5;
E := 5;
```

mais l'affectation suivante est interdite par le compilateur :

[1] $10^{-16}$ m représente le diamètre du quark, $10^{25}$ m est le diamètre de l'Univers (cf. [Mor82]).

```
A := E;    -- ERREUR : incompatibilité de types
```

Autrement dit, on ne peut pas «mélanger des choux et des carottes». Ada est le seul langage à offrir cette propriété, qui paraît tout à fait naturelle aux débutants en informatique... et pose parfois des problèmes aux programmeurs expérimentés. Il est toujours possible de *convertir* des types numériques entre eux (le nom du type sert d'opérateur de conversion). Si l'on veut demander à une personne d'aller à l'étage correspondant à son âge, on peut écrire :

```
E := Etage(A);
```

Ceci montre un point fondamental de la «philosophie» Ada : on n'empêche jamais le programmeur d'écrire ce dont il a besoin, mais si ce qu'il demande est potentiellement dangereux, on exige de lui qu'il décrive explicitement le comportement demandé afin d'avertir le futur programmeur de maintenance.

Ada dispose de types énumératifs et de formes habituelles pour les tableaux et enregistrements simples (nous verrons des formes plus perfectionnées par la suite) :

```
type Couleurs is (Bleu, Rouge, Vert, Jaune, Blanc);
type Valeur_Stock is range 0..1000;
type Stock_Peinture is array (Couleurs) of Valeur_Stock;

type Voiture is
  record
    Le_Modèle   : String (1..6);
    La_Couleur  : Couleurs;
    La_Longueur : Longueur;
  end record;
```

Des *agrégats* permettent de fournir directement des valeurs d'un type structuré :

```
   Un_Stock : Stock;
   Ma_Voiture : Voiture;
begin
   Un_Stock   := (Rouge => 20, Bleu => 30, others => 0);
   Ma_Voiture := ("Espace", Bleu, 3.50);
```

## *2.4 .2 Types et sous-types*

Ada fait une distinction très nette entre la notion de *type* et la notion de *sous-type*. Alors que le type décrit les propriétés générales d'une entité du monde réel, le sous-type exprime des restrictions applicables seulement à certains objets du type. Ces restrictions sont appelées *contraintes*. Une contrainte peut être par exemple la limitation des valeurs à un certain intervalle numérique, ou la précision des bornes d'un type tableau *non contraint* (dont on a laissé les bornes indéfinies au moyen du symbole «boîte» (<>), comme ci-dessous).

```
subtype Adolescent is Age range 12..18;
-- les adolescents doivent avoir entre 12 et 18 ans

type Matrice is array (Positive range <>,
                       Positive range <>) of Float;
-- Toutes les matrices, quelles que soient leurs dimensions

subtype Matrice_3_3 is Matrice (1..3, 1..3);
-- Les seules matrices 3x3
```

L'affectation, le passage de paramètre sont autorisés (à la compilation) si et seulement si les *types* correspondent ; une exception se produira à l'exécution si les *sous-types* sont incompatibles. Un sous-type peut toujours être dynamique : ceci permet notamment de choisir les tailles des tableaux lors de l'exécution, sans pour autant recourir à l'utilisation de pointeurs (qui existent par ailleurs dans le langage, où on les appelle des types *accès*). Ceci permet de définir des sous-programmes travaillant sur des tableaux de bornes quelconques sans perte de contrôle des débordements. Des

*attributs* permettent de connaître les valeurs effectives des bornes. Voici un exemple de sous-programme effectuant le produit de deux matrices :

```
Matrice_Error : exception;

function "*" (X, Y : Matrice) return Matrice is
   Produit : Matrice (X'Range(1), Y'Range(2));
begin
   if X'First(2) /= Y'First(1) or X'Last(2) /= Y'Last(1)
   then
      raise Matrice_Error;
   end if;

   for I in X'Range (1) loop
      for j in Y'Range (2) loop
         Produit(I,J) := 0.0;
         for K in X'Range (2) loop
            Produit(I,J) := Produit(I,J) + X(I,K) * Y(K,J);
         end loop;
      end loop;
   end loop;

   return Produit;
end "*";
```

Ce sous-programme effectue le produit de deux objets de type `Matrice` dès lors que les bornes de la deuxième dimension de la première matrice correspondent aux bornes de la première dimension de la deuxième matrice (sinon, il y a levée de l'exception `Matrice_Error`). Grâce aux attributs, le compilateur peut déterminer qu'aucun débordement d'indice n'est possible ; l'optimiseur supprimera tous les contrôles inutiles, et le gain de sécurité apporté par Ada ne se traduira par aucune perte de performances. La possibilité de définir des opérateurs arithmétiques entre types quelconques (y compris tableaux), ainsi que de définir des fonctions renvoyant des types structurés, permet de simplifier considérablement l'utilisation. On pourra écrire :

```
    Mat_1 : Matrice (1..3, 1..5);
    Mat_2 : Matrice (1..5, 1..2);
    Mat_3 : Matrice (1..3, 1..2);
begin
    -- Initialisation de Mat_1 et Mat_2
    Mat_3 := Mat_1 * Mat_2;
```

## *2.4 .3 Types paramétrables*

Ada offre un mécanisme original et extrêmement puissant pour paramétrer un type article de la même façon que l'on peut fournir des paramètres pour diriger le comportement des sous-programmes. Ces paramètres s'appellent *discriminants* et doivent être d'un type discret (entier ou énumératif), ainsi que d'un type accès (pointeur) en Ada 95. Les discriminants peuvent servir à contrôler la structure de l'article ; ils peuvent être lus, mais ne peuvent être modifiés. Voici par exemple comment représenter des matrices carrées :

```
type Matrice_Carrée (Taille : Positive) is
   record
      La_Matrice : Matrice(1..Taille, 1..Taille);
   end record;
```

Une *contrainte de discriminant* peut être dynamique ; on peut ainsi lire la taille de la matrice souhaitée :

```
Get (N);
declare
    Ma_Matrice : Matrice_Carrée (N);
begin
    -- traitements sur la matrice
end;
```

Les discriminants peuvent aussi servir à faire des structures dont les composants présents dépendent de leur valeur :

```
type Options is (Lettres, Sciences, Techno);
type Note is delta 0.1 digits 3 range 0.00 .. 20.0;
type Bulletin (L_Option : Options) is
   record
       Français      : Note;
       Mathématiques : Note;
       case L_Option is
       when Lettres =>
           Latin : Note;
       when Sciences =>
           Physique : Note;
       when Techno =>
           Dessin_Industriel : Note;
       end case;
   end record;
```

## *2.4 .4 Types dérivés*

Plutôt que de redéfinir un type avec toutes ses propriétés, on peut créer un nouveau type à partir d'un type existant dont il héritera les propriétés (domaine de définition et opérations). Un tel type est appelé type *dérivé*. Bien sûr, il s'agit d'un type différent, et donc *incompatible* avec le type d'origine.

```
   type Mètres is digits 5 range 0.0 .. 1.0E15;
   type Yards  is new Mètres;
   M : Mètres;
   Y : Yards;
begin
   Y := Y + M;     -- Interdit ! On ne peut additionner
                          -- des Yards et des Mètres.
```

Des types dérivés l'un de l'autre, ou dérivés d'un même ancêtre, sont convertibles entre eux. Ce mécanisme a été enrichi en Ada 95 pour fournir le mécanisme de base de la programmation orientée objet.

## *2.4 .5 Résumé de l'arborescence des types*

Les différentes formes de types et leurs relations sont schématisées à la figure 2 (nous avons inclus quelques types apportés par Ada 95 qui seront présentés dans la prochaine partie).

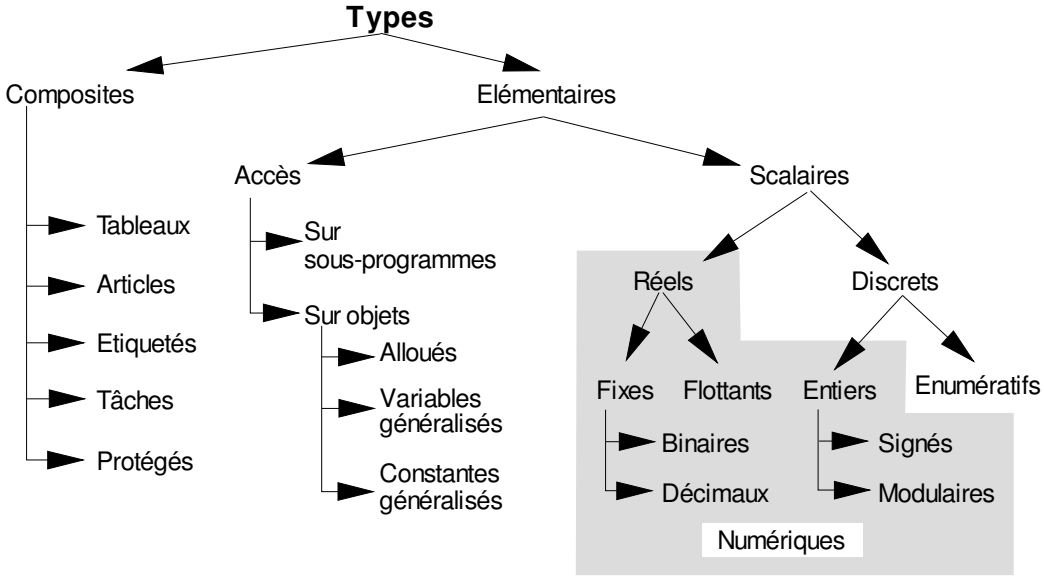


**Figure 2 :** Arborescence des types

Ada peut certainement se prévaloir d'être le langage le plus riche en matière de typage !

## 2.5 Paquetages

Une notion centrale en Ada est celle de paquetage (*package*). Le paquetage permet de regrouper dans une seule entité des types de données, des objets (variables ou constantes) et des sous-programmes (procédures et fonctions) manipulant ces objets. Un paquetage est constitué de deux parties, compilables séparément : la *spécification* et le *corps*. La spécification comporte une partie *visible* comprenant les informations utilisables à l'extérieur du paquetage, et une partie *privée* qui regroupe les détails d'implémentation auxquels les utilisateurs du paquetage n'ont pas accès (mais qui sont nécessaires au compilateur) ; ces deux parties sont séparées par le mot clé **`private`**. L'exemple ci-dessous montre la spécification d'un paquetage de gestion de nombres complexes.

```
package Nombres_Complexes is
   type Complex is private;
   I : constant Complex;

   function "+" (X, Y : Complex) return Complex;
   function "-" (X, Y : Complex) return Complex;
   function "*" (X, Y : Complex) return Complex;
   function "/" (X, Y : Complex) return Complex;

   function CMPLX (X, Y : Float) return Complex;
   function POLAR (X, Y : Float) return Complex;

private     -- L'utilisateur n'a pas accès aux déclarations
           -- ci-dessous
   type Complex is
      record
         Reel : Float;
         Imag : Float;
      end record;
   I : constant Complex := (0.0, 1.0);
end Nombres_Complexes;
```

Le type `Complex` est appelé *type privé* : dans la partie visible, on ne fait qu'annoncer sa présence, la définition complète étant fournie dans la partie privée (après le mot **`private`**) ; l'utilisateur ne peut faire usage des propriétés de l'implémentation. Si on décide par la suite de représenter les nombres complexes sous forme polaire plutôt que cartésienne, les règles du langage garantissent qu'aucune application utilisant ce paquetage n'aura à être modifiée. Un type privé dispose de l'affectation et de la comparaison d'égalité (et d'inégalité). Il est possible de supprimer ces propriétés en déclarant le type comme **`limited private`** ; le type est alors un type *limité*.

Une spécification ne définit que l'*interface externe* du paquetage ; on doit fournir dans le *corps* du paquetage l'implémentation des fonctionnalités offertes. Celui de l'exemple ci-dessus aurait la forme suivante :

```
package body Nombres_Complexes is
    function "+" (X, Y : Complex) return Complex is
    begin
        -- Implémentation de la fonction "+"
    end "+";

    -- De même pour les autres fonctions fournies
end Nombres_Complexes;
```

Les paquetages peuvent être compilés séparément, et seule leur spécification est nécessaire pour les utiliser. On réalise ainsi une séparation complète entre les spécifications d'une unité fonctionnelle et son implémentation. De plus, les règles très rigoureuses de recompilation garantissent, d'une part, qu'il n'est pas possible d'utiliser une information cachée d'un paquetage, et d'autre part que la recompilation des unités qui utilisaient une spécification est obligatoire lorsque celle-ci est modifiée : le langage garantit donc la cohérence globale des différentes unités constituant un programme.

## 2.6 Unités génériques

Les unités génériques sont des unités paramétrables permettant de définir un algorithme indépendamment des types d'objet manipulés. Voici par exemple une procédure générique qui intervertirait ses deux arguments :

```
generic
   type Elem is private;
procedure Permuter (X, Y : in out Elem);

procedure Permuter (X, Y : in out Elem) is
   Temp : Elem;
begin
   Temp := X;
   X    := Y;
   Y    := Temp;
end Permuter;
```

La déclaration du type `Elem` comme **`private`** signifie que n'importe quel type pour lequel l'affectation (et la comparaison d'égalité) est définie peut faire l'affaire. L'unité générique n'est pas utilisable par elle-même : ce n'est qu'un modèle (on dit parfois un moule) dont on doit faire une *instanciation* pour obtenir l'unité réelle. L'instanciation précise les valeurs des paramètres génériques. Ainsi, pour obtenir une procédure permettant d'échanger deux variables de type `Age`, nous écrirons :

```
procedure Permuter_Age is new Permuter (Age);
          ...
       Permuter_Age (Mon_Age, Son_Age);
          ...
```

L'exemple suivant donne la spécification d'une unité générique de tri de tableau :

```
generic
   type Index     is (<>);
   type Composant is private;
   type Tableau   is array (Index) of Composant;
   with function "<"(X,Y : Composant) return Boolean is <>;
procedure Tri (A_Trier : in out Tableau);
```

Les paramètres génériques sont le type d'indice du tableau, le type de composant, le type du tableau lui-même, et une fonction de comparaison. Cette fonction de comparaison est munie d'une valeur par défaut (clause **`is`** `<>`), signifiant que si l'utilisateur ne fournit pas de valeur, la comparaison prédéfinie (si elle existe) doit être utilisée. Les avantages de cette démarche sont évidents : à partir d'un algorithme écrit une fois pour toutes, et que l'on pourra optimiser soigneusement, il est possible d'obtenir instantanément les fonctions de tri sur n'importe quel type de données, avec n'importe quel critère de comparaison. On obtient par exemple ainsi une procédure triant un tableau de nombres flottants dans l'ordre croissant :

```
type Int is range 1..10;
type Arr is array (Int) of Float;
procedure Tri_Ascendant is new Tri (Int, Float, Arr);
```

Si l'on souhaite trier dans le sens décroissant, il suffit d'inverser le critère de comparaison :

```
procedure Tri_Descendant is new Tri (Index     => Int,
                                     Composant => Float,
                                     Tableau   => Arr,
                                     "<"       => ">");
```

Pour cette instanciation, nous avons utilisé une association *nommée* de paramètres : la correspondance entre paramètres formels et réels se fait au moyen du nom du formel, et non par la position. Ceci améliore considérablement la lisibilité et évite bien des erreurs. L'association nommée est également possible (et même recommandée) pour les appels de sous-programmes.

## 2.7 Parallélisme

Le langage Ada permet de définir des tâches, qui sont des unités de programme s'exécutant en parallèle. Les tâches sont des objets appartenant à des *types tâche* ; elles peuvent donc être membres de structures de données : on peut ainsi définir des tableaux de tâches, des pointeurs sur des tâches, etc.

Les tâches se synchronisent et échangent des informations au moyen d'un mécanisme unique appelé *rendez-vous*. Une tâche dite serveuse déclare des points d'entrée, dont la spécification ressemble à une spécification de procédure. Une instruction spéciale, **`accept`**, lui permet d'accepter un appel du point d'entrée. Une autre tâche peut appeler le point d'entrée à tout moment. Une tâche acceptant une entrée, ou une tâche appelant un point d'entrée, est suspendue jusqu'à ce que son partenaire ait effectué l'instruction complémentaire. Le rendez-vous a alors lieu, la communication s'établissant par échange de paramètres comme pour un appel de procédure. Une fois le rendez-vous terminé, les deux tâches repartent en parallèle. D'autres formes syntaxiques permettent de raffiner ce mécanisme, en permettant l'attente multiple d'un serveur sur plusieurs points d'entrée à la fois, l'attente avec temporisation (*time out*), l'acceptation conditionnelle, ou la possibilité de terminer automatiquement la tâche si le serveur n'est plus nécessaire.

Ada étant un langage «temps réel», il est possible de suspendre une tâche pendant un certain intervalle de temps et d'accéder à l'heure absolue. Le paquetage Calendar fournit la définition d'un type «temps» (`Time`) et d'une fonction renvoyant l'heure courante. Il est possible de mettre une tâche en attente au moyen de l'instruction **`delay`** :

```
delay 3.0;          -- Attente de 3s.
```

Ada 95 a rajouté une instruction **`delay until`** permettant d'attendre jusqu'à une heure absolue.

Voici un exemple de tâche simple permettant de tamponner la transmission d'un caractère entre une têche productrice et une tâche consommatrice. La spécification annonce que la tâche fournit deux services (`Lire` et `Ecrire`) :

```
task Tampon is
   entry Lire   (C : out Character);
   entry Ecrire (C : in  Character);
end Tampon;
```

Le corps de la tâche décrit l'algorithme utilisé :

```
task body Tampon is
   Occupe : Boolean := False;
   Valeur : Character;
begin
   loop
      select
      when not Occupe =>
         accept Ecrire (C : in Character) do
            Valeur := C;
         end Ecrire;
         Occupe := True;
      or when Occupe =>
         accept Lire (C : out Character) do
            C := Valeur;
         end Lire;
         Occupe := False;
      or terminate;
      end select;
   end loop;
end Tampon;
```

La tâche boucle sur une attente multiple (instruction **`select`**) de ses entrées `Lire` et `Ecrire`. Ces entrées sont munies de *gardes* (clauses **`when`**) qui feront que l'on n'acceptera de servir l'entrée `Lire` que s'il y a effectivement un caractère dans le tampon, et inversement que l'on n'acceptera `Ecrire` que si le tampon est vide. La clause **`terminate`** assure la terminaison automatique de la tâche lorsque plus aucun client potentiel n'existe.

## 2.8 Utilisation de bas niveau

Ada étant un langage également destiné à la programmation des systèmes, il permet facilement d'accéder au bas niveau : on peut forcer la représentation machine des structures abstraites, spécifier des traitements d'interruptions, inhiber les vérifications de type, et même inclure du code machine. Toutefois, des précautions ont été prises pour conserver la sécurité même avec ce genre de manipulations.

### *2.8 .1 Clauses de représentation*

Il est possible de spécifier au bit près la représentation des types de données. Normalement, la représentation interne est choisie par le compilateur, et le langage garantit la totale indépendance de l'utilisation des types vis-à-vis de leur représentation. Cependant, il faut parfois imposer une représentation, notamment lors d'interfaçages avec le matériel ou avec des sous-programmes écrits dans d'autres langages. On donne alors au programmeur une vision de haut niveau, tout en imposant la représentation de bas niveau. On peut ainsi décrire une cellule de mémoire écran d'IBM-PC :

```
type Couleurs is
 (Noir,         Bleu,          Vert,       Cyan,
  Rouge,        Magenta,       Marron,     Gris,
  Gris_sombre,  Bleu_clair,    Vert_clair, Cyan_clair,
  Rouge_clair,  Magenta_clair, Jaune,      Blanc);
-- Représentation des énumératifs :
for Couleurs use (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);

subtype Couleurs_Fond is Couleurs range Noir .. Gris;

type Animation is (Fixe, Clignotant);
for  Animation use (0, 1);

type Attribut is
   record
     Clignotement : Animation;
     Fond         : Couleurs_Fond;
     Devant       : Couleurs;
   end record;

-- Représentation du type article : at <mot> range <bits>
for Attribut use
   record
     Clignotement   at 0 range 0..0;
     Fond           at 0 range 1..3;
     Devant         at 0 range 4..7;
   end record;
```

Si l'on veut changer la valeur du bit `Clignotement` d'un attribut, on écrira simplement :

```
A.Clignotement := Fixe;
```

ce qui aura bien pour effet de changer la valeur du bit considéré. On voit qu'ainsi, il n'est plus jamais besoin de calculer des masques, décalages, etc., pour intervenir à bas niveau. En cas de modification de la représentation interne des données, aucune modification du code n'est nécessaire.

Les règles du langage ne sont nullement changées par la présence de clauses de représentation, ce qui constitue un outil très puissant. Supposons que nous ayons besoin, par exemple lors de l'écriture d'une interface, de deux représentations physiques différentes d'un même type de données. Il suffit de faire un type dérivé muni de clauses différentes de celles d'origine. Les types dérivés étant convertibles entre eux, le changement de représentation sera pris en charge par le compilateur, comme dans l'exemple suivant :

```
   type Format_1 is ...
   for  Format_1 use ...   -- Représentation de Format_1

   type Format_2 is new Format_1;
   for  Format_2 use...    --  Représentation de Format_2

   V1 : Format_1;
   V2 : Format_2;
begin
   ...
   V1 := Format_1(V2); -- Changement de représentation
   V2 := Format_2(V1); -- effectué par le compilateur
```

### *2.8 .2 Détypage*

Les conversions précédentes étaient des conversions de haut niveau, respectant la sémantique des types. Il est parfois nécessaire d'effectuer des conversions plus directes (*type cast*), notamment lorsqu'une même donnée doit être vue selon deux niveaux d'abstraction différents. Par exemple, des caractères sont des entités propres, qui ne sont pas considérées comme des valeurs numériques. On ne peut donc additionner des caractères, ce qui n'aurait aucun sens... sauf si l'on veut calculer un CRC[1]. On peut alors forcer une vue d'un caractère sous forme numérique grâce à l'instanciation de la fonction générique `Unchecked_Conversion` :

```
   type Octet is mod 256;  -- type modulaire
   for  Octet'Size use 8;  -- représenté sur 8 bit

   function Char_to_Octet is
       new Unchecked_Conversion (Character, Octet);

   CRC : Octet;
   C   : Character;
begin
   ...
   CRC := CRC * 2 + Char_to_Octet (C);
```

La sécurité des types n'est pas mise en cause, car les règles du langage font que l'utilisation de cette fonction n'est possible qu'à la condition de mettre en tête de module la clause **`with`** `Unchecked_Conversion;`. Ceci a deux conséquences importantes :

Celui qui relit le programme est immédiatement prévenu de la présence dans le module de fonctionnalités potentiellement dangereuses ou non portables. Réciproquement, l'*absence* de cette clause signifie qu'il ne peut y avoir de forçage de type.

Si les règles de codage du projet imposent une justification spéciale pour l'utilisation d'`Unchecked_Conversion`, le chef de projet peut s'assurer qu'aucune utilisation non autorisée n'en a été faite simplement en interrogeant son gestionnaire de bibliothèque, sans avoir à inspecter manuellement tout le code : la règle peut donc être commodément vérifiée. Ce point est fondamental : une règle qui serait invérifiable en pratique est lettre morte dans un projet quelque peu important.

## 2.9 Compilation séparée

Favoriser la modularité des programmes était une contrainte forte dans la définition du langage. Il fallait faire en sorte que l'on n'ait jamais intérêt à faire des modules trop gros, donc difficiles à maintenir.

Ceci a été obtenu au moyen de la notion de bibliothèque de programme. Une bibliothèque Ada est différente de ce que l'on appelle «bibliothèque» dans d'autres langages de programmation ; c'est

[1] *Cyclic Redundancy Code*, code utilisé pour vérifier la validité de données, et fondé sur une valeur clé obtenue par des manipulations arithmétiques sur les codes des caractères.

une sorte de base de données conservant les produits des compilations : codes objets bien entendu, mais également informations sémantiques sur les modules (résultats de la compilation des spécifications), et aussi souvent informations annexes : dates de première et dernière recompilations, copie des sources, noms des fichiers d'origine... La compilation d'une unité Ada consiste à la rajouter à cette bibliothèque, tout en mémorisant ses liens de dépendance (clauses **`with`**), ainsi que toutes les caractéristiques des entités exportées par les spécifications.

Ceci permet de garantir des contrôles aussi stricts, notamment en matière de typage, pour des unités compilées séparément que pour des unités compilées en même temps. Le compilateur refusera de construire un programme incohérent. On obtient ainsi le bénéfice des systèmes de contrôle externes utilisés dans d'autres langages (*makefile*), mais gérés automatiquement par le compilateur. On dit que le langage Ada fournit la possibilité de compilations *séparées*, mais non *indépendantes*.

Le processus de construction d'un programme en Ada est donc très différent de ce qui se passe dans les autres langages. On construit tout d'abord (avec un utilitaire fourni avec le compilateur) une bibliothèque vide. Puis l'on ajoute, module par module, des unités à cette bibliothèque. On bâtit ainsi progressivement le programme, en assemblant les différents morceaux qui le constituent comme les pièces d'un puzzle. On commencera par mettre dans la bibliothèque les spécifications communes à l'ensemble du projet, puis chaque programmeur codera les implémentations de ces spécifications ; le langage garantira la cohérence des implémentations par rapport aux spécifications aussi bien que celle des appels par l'utilisateur. De plus, tous les compilateurs possèdent des raffinements personnels améliorant ce modèle : bibliothèques liées (bibliothèques physiquement différentes, mais logiquement reliées), sous-bibliothèques fournissant des règles de visibilité des modules (un module est recherché dans la sous-bibliothèque avant d'être recherché dans la bibliothèque principale), familles de bibliothèques...

Ada 95 a quelque peu assoupli le modèle de bibliothèque, notamment pour diminuer les recompilations et permettre d'autres modèles de gestion de modules. Mais le principal a été conservé : il est impossible de construire un programme incohérent.

## 2.10 Exercices

1. Prendre n'importe quel programme d'exercice en C ou en Pascal, et le traduire directement en Ada. Reprendre le programme précédent et le recoder en utilisant au mieux les fonctionnalités d'Ada non présentes dans les autres langages. Comparer la puissance d'expression des deux versions.
2. Comparer les contrôles de type offerts par Ada avec ceux d'autres langages. Pourquoi aucun autre langage n'a-t-il été aussi loin dans les contrôles ?
3. Comparer les mécanismes de compilation séparée de divers langages avec ceux d'Ada.

# 3
# Les nouveautés d'Ada 95

Les nouveautés d'Ada 95 sont organisées autour de quelques grands thèmes : extensions pour la programmation orientée objet, nouvelles fonctionnalités pour la programmation système et le temps réel, meilleur support de très gros projets... et correction des problèmes connus d'Ada 83. D'autre part, la nouvelle norme est en deux parties, séparant le langage proprement dit de la bibliothèque prédéfinie. Les besoins spécifiques de domaines particuliers sont pris en compte dans des *annexes* facultatives.

## 3.1 Perfectionnements généraux

Le jeu de caractères de base est maintenant Latin 1 (8 bits) et non plus ASCII (7 bits). Le compilateur peut offrir des options permettant de gérer d'autres jeux de caractères, tels que Latin 2 ou IBM/PC. Les lettres accentuées sont autorisées même dans les identificateurs.

Ada 95 fournit un support direct de l'arithmétique décimale, exigence indispensable à son succès en informatique de gestion. Les types décimaux sont des types point-fixe, mais à base décimale au lieu de binaire. On notera également l'introduction de types entiers modulaires (les opérations «bouclent» au lieu de déborder) et l'ajustement du modèle de l'arithmétique aux évolutions du matériel (arithmétique IEEE notamment).

De nouvelles formes de paramètres génériques (passage de paquetage générique en paramètre formel générique) facilitent la combinaison de fonctionnalités, en supprimant une cause majeure de duplication de code. Le plus simple est de l'illustrer par un exemple :

```
generic
   type Real is digits <>
package Nombres_Complexes is
   ...
end Nombres_Complexes;

generic
   with Package_Complexe is new Nombres_Complexes(<>);
package Matrices_Complexes is
   ...
end Matrices_Complexes;
```

Lors de l'instanciation du paquetage `Matrices_Complexes`, on passera n'importe quel paquetage obtenu par instanciation du paquetage `Nombres_Complexes` :

```
-- Simple précision
package Complexes_Simples is new Nombres_Complexes(Float);

package Matrices_Complexes_Simples is
   new Matrices_Complexes (Complexes_Simples);

-- Double précision
package Complexes_Doubles is
   new Nombres_Complexes(Long_Float);
```

```
package Matrices_Complexes_Doubles is
   new Matrices_Complexes (Complexes_Doubles);
```

Enfin de nouvelles formes de pointeurs sont autorisées : pointeurs sur objets globaux et pointeurs sur sous-programmes facilitent les interfaces avec des systèmes extérieurs, systèmes de fenêtrage notamment. Des contrôles à la compilation comme à l'exécution garantissent l'impossibilité de référencer un objet qui n'existerait plus, comme un objet local à un sous-programme après la sortie de celui-ci par exemple.

## 3.2 Extensions pour la programmation orientée objet

De nombreux utilisateurs critiquaient le manque de support d'Ada pour le paradigme d'héritage. Ce point était cependant délicat, car l'héritage s'oppose à certains des principes d'Ada, entraînant notamment un affaiblissement du typage (certaines cohérences de type ne peuvent être vérifiées qu'à l'exécution, alors qu'Ada les assure traditionnellement lors de la compilation). Enfin, la programmation orientée objet conduit souvent à une multiplication de pointeurs et d'allocations dynamiques cachées. Or le monde du temps réel, particulièrement pour les applications devant fonctionner 24 heures sur 24, est très hostile à l'allocation dynamique à cause des risques de fragmentation mémoire et de perte d'espace qu'elle induit. Il fallait donc introduire un mécanisme permettant d'ouvrir Ada à de nouvelles applications, sans lui faire perdre ses principes fondamentaux : primauté de la facilité de maintenance sur la facilité de conception, contrôle précis de l'allocation et vérifications strictes lors de la compilation.

La solution retenue est fondée sur une extension naturelle du mécanisme des types dérivés : les types étiquetés (*tagged*). Nous ne présenterons ici que quelques lignes directrices, car nous en détaillerons l'utilisation dans la deuxième partie de ce livre. Il s'agit de types munis d'un attribut supplémentaire caché permettant d'identifier le type d'un objet à l'intérieur de sa *classe* (ensemble des types dérivés directement ou indirectement d'un même ancêtre). Ces types offrent trois nouvelles possibilités :

*L'extension de types*. Il est possible, lorsque le type parent d'un type dérivé est un type étiqueté, d'ajouter des champs supplémentaires lors de la dérivation. Les opérations du parent restent disponibles, mais n'opèrent que sur la partie commune.
*Le polymorphisme*. Des sous-programmes peuvent posséder des paramètres se référant à la *classe* d'un type étiqueté : ils sont applicables alors non seulement au type d'origine, mais également à tous les types qui en sont dérivés. Il est possible de définir des pointeurs sur classe, pouvant désigner n'importe quel objet appartenant à la classe. Cette dernière forme de polymorphisme permet l'implémentation des mécanismes habituels de la programmation orientée objet, tout en conservant le contrôle explicite des pointeurs.

- *Les liaisons dynamiques*. Si on appelle un sous-programme en lui passant un paramètre de type *classe,* le sous-programme appelé n'est pas connu à la compilation. C'est l'étiquette (*tag*) de l'objet réel qui détermine à l'exécution le sous-programme effectivement appelé. Nous reviendrons sur ce mécanisme lors de l'étude des méthodes par classification, mais remarquons qu'il n'exige *pas* de pointeurs : contrairement à nombre de langages orientés objet[1], on peut effectuer des liaisons dynamiques sans imposer de pointeurs, cachés ou non.

Généralement, les langages orientés objet fournissent toutes ces possibilités «en bloc». En Ada 95, l'utilisateur ne demandera que les fonctionnalités dont il a besoin, et ne paiera que le prix de ce qu'il utilise. Là où l'extensibilité et la dynamicité sont plus importants que la sécurité, Ada offrira toutes les fonctionnalités nécessaires. En revanche, un concepteur qui ne souhaiterait *pas* utiliser ces nouvelles fonctionnalités conservera le même niveau de contrôle de type et la même efficacité du code généré qu'avec Ada 83.

[1] Dont Eiffel, Java, C# et C++. Noter qu'en Java, tout est pointeur, ceux-ci n'apparaîssent donc plus explicitement. Cela ne signifie pas qu'ils ne sont pas là!

On notera que l'héritage multiple est obtenu au moyen des autres blocs de base du langage plutôt que par une construction syntaxique spéciale. Ceci n'est pas une omission, mais un choix délibéré : le risque de complexité lié à ce mécanisme aurait été trop important, et les autres perfectionnements, notamment concernant les génériques, ont permis de construire tous les cas d'héritage multiple que l'on peut rencontrer dans la littérature.

## 3.3 Nouvelles facilités pour la gestion de très grosses applications

Une grande force du langage Ada est son contrôle automatique de configuration : le compilateur vérifie toutes les dépendances entre modules compilés séparément, et aucun module ne peut référencer un module périmé. Dans de grosses applications, ceci peut conduire à des recompilations massives, d'autant plus regrettables que la modification n'est bien souvent due qu'à l'ajout de fonctionnalités pour un besoin nouveau, ce qui ne perturbait pas en fait les anciens utilisateurs. D'autre part, il est souvent nécessaire de regrouper logiquement des paquetages concourant à un but commun ; un tel ensemble est appelé «sous-système» dans la terminologie de Booch. Des environnements de programmation sont actuellement capables de gérer de tels sous-systèmes, mais il n'y avait pas de support direct au niveau du langage.

Ada 95 introduit la notion de paquetages hiérarchiques. Il est possible de créer des paquetages «enfants» qui rajoutent des fonctionnalités à des paquetages existants sans avoir à toucher à leurs «parents», donc sans recompilation des utilisateurs de ces parents. Ceci permet une meilleure séparation en fonctionnalités «principales» et fonctionnalités «annexes», facilitant l'utilisation aussi bien que la maintenance. Par exemple, lorsque l'on fait un type de donnée abstrait, on ne souhaite généralement pas mélanger les propriétés de base avec les entrées-sorties. Ceci peut s'obtenir de la façon suivante :

```
package Nombres_Complexes is -- encore lui !
   type Complex is private;

   -- Opérations sur les nombres complexes
private
   type Complex is ...
end Nombres_Complexes;

package Nombres_Complexes.IO is -- Un enfant
   -- Entrées-sorties sur les complexes
end Nombres_Complexes.IO;
```

Le corps du paquetage enfant a accès à la partie privée du parent, autorisant donc des implémentations efficaces fondées directement sur la représentation interne du type abstrait. De plus, il est possible de définir des enfants «privés», non accessibles depuis les unités extérieures à la famille. On obtient ainsi des contrôles supplémentaires, puisque seul le parent est visible de l'extérieur et constitue le point d'entrée obligé du sous-système dont les autres éléments sont cachés.

## 3.4 Nouvelles facilités pour le temps réel et la programmation système

### *3.4 .1 Parallélisme et synchronisation*

Un certain nombre de perfectionnements ont été apportés à la demande des utilisateurs. Si, comme en Ada 83, chaque tâche peut être munie d'une *priorité d'exécution*, celle-ci n'a plus à être statique. La priorité initiale est définie par une déclaration dans la tâche, et un paquetage prédéfini fournit les services permettant de changer dynamiquement cette priorité.

Les types tâches peuvent être munis de discriminants, qui servent à paramétrer leur comportement. On peut transmettre ainsi à la tâche des valeurs simples (notamment l'espace

mémoire requis ou la priorité d'exécution), mais également des pointeurs sur des blocs de paramètres, sur d'autres tâches...

Ada 95 a rajouté une nouvelle forme d'appel d'entrée, le *select asynchrone*, qui permet à un client de continuer à travailler pendant qu'une demande de rendez-vous est active, ou après avoir armé un délai. Si la demande est servie, ou si le délai expire, avant la fin du traitement associé, celui-ci est avorté ; sinon, c'est la demande qui est annulée. Les deux formes de select asynchrone sont montrées ci-dessous :

```
select                              select
   appel d'entrée                      delay [until] ...;
then abort                          then abort
   instructions                        instructions
end select;                         end select;
```

La synchronisation et la communication entre tâches étaient fondées en Ada 83 sur un mécanisme unique : le rendez-vous. Pour protéger une variable contre des accès simultanés, il fallait l'enfermer dans une tâche «gardienne». Ce mécanisme fonctionnait bien ; il conduisait cependant à une multiplication des petites tâches, et la communauté temps réel souhaitait un moyen simple, plus léger et très performant de protéger des variables.

La solution proposée par Ada 95 s'appelle les *articles protégés*. Il s'agit d'objets (ou de types d'objets) auxquels sont associés des sous-programmes spéciaux, dont on garantit l'exclusivité d'accès. Ils s'apparentent donc aux moniteurs de Hoare, mais en plus perfectionné (possibilité de mettre des barrières notamment). Voici un tel objet protégé :

```
protected Barrière is
   entry Passer;
   procedure Ouvrir;
   function En_Attente return Natural;
private
   Ouverte : Boolean := False;
end Barrière;

protected body Barrière is
   entry Passer when Ouverte is
   begin
      if Passer'Count = 0 then  -- Le dernier referme
        Ouverte := False;       -- la barrière.
      end if;
   end Passer;

procedure Ouvrir is
   begin
      if En_attente > 0 then
        Ouverte := True;
      end if;
   end Ouvrir;

   function En_Attente return Natural is
   begin
      return Passer'Count;
   end En_Attente;
end Barrière;
```

Une tâche appelant l'entrée `Passer` est bloquée jusqu'à ce qu'une autre tâche appelle la procédure `Ouvrir` ; toutes les tâches en attente sont alors libérées. La fonction `En_Attente` permet de connaître le nombre de tâches en attente. Pour bien comprendre cet exemple, il faut savoir que :

Plusieurs tâches peuvent appeler simultanément une fonction de l'objet protégé, ou (exclusivement) une seule tâche peut exécuter une procédure ou une entrée (algorithme des lecteurs/écrivains).

- Des tâches susceptibles de s'exécuter parce que la condition associée à une entrée est devenue vraie s'exécutent toujours *avant* toute autre tâche appelant une opération de l'objet protégé.

Par conséquent, les tâches libérées lorsqu'un appel à `Ouvrir` a remis la variable `Ouverte` à `True` reprendront la main avant toute tâche non bloquée sur l'entrée. Une tâche qui appellerait l'entrée `Passer` juste après qu'une autre a appelé `Ouvrir` se bloquera donc bien (après que toutes les tâches en attente auront été libérées) ; aucune condition de course n'est possible.

Il est possible (avec les implémentations supportant l'annexe «temps réel») d'associer une priorité à un objet protégé, qui est une priorité «plafond» : toute opération protégée s'effectuera à ce niveau de priorité, et il est interdit à une tâche de niveau de priorité supérieure d'appeler ces opérations protégées. Ce mécanisme permet de garantir l'absence de phénomènes d'inversion de priorité[1] ; de plus, sur un système monoprocesseur, il est suffisant pour garantir l'exclusivité d'accès aux objets protégés, sans nécessiter de sémaphore supplémentaire, ce qui rend les objets protégés particulièrement efficaces.

Retenons sur le plan philosophique l'introduction dans Ada d'un mécanisme de *synchronisation par les données*, ce qui représente une nouveauté importante par rapport à la version précédente du langage.

Une autre fonctionnalité importante est l'instruction **`requeue`** qui permet, pendant le traitement d'un **`accept`** de tâche ou d'un appel d'une entrée d'objet protégé, de renvoyer le client en attente sur une autre queue. Par exemple :

```
type Urgence is (Lettre, Télégramme);
   ...
accept Envoyer (Quoi : Urgence; Message : String) do
   if Quoi = Télégramme then
       -- On s'en occupe tout de suite
   else
       requeue Voir_Plus_Tard (Quoi, Message);
   end if;
end Envoyer;
```

Ici une tâche serveuse de messages dispose d'un point d'entrée unique pour des messages urgents (les télégrammes) ou moins urgents (les lettres). Elle traite les télégrammes immédiatement, mais renvoie les lettres sur son entrée `Voir_Plus_Tard`, qu'elle ira traiter lorsqu'il n'y aura plus de messages en attente sur `Envoyer`.

## *3.4 .2 Gestion du temps*

S'il existe plusieurs bases de temps dans le système, l'implémentation a le droit de fournir d'autres types «temps» en plus du type prédéfini `Calendar.Time`. L'annexe «temps réel» impose la présence d'un temps «informatique», garanti monotone croissant[2] (dans le paquetage `Ada.Real_Time`). Il est possible de mettre une tâche en attente jusqu'à une heure *absolue* par l'instruction **`delay until`** :

```
delay until Time_Of((1995, 02, 15, 08*Heure+30*Minute));
```

L'attente absolue est exprimée dans un quelconque «type temps» (au moins `Calendar.Time`). Si l'implémentation en fournit plusieurs, c'est le type de l'argument qui déterminera l'horloge utilisée. Ce mécanisme permet de disposer d'instructions liées au temps portables, tout en bénéficiant de l'accès à des horloges spécifiques si la précision est plus importante que la portabilité.

---

[1] Cas qui se produit lorsqu'une tâche de haute priorité est bloquée par une de plus basse priorité à cause d'un accès à une ressource commune.

[2] Le temps par défaut peut correspondre à l'heure légale, et donc subir des sauts vers l'avant ou vers l'arrière lors des passages heure d'été / heure d'hiver.

### 3.4 .3 Contrôle de l'ordonnancement et gestion des queues

Normalement, l'ordonnancement des tâches est géré par l'exécutif fourni avec le compilateur. Ada 95 autorise plusieurs politiques d'ordonnancement, sélectionnables au moyen d'un pragma. Une seule est standardisée : `FIFO_Within_Priorities`, qui exprime que les tâches sont ordonnancées dans l'ordre où elles deviennent candidates, les tâches moins prioritaires ne s'exécutant que si aucune tâche plus prioritaire n'est susceptible de prendre le contrôle d'un processeur. Si cette politique est choisie, alors il est obligatoire de spécifier également le plafond de priorité pour les types protégés. Une implémentation est autorisée à fournir d'autres politiques d'ordonnancement.

Cependant, de nombreuses applications temps réel souhaitent un contrôle plus précis de l'ordonnancement. Dans ce but, l'annexe «programmation système» fournit des attributs et un paquetage permettant de récupérer le «Task_ID» interne d'une tâche (en particulier, de l'appelant dans un rendez-vous) et d'effectuer certaines opérations dessus. Un paquetage permet de créer des attributs de tâches, sortes de variables associées à chaque tâche (concrètement, cela signifie que l'on rajoute des variables utilisateur dans le bloc de contrôle de tâche). A ces fonctionnalités, l'annexe «temps réel» rajoute un paquetage de contrôle synchrone fournissant une sorte de sémaphore sur lequel les tâches peuvent venir se bloquer, et un paquetage de contrôle asynchrone permettant de suspendre et de relancer n'importe quelle tâche dont on connaît le «Task_ID». L'ensemble de ces fonctionnalités permet à l'utilisateur de contrôler entièrement l'ordonnancement des tâches ; en particulier, les attributs permettent de stocker dans chaque tâche les données nécessaires à la détermination de la prochaine tâche à ordonnancer.

En ce qui concerne la gestion des queues, les requêtes de rendez-vous et les files d'attente des entrées protégées sont traitées par défaut dans l'ordre premier arrivé - premier servi. L'annexe «temps réel» fournit un pragma (`Queing_Policy`) demandant de traiter les requêtes par ordre de priorité (les tâches plus prioritaires «doublent» les tâches moins prioritaires dans la queue) ; toute implémentation est autorisée à fournir des pragmas définissant d'autres algorithmes de mise en queue.

### 3.4 .4 Gestion des interruptions

Ada 83 faisait gérer les interruptions par des tâches, qui associaient des entrées à des interruptions physiques. A l'usage, ce mécanisme s'est révélé peu conforme à l'attente des utilisateurs temps réel. Il existe toujours en Ada 95 (ne serait-ce que par compatibilité ascendante), mais est classé «obsolète», ce qui signifie que l'on est censé lui préférer le nouveau mécanisme, l'attachement d'interruptions à des procédures protégées. Ce mécanisme fait partie de l'annexe «programmation système».

Il existe deux moyens d'attacher une procédure protégée à une interruption : statiquement, au moyen de pragmas, ou dynamiquement au moyen de sous-programmes fournis dans le paquetage `Ada.Interrupts`. Ce paquetage fournit également des fonctionnalités permettant de savoir si une procédure est actuellement attachée à une interruption, d'attacher une interruption à une procédure protégée en récupérant un pointeur sur le traitement d'interruption précédemment attaché (pour le rétablir plus tard), etc.

Le choix de représenter les traitements d'interruption par des procédures protégées plutôt que par des sous-programmes normaux permet d'exprimer de façon commode la nature non réentrante des traitements d'interruptions. Il permet également d'interdire l'utilisation d'instructions bloquantes, comme **`delay`** ou les appels d'entrées, depuis les gestionnaires d'interruption.

### *3.4 .5 Flots de données*

Un flot de données est un type descendant du type (étiqueté) `Root_Stream_Type` défini dans le paquetage `Ada.Streams`. Il représente une suite d'octets et est muni de procédures `Read` et `Write` permettant de lire et d'écrire des tableaux d'octets dans le flot. En plus de cette utilisation de premier niveau, l'intérêt des flots de données vient de ce que le langage fournit (sous forme d'attributs) des procédures permettant de lire et d'écrire *tout* type de donnée (non limité) depuis ou vers un flot de données. L'utilisateur peut redéfinir lui-même ces procédures s'il souhaite une représentation externe différente. Il peut aussi définir ces procédures pour des types limités. On peut ainsi passer de façon sûre et portable d'une vue abstraite d'un type de donnée vers un simple ensemble d'octets. Comme les fonctionnalités des flots permettent de convertir ces octets depuis/vers n'importe quel type, on peut créer des fichiers de données hétérogènes (contenant des valeurs provenant de types différents) sans perdre les avantages du typage fort. C'est en particulier indispensable pour transmettre des types de haut niveau sur un réseau, ou si l'on veut écrire des méthodes d'accès pour des fichiers indexés ou autre.

Concrètement, un flot peut stocker ses données en mémoire, dans un fichier, à travers un réseau ou par tout autre moyen. Le langage prédéfinit le paquetage `Ada.Streams.Stream_IO` qui fournit une implémentation des flots de données dans un fichier binaire, et `Ada.Text_IO.Text_Stream` qui utilise un fichier texte.

### *3.4 .6 Contrôle de l'allocation mémoire*

Le paquetage `System.Storage_Elements` fournit la notion de tableau d'octets (`Storage_Array`), ainsi que le type `Integer_Address,` qui représente une adresse sous forme de nombre entier, et est muni de fonctions de conversion sûres depuis/vers le type `System.Address`. Le paquetage `System.Address_To_Access_Conversion` procure des fonctionnalités de conversion (propres !) entre types accès et adresses. Il est donc possible d'accéder directement à la mémoire sans recourir à des utilisations douteuses (et souvent non portables) de `Unchecked_Conversion`.

Enfin le paquetage `System.Storage_Pools` fournit un type de données permettant à l'utilisateur de définir des «objets pools» et de les associer à des types accès. L'allocation et la désallocation dynamiques (opérations **`new`** et `Unchecked_Deallocation`) utilisent alors obligatoirement ces objets. L'utilisateur contrôle donc entièrement les algorithmes d'allocation/désallocation de mémoire dynamique, et peut garantir l'absence de fragmentation, le temps maximum d'une allocation dynamique, etc., indépendamment de la bibliothèque d'exécution (*run-time*) du compilateur.

## 3.5 Nouveaux paquetages prédéfinis

Ada 83 n'offrait que peu de paquetages prédéfinis, principalement des fonctionnalités d'entrées-sorties. Le langage était suffisamment puissant pour permettre à l'utilisateur d'écrire lui-même tout ce dont il avait besoin. A l'usage, il est apparu cependant que l'utilisateur aurait préféré trouver un environnement plus complet, plutôt que de devoir tout refaire lui-même. Ada 95 offre donc un plus grand nombre de paquetages, fournissant les fonctionnalités faisant partie généralement de la définition des autres langages.

Nous donnons ici un bref survol de cette nouvelle bibliothèque standard. Tous ces paquetages doivent être obligatoirement fournis par tout compilateur validé. Nous n'avons pas inclus les paquetages définis au niveau des annexes (cf. prochaine section). Remarquons que la plupart de ces paquetages n'ont nullement besoin de faire appel aux nouvelles fonctionnalités «95». Les personnes

programmant actuellement encore en Ada 83 peuvent donc les utiliser (il existe une implémentation dans le domaine public).

### *3.5 .1 Organisation générale*

Le concept d'unités hiérarchiques a permis d'organiser tous les paquetages de façon logique. Toutes les unités de compilation sont des enfants de `Standard`. Les éléments prédéfinis sont regroupés comme enfants de trois paquetages principaux : `Ada`, pour les unités indépendantes de l'implémentation, `Interfaces`, pour ce qui concerne la liaison avec d'autres langages, et `System` pour ce qui concerne les éléments dépendant de l'implémentation. `Text_IO` est ainsi devenu `Ada.Text_IO`. Des surnommages permettent d'assurer la compatibilité avec les programmes utilisant les anciens noms. La hiérarchie complète de l'environnement est résumée à la figure 3 (nous avons inclus les paquetages définis dans les annexes, décrits dans la prochaine section).

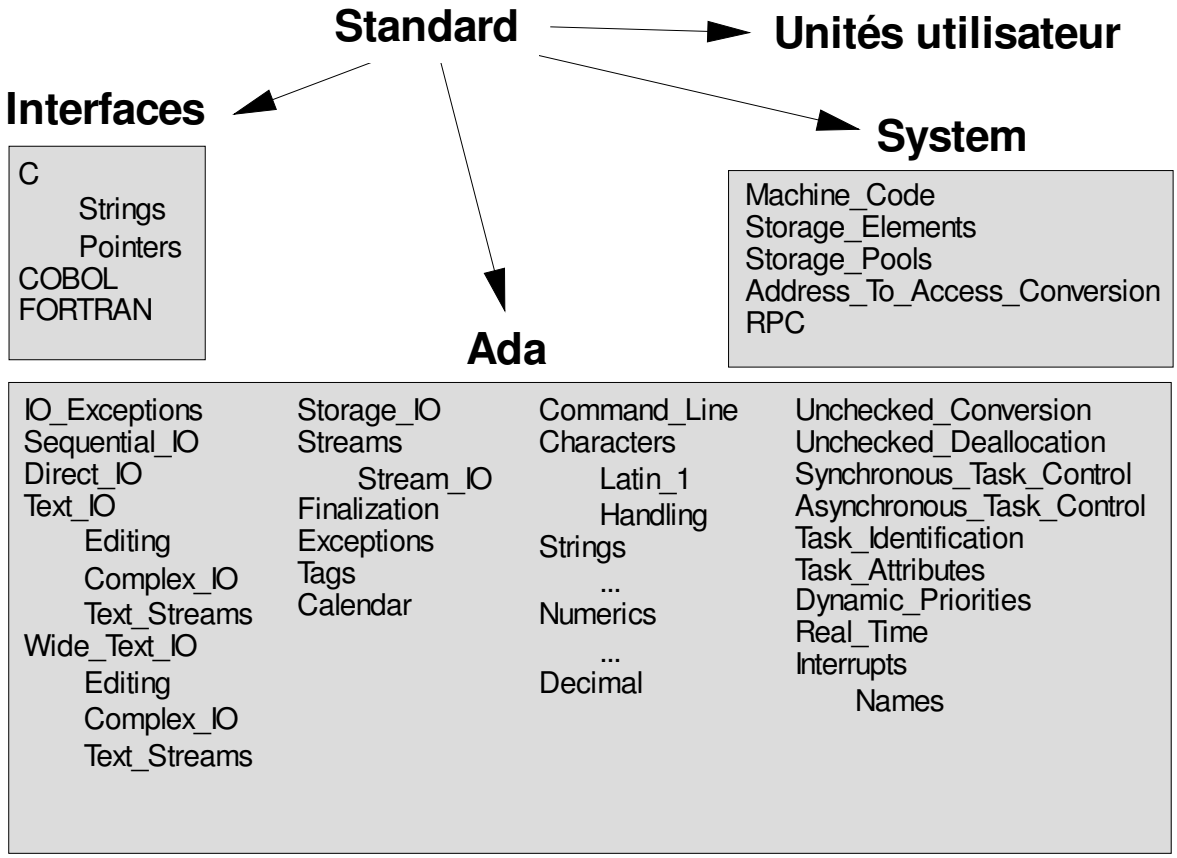


**Figure 3 :** Organisation de l'environnement standard

### *3.5 .2 Le paquetage Standard*

La seule modification apportée au paquetage `STANDARD` (qui définit les éléments directement utilisables par tout programme) est l'addition du type `Wide_Character` (caractères 16 bits) et d'un type `Wide_String` associé.

### *3.5 .3 Chaînes et caractères*

Le paquetage `Ada.Characters` définit des fonctions de test (`Is_control`, `Is_Upper`, `Is_Decimal_Digit`...) et de conversion (`To_Lower`, `To_Upper`...) sur les caractères, ainsi

qu'entre `Character` et `Wide_Character`. Son enfant `Ada.Characters.Latin_1` contient des définitions de constantes nommant tous les caractères du jeu Latin 1.

Plusieurs paquetages offrent des fonctionnalités assez évoluées de gestion de chaînes de caractères et d'ensembles de caractères. Ces fonctionnalités utilisent le type `String` (et `Wide_String`) ou des types définis dans les paquetages pour des chaînes de longueur variable soit bornée (`Bounded_String`), soit non bornée (`Unbounded_String`).

### *3.5 .4 Traitement numérique*

Le paquetage de fonctions élémentaires (`Sin`, `Sqrt`, etc.) qui était déjà une norme secondaire d'Ada 83, fait désormais partie de l'environnement standard. De plus, il existe des paquetages de nombres aléatoires[1].

### *3.5 .5 Entrées-sorties*

Le paquetage `Ada.Text_IO` s'est vu rajouter les nouveaux sous-paquetages génériques `Modular_IO` et `Decimal_IO` pour les types modulaires et décimaux, respectivement. `Modular_IO` est semblable à `Integer_IO`, et `Decimal_IO` à `Fixed_IO`. `Text_IO` s'est également enrichi d'un nouveau fichier prédéfini, `Standard_Error`, correspondant à la sortie d'erreur que l'on trouve sur de nombreux systèmes d'exploitation, d'une procédure `Get_Immediate` permettant de lire un caractère «au vol» sans attendre de retour chariot et d'une procédure `Look_Ahead` permettant d'anticiper sur le prochain caractère à lire.

Un nouveau paquetage `Ada.Wide_Text_IO` est semblable à `Text_IO`, mais travaille avec des `Wide_Character` et des `Wide_String`.

### *3.5 .6 Interfaces*

Le paquetage `Interfaces` regroupe des enfants pour les différents langages dont l'interfaçage est supporté. Les spécifications de `Interfaces.C`, `Interfaces.FORTRAN` et `Interfaces.COBOL` sont données dans la norme. On y trouve les déclarations Ada des types de ces langages, ainsi que des fonctionnalités nécessaires à une bonne liaison. C'est ainsi que, pour C, le paquetage enfant `Interfaces.C.Strings` gère des chaînes de caractères «à la C» (pointeur sur un flot d'octets terminé par un caractère NUL), et que `Interfaces.C.Pointers` gère une arithmétique sur pointeurs.

Nous ne pouvons qu'insister sur la facilité d'interfaçage apportée par ces paquetages ; dans le cadre du projet GNAT (cf. paragraphe 3.8), nous avons implémenté une interface complète avec le système d'exploitation OS/2 (en utilisant la bibliothèque C) ; grâce à ces paquetages, tout a fonctionné sans problème dès le premier essai !

## 3.6 Annexes

Ada est un langage à spectre très large ; ses plus beaux succès se trouvent aussi bien dans le contrôle de simulateurs ou l'aéronautique qu'en gestion financière ou en CAO. Or il est évident qu'un utilisateur «temps réel» n'aura pas les mêmes exigences qu'un utilisateur «gestion». C'est pourquoi la norme définit certains éléments comme «dépendants de l'implémentation», afin de ne pas privilégier un type d'application aux dépens d'un autre. D'autre part, Ada est un langage

[1] Notons pour la petite histoire que la formulation du générateur aléatoire a été l'un des points les plus débattus de l'histoire d'Ada 95, et pour lequel le consensus a été le plus difficile à obtenir...

extensible : son mécanisme de paquetages permet de rajouter toutes les fonctionnalités voulues sans avoir à toucher au langage lui-même.

Ada 95 est muni d'un certain nombre d'annexes correspondant à divers domaines d'utilisation. Elles précisent des éléments du langage autrement laissés libres, et définissent des paquetages spécifiques aux différents domaines. Les annexes ne contiennent ni nouvelle syntaxe, ni modification des règles du langage telles que définies dans la norme principale. Les fournisseurs de compilateurs seront libres de passer une, plusieurs, ou même aucune des suites de tests correspondant aux différentes annexes. Bien entendu, le certificat de validation mentionnera quelles annexes ont été passées.

Si la validation n'est accordée pour une annexe que si elle est entièrement implémentée, il n'est pas interdit de n'en fournir qu'une partie, *à condition que les fonctionnalités offertes soient conformes à la définition de la norme*. Il sera donc possible d'offrir, par exemple, le paquetage de gestion des nombres complexes même si des contraintes matérielles empêchent de valider totalement l'annexe numérique (précision de la machine insuffisante par exemple).

Il est important de comprendre que ce mécanisme n'est pas une rupture avec le dogme d'unicité du langage ; au contraire, il augmente la portabilité des applications en poursuivant la standardisation d'éléments spécifiques d'un domaine au-delà de ce qu'il était possible de faire dans le cadre d'un langage général. On trouve ainsi six annexes :

L'annexe sur la *programmation système* décrit des paquetages et des pragmas permettant une gestion plus directe des interruptions, la pré-élaboration de paquetages (mécanisme facilitant la mise en mémoire morte de parties de systèmes) et des manipulations supplémentaires de bas niveau sur les tâches. La documentation devra comporter obligatoirement des précisions sur l'efficacité en temps d'exécution de certains mécanismes.

- L'annexe sur les *systèmes temps réel* rajoute des fonctionnalités de gestion des priorités (priorités dynamiques notamment), d'ordonnancement des tâches et de gestion des files d'attente (files prioritaires). Elle comporte des exigences sur les délais avant avortement et la précision de l'instruction **`delay`**, un modèle de tâches simplifié permettant une implémentation extrêmement efficace, et de nouvelles fonctionnalités de gestion du temps. Noter qu'un compilateur enregistré comme supportant cette annexe devra obligatoirement supporter également l'annexe sur la programmation système.

  L'annexe sur les *systèmes distribués* permet de définir un programme comme un ensemble de partitions s'exécutant sur des processeurs différents. Le programmeur peut contrôler le format des messages échangés entre les différentes partitions, et il est possible de remplacer dynamiquement une partition sans relancer tout le système. Cette annexe définit également un modèle d'appel de sous-programme distant (*remote procedure call*). Vu son importance, nous la détaillerons quelque peu ci-dessous.

  L'annexe sur les *systèmes d'information* fournit de nouvelles fonctionnalités de présentation des données (utilisant des clauses *picture* analogues à celles de COBOL) et des paquetages d'arithmétique complémentaire sur les nombres décimaux. Cette annexe s'est réduite au fil du temps, car une grande partie de ce qu'elle comprenait a été jugée tellement importante qu'elle a été remise dans le corps du langage !

  L'annexe *sûreté et sécurité* impose des contraintes supplémentaires au compilateur sur le traitement des cas appelés «erronés» dans le langage et sur un certain nombre de points laissés à la discrétion de l'implémentation. De plus, le compilateur doit fournir tous les éléments (*listings*) permettant de contrôler manuellement le code généré et offrir des pragmas permettant la vérification par des outils extérieurs.

  L'annexe sur les *numériques* fournit un modèle encore plus précis des calculs réels, notamment pour les machines utilisant le modèle IEEE des nombres flottants, et inclut la définition des paquetages de nombres complexes, ainsi que les fonctions élémentaires et les entrées-sorties sur ces nombres complexes.

## 3.7 Systèmes répartis

L'annexe *systèmes distribués* définit des éléments supplémentaires permettant d'écrire des systèmes répartis. Ada 95 est le premier langage à inclure un tel modèle *au niveau de la définition du langage*, donc de façon portable et indépendante des systèmes d'exploitation ou des exécutifs sous-jacents.

### *3.7 .1 Partitions*

Un programme Ada est constitué d'un ensemble de «partitions» faiblement couplées, susceptibles de se trouver physiquement sur des machines différentes. Les partitions peuvent être actives ou passives ; dans ce dernier cas, elles ne peuvent posséder de tâche en propre ni de programme principal. La façon de construire des partitions à partir du programme est laissée à la discrétion de l'implémentation, mais les contrôles à la compilation sont faits au niveau «programme» : on garantit donc la cohérence de toutes les partitions figurant dans une même bibliothèque.

La définition de la notion de partition est volontairement vague, afin de lui permettre de correspondre à différentes utilisations, que nous avons résumées dans le tableau de la figure 4.

| | **Partition active** | **Partition passive** |
|---|---|---|
| **Cartes autonomes** | Carte processeur | Carte mémoire |
| **Réseau local** | Ordinateur | - |
| **Système d'exploitation** | Processus | DLL,<br>mémoire partagée |

**Figure 4 :** Utilisations des partitions

### *3.7 .2 Classification des paquetages*

L'écriture d'un système distribué nécessite certaines précautions. Par exemple, si plusieurs partitions utilisent un paquetage «normal», celui-ci sera dupliqué dans chacune d'elles : s'il possède des états cachés, ils évolueront indépendamment dans chaque partition. Si l'on souhaite un autre comportement, on peut fournir des pragmas de catégorisation pour définir des sémantiques différentes qui apportent les restrictions (vérifiées par le compilateur) nécessaires au bon fonctionnement du système. On trouve ainsi :

Les paquetages «purs» (*pure*). Ils sont dupliqués dans chaque partition, mais les restrictions imposées garantissent que leur comportement est indépendant de leur duplication (pas d'état local).

Les paquetages «partagés passifs» (*shared_passive*). Ils ne peuvent contenir aucune entité active (ni servir à accéder à une entité active de façon indirecte) et sont présents dans une seule partition passive du système. Ils peuvent contenir des sous-programmes et des états rémanents, qui sont partagés par tous les utilisateurs du système.

Les paquetages «types distants» (*remote_types*). Ils servent à définir les types de données susceptibles d'être échangés entre partitions. Les restrictions associées garantissent que ces types ont les propriétés nécessaires pour permettre leur transmission sur un réseau (pas de pointeurs locaux par exemple, sauf si l'utilisateur a défini une procédure permettant la transmission).

- Les paquetages «interface d'appel distant» (*remote_call_interface*, en abrégé RCI). La spécification de ces paquetages est dupliquée dans chaque partition, mais le corps n'est physiquement présent que dans une seule ; dans les autres, un corps de remplacement route les appels vers la partition qui possède le corps vrai.

### *3.7 .3 Appels à distance*

Lorsqu'un sous-programme est défini dans un paquetage «interface d'appel distant», tous les appels sont routés vers la partition sur laquelle il se trouve physiquement. De même, un pointeur sur sous-programme défini dans un paquetage «interface d'appel distant» ou «types distants» est un type accès à distance (*remote access type*). Ceci signifie que le pointeur contient l'information nécessaire pour localiser la partition où se trouve physiquement le sous-programme. Tout appel à travers un tel pointeur sera également un appel distant. Les restrictions sur ces paquetages garantissent que les paramètres de tels sous-programmes sont toujours transmissibles sur le réseau.

De plus, lorsqu'un appel est transmis à travers le réseau, le compilateur doit utiliser pour le routage les services d'un paquetage prédéfini (`System.RPC`), dont le corps peut être fourni par l'utilisateur, le fournisseur du réseau, etc. Le compilateur est ainsi indépendant de la couche «transport», et il est possible d'implémenter la distribution par-dessus n'importe quelle couche réseau sans avoir à toucher au compilateur.

### *3.7 .4 Cohérence d'un système réparti*

Le problème de la cohérence d'un système réparti est délicat, car il doit répondre à deux exigences contradictoires :

- d'une part, il faut garantir que toutes les partitions du système «voient» les données communes de la même façon ;
- d'autre part, il faut pouvoir arrêter et relancer une partition, éventuellement pour corriger une erreur, sans être obligé d'arrêter tout le système.

Un programme Ada est dit «cohérent» si toutes les partitions qui le constituent utilisent la même version de toutes les unités de compilation. On autorise un programme Ada à être incohérent, c'est-à-dire qu'il est possible de corriger une erreur qui n'affecte qu'une seule partition, d'arrêter la partition et de la relancer avec la nouvelle version. Toutefois, si la correction implique un paquetage «partagé passif» ou «interface d'appel distant» (les seuls qui ne soient pas dupliqués), alors toute communication est coupée entre la nouvelle partition et le reste du système. On autorise donc un système *localement incohérent*, tant que cela ne sort pas de la partition en cause ; la cohérence est exigée pour les seuls éléments qui sont physiquement répartis sur le réseau.

Enfin, chaque unité de compilation possède un attribut `'Version` (pour la spécification) et `'Body_Version` (pour le corps), dont la valeur change à chaque modification. Il est donc possible de programmer des contrôles de cohérence plus fins si cela est nécessaire.

## 3.8 Le compilateur GNAT

Un des obstacles importants à la diffusion d'Ada a longtemps été le prix des compilateurs, notamment pour les institutions universitaires. Il est vrai que les fournisseurs ont consenti un important effort financier pour les organismes d'enseignement, mais le meilleur moyen de convaincre les gens d'utiliser Ada est de leur faire essayer le langage[1]. Or peu d'enseignants (aux crédits limités !) sont prêts à acheter un compilateur, même à prix réduit, juste pour voir...

Conscient de cet état de fait, le DoD a financé le développement d'un compilateur Ada 95 dont la diffusion est entièrement libre et gratuite. Il s'agit en fait d'un frontal du célèbre compilateur multilangages GCC[2], faisant partie de l'environnement GNU de la Free Software Foundation, connu sous le nom de GNAT (GNU Ada Translator). La réalisation en a été confiée à la fameuse équipe de New York University, qui s'était déjà illustrée en réalisant le premier compilateur Ada (83) validé.

---

1 Nous avons souvent constaté que les plus farouches opposants à Ada ne l'avaient jamais essayé...

2 Pour Gnu Compiler Collection; le langage C n'est qu'un parmi les nombreux autres langages acceptés.

Comme tous les logiciels diffusés par la Free Software Foundation, GNAT est un logiciel libre, disponible non seulement sous forme exécutable, mais aussi sous forme de sources. Tous les enseignants des cours de compilation peuvent ainsi offrir à leurs élèves d'intervenir dans un compilateur Ada, c'est-à-dire dans ce qui se fait de mieux en matière de compilation.

GNAT est disponible par les canaux de diffusion habituels du GNU, en particulier on peut le charger par FTP anonyme depuis tous les bons serveurs de logiciels libres.

## 3.9 Exercices

Ces exercices s'adressent plus particulièrement aux anciens utilisateurs d'Ada 83.

1. Etudier comment la notion de bibliothèque hiérarchique peut être utilisée pour réaliser la hiérarchisation des unités de la méthode HOOD (cf. chapitre 10, paragraphe 10.3.2)
2. Reprendre les exemples classiques de boîte aux lettres ou de variable protégée qui utilisaient des tâches en Ada 83, et les récrire avec des types protégés.
3. Le manuel de référence Ada 83 donne un exemple d'utilisation de l'instruction **`delay`** pour attendre jusqu'à une heure absolue. Expliquer pourquoi cet exemple est faux, ce qui a rendu nécessaire l'introduction de **`delay until`**.
4. Comparer les fonctionnalités offertes par les nouveaux paquetages prédéfinis avec ceux de la bibliothèque standard d'autres langages, C notamment.

# Première partie

# Langages et méthodes

Méthodes et langages ont toujours eu des rapports conflictuels. Voici un rapprochement de quelques citations et/ou lieux communs pour comprendre l'étendue du problème.

> *Le langage n'a aucune importance ; seule l'application de bonnes méthodes peut résoudre la crise du logiciel.*
>
> *Le langage Ada a été spécifiquement conçu pour soutenir les méthodes de conception.*
>
> *La méthode doit être indépendante du langage d'implémentation utilisé.*
>
> *La méthode HOOD a été conçue pour répondre à un appel d'offres de l'Agence spatiale européenne qui voulait une méthode spécifique pour Ada.*
>
> *Langage sans méthode n'est que ruine... de la société de service.*

Nous voyons à travers ces quelques exemples une contradiction dans la perception du rôle du langage par rapport à celui de la méthode : simple moyen de traduire une conception en une forme compréhensible par l'ordinateur pour les uns, expression directe *et vérifiée* de la conception pour les autres. En fait, les deux ont raison... selon le langage considéré. Dans cette partie, nous allons étudier les rapports des méthodes aux langages, et nous verrons que si les autres langages ne peuvent que partiellement apporter leur soutien aux méthodes, Ada a été conçu précisément pour leur servir de prolongement naturel.

# 4

# Rôle et principes des méthodes de conception

Si la nécessité d'une méthode est bien établie dans le développement de logiciels industriels, elle est trop souvent ressentie comme une gêne par le programmeur, qui souhaiterait toujours coder immédiatement. D'ailleurs, celui-ci a souvent écrit des programmes sans utiliser de méthode, et ils ont très bien marché. L'intérêt d'une méthode ne se fait sentir qu'à long terme, souvent au niveau des phases d'intégration. Ceux qui pensent pouvoir s'en dispenser nous rappellent cette citation d'un industriel américain du début du siècle :

> *Nous n'avons que faire du téléphone ; nous possédons un système de coursiers qui fonctionne très bien.*

Aussi est-il important de bien comprendre le rôle joué par les méthodes pour la conception et leur impact sur le cycle de vie. La décision d'écrire un programme d'informatique est toujours prise dans le but de résoudre un problème du monde réel[1]. L'expression de la solution informatique souhaitée est le cahier des charges, qui définit ce que doit faire l'application informatique. Nous allons parler ici des méthodes permettant, à partir de là, d'obtenir un programme répondant à ses exigences ; il s'agit donc de ce que l'on appelle les méthodes de conception détaillée, par opposition aux méthodes de spécification qui interviennent dans l'établissement du cahier des charges.

## 4.1 Complexité et limitations de l'esprit humain

Lorsque le problème à résoudre est suffisamment simple, il est relativement facile de concevoir directement le programme à partir du cahier des charges. Dans l'enseignement traditionnel de l'informatique, qui se réduit trop souvent à l'enseignement de la programmation, on fait exprès de donner des problèmes dont la solution se déduit relativement facilement de l'énoncé. Même si l'on impose une méthode à l'étudiant, celui-ci a généralement l'impression qu'elle est inutile, car il se sent capable de produire directement la solution. Bien souvent, il écrira tout de suite le programme et ne produira les documents de conception que par la suite, pour faire plaisir au professeur.

Ne jetons pas la pierre aux seuls étudiants : deux ans plus tard, ceux-ci sont ingénieurs et continuent à appliquer les mêmes méthodes de travail. *Le problème est que cela fonctionne parfaitement !* Bien sûr on connaît parfois quelques difficultés de maintenance, la survie du programme est mise en péril lors du départ du concepteur initial, mais *grosso modo* on y arrive, d'autant mieux que l'entreprise dispose de personnels «doués». Mais insensiblement, la taille des programmes tend à augmenter. On continue à développer de la même façon, avec des difficultés croissantes, mais puisqu'on y arrivait jusqu'à présent, il n'y a pas de raison que cela ne continue pas à fonctionner... Le problème est qu'il existe une taille critique, que l'on peut évaluer aux alentours de

[1] Sauf pour ceux qui écrivent des virus, qui cherchent alors plutôt à *créer* des problèmes...

10 000 lignes de code par programmeur[1], pour laquelle les problèmes de gestion vont soudain déborder les capacités des programmeurs. Un proverbe résume bien ce phénomène :

*On ne construit pas un pont sur un estuaire en extrapolant une passerelle sur une rivière.*

On ne peut définir précisément à partir de quelle largeur de rivière il faudra changer de technologie ; mais il est certain que personne ne songerait à faire le pont de Normandie au moyen d'un tronc jeté en travers ! En logiciel, le problème est qu'en quelque sorte, la rivière s'élargit progressivement jusqu'au point de rupture : que survienne une demande nouvelle, le besoin d'intégrer des interfaces graphiques, le portage sur une nouvelle machine, et la taille critique sera dépassée. Tout à coup, le programmeur sera incapable de répondre à la demande, il sera dépassé par son programme. Que s'est-il passé ?

On a découvert [Mil56] que le cerveau humain ne peut s'occuper en moyenne que de 7 ± 2 éléments en même temps ; tant que le programmeur reste au dessous de cette limite, tout va bien. La complexité du logiciel augmentant, elle va atteindre *insensiblement* la limite des capacités du concepteur. Celui-ci ne parviendra plus à maîtriser simultanément tous les éléments nécessaires. L'effet de ce phénomène est particulièrement frustrant, car le programmeur continue à maîtriser les vues partielles de son problème ; simplement, lorsqu'il commence à saisir une partie des éléments nécessaires à la compréhension, cela se fait au détriment d'une autre partie qu'il avait comprise précédemment. Face à cette situation, il va chercher désespérément à continuer à percevoir tous les éléments. S'il est très doué, il y parviendra temporairement. Mais une fois le problème résolu et l'échauffement intellectuel passé, ni lui, ni aucun autre programmeur chargé d'effectuer la maintenance ne retrouvera cet «état de grâce» temporaire : le programme sera devenu totalement incompréhensible.

Il faut se faire une raison : la seule chose que la technologie ne peut améliorer, ce sont nos facultés mentales. Comme le disait Dijkstra : «*I only have a very small head and I must live with it*[2]». Méthodes et langages sont les outils qui nous permettent de réaliser des logiciels toujours plus complexes, en tenant compte de nos limitations intellectuelles.

## 4.2 Notion de saut sémantique

Le cahier des charges d'un problème informatique représente une description de haut niveau de la solution souhaitée. Un programme est une réalisation particulière du cahier des charges, c'est-à-dire une description de bas niveau d'une solution. La conception, c'est-à-dire le processus permettant de passer d'une description à l'autre, est donc fondamentalement un *saut sémantique*.

Dans un problème simple, un tel saut peut s'accomplir directement : des méthodes intermédiaires sont donc inutiles. Au fur et à mesure que le problème se complique, la taille du saut sémantique augmente. Rapidement, les limitations intrinsèques de nos capacités intellectuelles ne nous permettent plus de maîtriser ce saut, et il ne devient plus possible de passer directement de l'énoncé du problème à sa solution. Or que fait-on dans la vie courante lorsque l'on est en présence d'un saut trop important pour pouvoir être effectué directement ? On le remplace par une succession de petits sauts (communément appelée *escalier*), individuellement faisables, et dont la somme permet d'accomplir la descente souhaitée. De même en génie logiciel, il faudra définir des étapes de conception intermédiaires destinées à abaisser graduellement le niveau sémantique depuis le niveau du cahier des charges jusqu'à celui du langage de programmation.

---

[1] Cette limite ne provient pas de mesures précises, mais est plutôt déterminée intuitivement par nos expériences personnelles. Nous pensons cependant que la plupart des chefs de projet seront d'accord avec ce chiffre.

[2] Je n'ai qu'une toute petite tête, et il faut bien que je vive avec.

## 4.3 Surmonter la complexité

Comment faire pour réaliser ces différents «sauts» qui constituent l'essentiel d'une méthode de conception ? Le problème, qui n'est pas propre à l'informatique, a été résolu par des moyens connus depuis Descartes :

> *... diviser chacune des difficultés que j'examinerais en autant de parcelles qu'il se pourrait et qu'il serait requis pour les mieux résoudre. [Des37]*

Informatiquement parlant, ceci signifie que l'on divisera un programme en *modules*, qui seront des unités de programme pouvant être considérées individuellement par le programmeur. *Dans le reste de ce chapitre, nous utiliserons ce terme de «module» pour désigner l'unité de structuration conceptuelle*, même si cela ne correspond pas forcément à la notion informatique de «quantité compilable individuellement».

Ce que Descartes ne dit pas, c'est le nombre requis de telles divisions (*autant qu'il serait requis...* voilà qui ne nous avance guère !). On pourrait croire qu'il suffit de diviser le problème en un grand nombre de modules pour surmonter la complexité. Or il n'en est rien ; le fait de diviser un programme introduit un niveau de complexité supplémentaire : celui dû aux relations nécessaires entre les parties. Supposons une échelle arbitraire, où 7 représenterait la limite de ce qui peut être raisonnablement fait par une seule personne ; si un programme est de complexité 14, il faudra non pas deux, mais trois personnes pour le résoudre, la troisième étant chargée de gérer la communication entre les deux autres. Ceci va en fait limiter le gain pouvant être obtenu de la décomposition. Considérons en effet deux cas extrêmes :

Nombre de modules = 1. Aucune complexité de relation, mais complexité interne au module maximale.

- Nombre de modules = autant que d'instructions (chaque module ne contient qu'une seule instruction). Aucune complexité interne, mais énorme complexité de relation. En fait, la complexité globale dans ce cas est équivalente au cas précédent ; on n'a fait que *transformer* une complexité interne en complexité de relation.

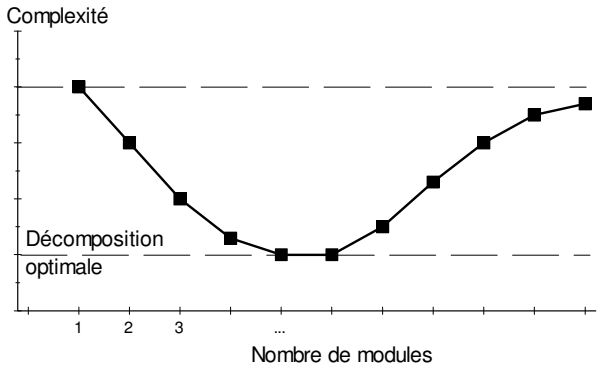


**Figure 5 :** Variation de la complexité avec le nombre de modules

Les deux cas extrêmes étant équivalents, cela signifie qu'il existe une taille optimale de décomposition, pour laquelle la complexité globale (interne + relation) est minimale[1], comme illustré sur la figure 5. En revanche, rien n'indique que cette complexité minimale soit inférieure à 7 ; autrement dit, dans un projet d'une taille suffisante, si l'on se contente de le diviser simplement en modules, on atteindra une complexité de relation trop importante *avant* d'avoir obtenu une complexité interne acceptable. En quelque sorte, décomposer un problème en sous-problèmes ne nous fournit qu'une seule «marche» dans notre descente des niveaux sémantiques. Il faut raffiner le processus afin d'obtenir autant d'étapes que nécessaire.

Pour cela, on décomposera le problème en un nombre de modules restreint ; la complexité de relation sera acceptable, mais chaque module possédera une complexité interne trop importante. On

[1] Mathématiquement parlant, cela pourrait aussi bien être un maximum... fort heureusement, il n'en est rien.

considérera alors chaque module comme un problème *autonome*, et on lui appliquera de nouveau la méthode ci-dessus, définissant ainsi des sous-modules, puis des sous-sous-modules jusqu'à obtenir des unités de taille acceptable. Ceci implique, pour garder une complexité de relation gérable, qu'un module d'un certain niveau ne doit avoir de relation conceptuelle qu'avec les modules de *même* niveau. Le programme sera donc organisé en *couches logicielles hiérarchisées*, dont l'organisation générale correspondra à la figure 6. Chaque modules dépendra logiquement d'un nombre restreint d'autres modules : le graphe de ces dépendances s'appelle la *topologie de programme*.

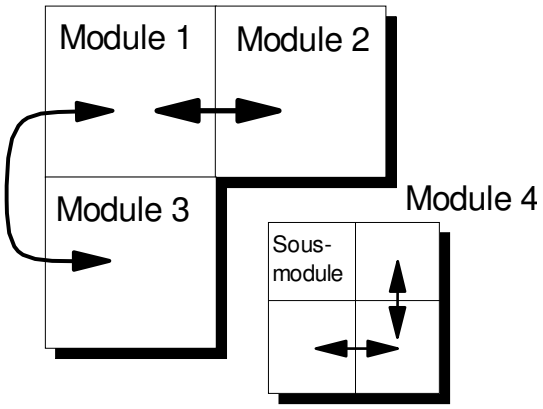


**Figure 6 :** Décomposition hiérarchique

Ces considérations ne sont que la justification en termes de maîtrise de la complexité d'une technique connue depuis longtemps : la méthode d'analyse descendante (ou de décomposition hiérarchique). Il reste pourtant une question en suspens dans ce mode de décomposition : en fonction de quels critères va-t-on décomposer les *difficultés* en *parcelles* (ou, disons, les programmes en modules) ? Un problème, quel qu'il soit, mais en particulier en informatique, ne se découpe pas en tranches comme un vulgaire saucisson : il faut établir des *critères logiques* pour répartir les différentes parties du logiciel entre les modules. L'analyse précédente montre qu'il existe *deux dimensions* dans l'analyse : la décomposition *horizontale* qui détermine les sous-modules d'un module donné, et la décomposition *verticale* qui détermine à quel sous-niveau un module doit appartenir. Il sera donc nécessaire de posséder *deux* ensembles de critères : l'un, pour la décomposition horizontale, qui détermine la répartition des éléments entre modules d'une même étape de décomposition, et l'autre, pour la décomposition verticale, qui détermine à quel niveau de la décomposition un élément donné doit apparaître.

## 4.4 Caractérisation des méthodes

Il existe de nombreuses méthodes de conception, regroupées dans un certain nombre de *familles* (méthodes structurées, en flots de données, entités-relations, orientées objet...). Chaque méthode a ses partisans, et selon le domaine du problème à résoudre, l'une ou l'autre peut s'avérer plus performante. Comment donc définir ce que contient une méthode ?

L'élément essentiel qui caractérise une méthode est son principe de base, c'est-à-dire les *critères* utilisés pour les décompositions horizontale et verticale. On dit que des méthodes appartiennent à la même famille lorsque leurs critères de décomposition sont identiques, ou au moins très voisins. Noter que comme la décomposition résulte de l'application de ces critères, la topologie de programme obtenue sera, en général, caractéristique de la méthode.

Ensuite, il convient de mettre en œuvre ces principes : les méthodes définissent donc une démarche précise à suivre pour guider pas à pas le processus de conception. En particulier, elles définissent des étapes de conception et des critères d'acceptabilité permettant de passer d'une étape à la suivante.

Enfin, une méthode définira les représentations (textuelle, graphique) et les documents qui doivent accompagner la conception, dans un but de suivi du développement aussi bien que d'aide à la maintenance.

En général, des outils informatiques ont été développés pour soutenir la bonne application des méthodes, en particulier en automatisant le suivi de la conception et la production des documents associés.

## 4.5 Méthodes de conception et langages

Comme nous l'avons vu précédemment, il est nécessaire d'abaisser graduellement le niveau sémantique de la description de la solution jusqu'au point où l'on peut la décrire au moyen du langage de programmation. La comparaison avec l'escalier montre bien comment définir de façon optimale le nombre et le niveau de ces étapes :

Il y aura d'autant plus d'étapes intermédiaires que la distance entre le niveau du problème et celui du langage sera plus grande.

- Les étapes seront optimales si elles correspondent à des sauts sémantiques d'égale hauteur.

Ces constatations, qui semblent évidentes, ont pourtant des conséquences qui ne sont pas toujours comprises. En particulier, on a longtemps dit que les méthodes de conception devaient être indépendantes du langage de codage utilisé. Si l'on utilise un langage de plus haut niveau, la répartition optimale des «marches» sera différente, certaines pouvant même être appelées à disparaître (Fig. 7). C'est ainsi que s'il est fondamental de décrire des algorithmes au moyen de pseudo-code si l'on programme en FORTRAN, c'est totalement inutile lorsque l'on utilise Ada, car le langage de codage est alors de même niveau sémantique (si ce n'est supérieur !) que le pseudo-code.

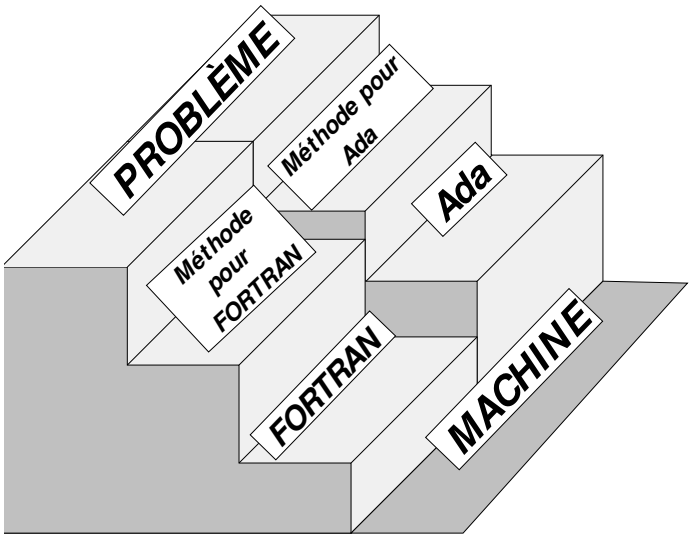


**Figure 7 :** Niveau relatif des méthodes et des langages.

Retenons donc que l'utilisation d'un langage de haut niveau comme Ada va permettre une *remontée du niveau sémantique* de toutes les étapes intermédiaires. C'est donc tout le processus de développement qui est remis en cause.

Ne peut-on aller plus loin, inventer un langage de niveau si haut qu'il nous dispenserait totalement de l'utilisation d'une méthode ? En un sens oui : à partir d'une méthode comme HOOD, il existe des outils qui produisent (presque) automatiquement le code Ada correspondant. Dans l'idéal, il n'existerait plus d'intervention manuelle entre la méthode et l'exécution par l'ordinateur ; c'est la méthode qui serait devenue langage de très haut niveau.

En pratique, il existe une limite à la remontée de niveau sémantique des langages de programmation conventionnels. Ceux-ci se veulent en effet «universels», donc permettant de

réaliser des applications de domaines, et de contraintes, très variés. Or contrôler, c'est nécessairement restreindre. Certaines possibilités offertes par un langage sont impératives pour un domaine d'application, et dangereuses pour d'autres : on n'utilisera pas l'héritage dans des applications temps réel, ni la manipulation directe d'interruptions dans une application de gestion. Notons au passage qu'il est par conséquent *normal* qu'aucune application n'utilise l'intégralité des possibilités d'un langage.

Tout langage est donc un compromis entre puissance d'expression (pouvoir tout faire) et sécurité (tout contrôler). Un langage à vocation universelle sera limité dans ses possibilités de contrôles par la nécessité d'offrir les services requis par diverses classes d'applications. La méthode en revanche est généralement spécifique d'un domaine particulier : personne ne songerait à utiliser Merise en temps réel, ni Buhr en gestion ! Etant plus spécifique, elle peut donc exprimer des contraintes supplémentaires.

> *En plus de sa fonction d'orientation et de guidage du processus de développement, le rôle d'une méthode est d'imposer des restrictions supplémentaires, au-delà de ce qui peut être vérifié au niveau du langage de programmation.*

# 5

# Rôle et principes d'un langage de programmation

Après avoir étudié le rôle des méthodes, nous allons nous intéresser dans ce chapitre à leur prolongement, le langage, et à son influence sur la conception et sur la maintenabilité des applications informatiques. Dans cette réflexion sur le rôle des langages de programmation, nous utiliserons et nous contrasterons principalement Ada d'une part, et le couple C/C++ d'autre part. Ce choix n'est pas seulement dû au fait que ces langages sont les plus utilisés actuellement : nous verrons qu'ils correspondent à deux approches, deux philosophies pourrait-on dire, de la programmation fondamentalement différentes, qui les ont conduit à faire des choix radicalement opposés. Pourquoi ne pas faire la différence entre C et C++ ? C++ est un perfectionnement technologique du langage C ; on l'appelle souvent *the C after*[1]. Mais malgré l'introduction d'outils de plus haut niveau (notamment l'héritage), les principes, les rapports du programmeur avec son langage, sont restés les mêmes.

## 5.1 Le double niveau de lecture

On considère généralement que le rôle d'un langage est de porter une information, un message, d'une personne qui parle à une personne qui écoute. Dans le cas des langages de programmation, celui qui «parle» est bien entendu le programmeur. Mais qui est celui qui «écoute» ? L'ordinateur, pense-t-on généralement. Certes, mais aussi celui qui relira le logiciel plus tard, c'est-à-dire le programmeur de maintenance[2].

Hofstadter [Hof85] soutient que le message ne *porte* pas l'information, mais ne fait que *déclencher* l'information contenue dans l'interlocuteur en «appuyant» sur des «boutons» préexistants. Considérons par exemple la phrase suivante (tirée d'un livre de lecture, classe de CM1) :

*Dans la boulangerie, Madame Durand dit à Florence : «Tu n'as pas vu Minet ?»*

En lisant cette phrase, vous avez parfaitement compris que

*La boulangère s'inquiète auprès de la petite fille d'avoir perdu son chat.*

Or *aucun* des éléments de la seconde formulation ne figure dans la première ! Comment savez-vous que Madame Durand est la boulangère ? Parce que *vous savez* que si ce n'était pas la boulangère, la formulation «dans la boulangerie» n'aurait aucune utilité. Qui vous a dit que Florence était une petite fille ? Parce que *vous savez* que la formulation «Madame X» d'un côté, un prénom de l'autre traduit une relation d'adulte à enfant. Qui vous a dit que le chat était perdu ? Parce que

[1] Le C d'après **.**

[2] Dans le cadre du génie logiciel, on doit toujours considérer que celui qui relit un logiciel n'est pas celui qui l'a écrit.

*vous savez* que la construction «Tu n'as pas vu...» traduit une inquiétude. Et bien entendu, *vous savez* que «Minet» est un nom de chat...

Cet exemple montre bien que la compréhension *par l'homme* d'un texte fait usage de toute une culture extérieure au seul message. Or la lecture d'un texte de programme par un ordinateur ne mettra en jeu que le texte lui-même et les règles du langage. C'est de cette différence de lecture que provient un grand nombre d'erreurs, la plus célèbre étant celle survenue à la NASA dans les années 60. Une sonde destinée à observer Vénus est passée à dix fois la distance prévue, suite à une erreur de calcul dans la trajectoire. Une commission d'enquête a révélé qu'un programmeur avait écrit (en FORTRAN) :

```
DO 10 I = 1.5
```

Dans l'esprit de celui qui avait écrit cette instruction, *ainsi que dans l'esprit de tous ceux qui l'avaient relue par la suite*, il s'agissait d'une boucle `DO`. Cette conviction était suffisamment forte pour que l'on ne remarque pas la présence d'un point à la place de la virgule normalement requise dans une boucle `DO`... Le compilateur qui ne comprend en revanche que ce qui est écrit a relu cette instruction sous la forme (sachant que les espaces ne sont pas significatifs en FORTRAN) :

```
DO10I = 1.5
```

c'est-à-dire comme l'affectation de la valeur réelle 1.5 à la variable de nom `DO10I`, non déclarée (mais ce n'est pas un problème en FORTRAN où la déclaration de variable n'est pas obligatoire[1])...

Plus prosaïquement, si vous lisez dans un programme :

```
for I in 1..MILLE loop
   ...
end loop;
```

vous serez persuadé que la boucle sera exécutée 1000 fois, car vous *savez* que la chaîne de caractères 'M', 'I', 'L', 'L', 'E' a la même signification que le nombre 1000. Le compilateur, lui, ira regarder dans sa table des symboles, où il trouvera que l'identificateur `MILLE` résulte de la déclaration :

```
MILLE : constant := 10_000;
```

il exécutera donc la boucle dix mille fois...[2]

La compréhension *par le programmeur* de son programme s'appuie donc sur des éléments non écrits faisant partie de sa culture propre. Pourquoi est-ce nécessaire ? Après tout, le compilateur est bien capable de s'y retrouver sans ces éléments «culturels». Mais l'ordinateur dispose d'un atout complémentaire : une mémoire illimitée[3] ; toute information qu'on lui a communiquée une fois reste disponible. Au contraire, le programmeur ne dispose que d'une mémoire à court terme limitée (dont, comme nous l'avons vu, on évalue la capacité à sept «cases»). L'homme va compenser ce manque de mémoire en déduisant les éléments dont il a besoin, recréant ainsi l'information pour éviter de la stocker. Plus prosaïquement, ceci justifie la nécessité (toujours répétée, jamais justifiée, et pas toujours observée) d'avoir des identificateurs *parlants* : si l'on veut stocker le nombre de lignes imprimées, cela ne fait aucune différence pour l'ordinateur d'appeler la variable `XYZ123`, `NBLN` ou `NOMBRE_DE_LIGNES`. En tant qu'être humain, je dois entièrement mémoriser le rôle de la variable `XYZ123`. Dans le deuxième cas, je dois mémoriser que `NB` est une abréviation pour «nombre» et `LN` pour «lignes». Ce stockage est cependant plus économique, car de nombreuses variables peuvent être construites en combinant ainsi un petit nombre d'abréviations. Avec la troisième forme, je n'ai rien à mémoriser : je peux déduire entièrement la signification de la variable de son nom.

Cependant, cette information portée par le nom de la variable échappe totalement au compilateur : rien n'empêche la variable `NOMBRE_DE_LIGNES` de désigner la distance de la Terre à la Lune ! Tout l'art du programmeur va donc être d'écrire son programme de façon qu'il porte deux

---

1 C souffre du même problème : il existe de nombreux cas où une faute de frappe d'un seul caractère conduit à un programme *légal*, mais faisant quelque chose de totalement différent de ce qui était prévu.

2 Cet exemple n'est *pas* irréaliste ; nous l'avons effectivement trouvé dans un projet réel...

3 Tout au moins comparée à la mémoire (à court terme) humaine.

messages à la fois : un pour le compilateur, gouverné par les seules règles du langage, et un pour le lecteur, gouverné par son fonds culturel, *et de faire en sorte que les deux correspondent* ! La difficulté réside bien entendu en ce que le «fonds culturel» varie d'un programmeur à l'autre. Un exemple typique se trouve dans l'utilisation d'abréviations : par exemple, `LN` est une abréviation «évidente» de «ligne» pour certains, alors que pour d'autres elle signifiera «longueur» ou «logarithme népérien». Il convient donc d'éviter de façon générale l'usage de tout «fonds culturel» non explicite, et plus particulièrement des abréviations[1].

Un exemple caractéristique de la non-compréhension de ce double rôle du langage de programmation peut être trouvé dans les langages C et C++ : dans ceux-ci, deux identificateurs qui ne diffèrent que par l'usage des minuscules ou des majuscules sont *différents*. Ainsi, `Nombre_de_Lignes` ne désigne pas la même variable que `NOMBRE_DE_LIGNES`. Du point de vue du compilateur, il s'agit de deux chaînes de caractères différentes, donc de deux identificateurs différents. En revanche, la signification «culturelle» pour l'être humain ne dépend pas de la façon d'écrire les mots, donc la compréhension de la signification est la même. On oblige le programmeur C à raisonner avec la vue de l'ordinateur, donc à mémoriser un nombre d'informations plus grand pour comprendre la signification d'un programme.

## 5.2 Niveau sémantique des langages

On entend souvent l'expression «langage de haut niveau». Nous avons dit précédemment que les méthodes de conception s'arrêtaient lorsque l'on avait atteint le «niveau» du langage de programmation. Mais comment définit-on le niveau d'un langage ? La meilleure définition se rapporte à la position du langage dans le processus de développement : plus la formulation sera proche de la définition du problème, plus nous dirons que le langage est de haut niveau. Inversement, si le langage exprime les contraintes de la réalisation sur machine, nous parlerons d'un langage de bas niveau.

Nous allons illustrer cette notion par un exemple simple, et voir comment différents langages permettent de prendre le relais des méthodes plus ou moins tôt (cet exemple est critiquable du point de vue des algorithmes utilisés ; son but n'est que de montrer la démarche d'abaissement du niveau sémantique dans un cas simple).

Supposons que nous voulions trouver tous les diviseurs d'un nombre donné `N`. La spécification du problème (point de départ) peut s'exprimer comme :

*Calculer la liste de tous les nombres P tels que P divise N*

La première étape consiste à *exprimer la description de la solution* sans faire référence à un quelconque *algorithme informatique*. Un langage comme SETL nous permet de nous situer directement à ce niveau. On écrirait :

```
LISTE := [P in [1..N] | N mod P = 0];
```

qui se lit «Mettre dans la variable `LISTE` la suite des nombres `P` de l'intervalle de 1 à `N` tels que `N` modulo `P` soit égal à 0». Noter qu'à ce niveau, on ne spécifie pas d'algorithme : on exprime le résultat souhaité, et c'est le travail du compilateur de déterminer *comment* obtenir ce résultat[2]. Nous ne nous préoccupons pas non plus de problèmes annexes, comme de savoir quelle quantité d'espace mémoire réserver : là encore, le système n'a qu'à se débrouiller.

La deuxième étape va consister à trouver un algorithme décrivant *comment* obtenir la solution désirée, ce qui nous obligera également à introduire les *structures de données* nécessaires à sa réalisation. L'algorithme s'exprimera au moyen de constructions de haut niveau (boucles, tests,

[1] Ceci n'est pas limité au logiciel... Le français qui arrive aux Etats-Unis reste en général perplexe devant les panneaux annonçant «PED. XING». Il ignore que la lettre X, qui représente une croix, est souvent utilisée comme abréviation pour le mot «cross» (croix). La signification du panneau est donc «Pedestrian Crossing» (passage piéton).

[2] Ce principe de description du résultat à obtenir et non du moyen d'y parvenir est également à la base du langage SQL dans le domaine des bases de données.

aiguillages...) et les structures de données pourront être relativement abstraites (piles, listes...). Si nous ne disposons pas d'un langage comme SETL[1], nous devrons exprimer notre conception à ce niveau. En Ada, ceci s'exprimerait comme :

```
declare
   type Index is new Positive range 1..N;
   Prochain : Index := 1;

   subtype Diviseurs is Natural range 0..N;
   Non_Alloué : constant Diviseurs := 0;

   Liste : array (Index) of Diviseurs
         := (others => Non_Alloué);
begin
   for P in 1..N loop
      if N mod P = 0 then
         Liste (Prochain) := P;
         Prochain := Prochain+1;
      end if;
   end loop;
end;
```

Nous avons choisi de représenter notre liste au moyen d'un tableau, ce qui nous pose des *contraintes d'implémentation*. Pour pouvoir ajuster nos dimensionnements, nous supposerons que la variable `N` est déclarée du type `Integer`[2]. Nous ne savons pas *a priori* combien il y aura de diviseurs, mais il ne peut y en avoir plus de `N`. Nous allons déclarer un type pour repérer les éléments dans le tableau ; bien entendu, la *nature* de ce type n'a aucun lien avec le type de `N` : nous choisissons donc un *type* différent, et non un sous-type. Pourquoi avoir choisi de le dériver de `Positive` plutôt que de `Integer` ? Tout type servant à compter ne peut *intrinsèquement* avoir que des valeurs positives ; en revanche, le fait que ce type soit limité à `N` provient de notre problème particulier. Il est donc logique d'exprimer cela en faisant dériver le type `Index` de `Positive`, puis en le contraignant à l'intervalle `1..N`.

> `Natural` et `Positive` sont deux sous-types prédéfinis de `Integer`, comportant respectivement les nombres positifs ou nuls et les nombres strictement positifs.

En revanche, les diviseurs que nous allons stocker sont de même nature que `N` : nous préférons donc déclarer `Diviseurs` comme un *sous-type* de `Natural`, et nous exprimons qu'en plus tous les diviseurs sont inférieurs ou égaux à `N`. Pourquoi avoir autorisé la valeur 0 ? Nous avons fait ici le choix de conception d'initialiser à 0 les éléments non significatifs du tableau (encore une fois ce n'est ni le seul, ni même certainement le meilleur choix possible). Nous devons donc autoriser cette valeur «de garde». Cependant, pour exprimer ce rôle spécial de la valeur 0, nous déclarons la constante `Non_Alloué`, qui nous servira plutôt que la valeur 0 elle-même là où ce sera nécessaire : nous exprimons ainsi que nous ne considérons pas 0 comme un diviseur.

Il ne nous reste plus qu'à exprimer qu'une liste est un tableau repéré par un `Index` de `Diviseurs` garni au départ uniquement de valeurs `Non_Alloué`. Nous avons choisi d'appeler `Prochain` la variable servant à repérer l'endroit où mettre le diviseur, pour bien marquer qu'elle désigne le prochain élément à remplir, et non le dernier rempli[3]. Remarquez qu'à ce niveau, notre préoccupation principale a été d'exprimer autant que possible au moyen du langage Ada la connaissance que nous avions du domaine de problème et des propriétés de la solution, notamment au niveau du typage.

Les notions utilisées à ce stade (tableau dynamique, boucle à compteur, types abstraits) sont encore assez éloignées de ce que peut connaître la machine. Celle-ci ne manipule que des types

---

1 Ou si nous ne sommes pas prêts à payer en temps d'exécution le prix de la facilité offerte par ce langage...

2 Plus vraisemblablement, dans un programme correctement écrit, elle serait déclarée d'un type dérivé d'`Integer`.

3 Noter qu'il existe un cas particulier où le programme tel qu'il est écrit lèvera l'exception `Constraint_Error`. Nous laissons à la perspicacité du lecteur le soin de le découvrir...

élémentaires (adresses, octets, mots mémoire) et des instructions très simples (tests et branchements). L'étape suivante va donc consister à *représenter les éléments abstraits* au moyen des *entités machine*. Si nous devions maintenant écrire ce programme en langage C, il deviendrait :

```
main()
{ int liste[100];
  int p;
  int *prochain;

   for (prochain = liste;
        prochain < liste+100;
        prochain++)
     *prochain = 0;

   prochain = liste;
   for (p = 1; p <= n; p++)
     if (n % p == 0)
       {*prochain = p;
        prochain++;
       };
}
```

Nous avons dû faire ici un choix d'implémentation supplémentaire : notre besoin est celui d'un tableau de taille choisie dynamiquement. C ne fournit pas cette possibilité[1] ; nous devons donc la réaliser au moyen d'un tableau de taille fixe. En général, l'espace supplémentaire sera perdu. Si au contraire l'espace est insuffisant, nous courons le risque de déborder du tableau et d'écraser les zones mémoire qui le suivent. Une bonne programmation exigerait donc de vérifier que le tableau ne déborde pas ; mais que ferait-on alors ? C ne nous fournit pas non plus de mécanisme d'exception. Nous devrions donc adopter une politique particulière de traitement d'exception (comme d'avoir systématiquement une variable de retour pour signaler si la fonction s'est bien terminée), politique qui devrait être traitée par l'appelant... On voit que le traitement complet des cas exceptionnels nous entraînerait vite très loin, et c'est pourquoi beaucoup de programmeurs C préfèrent prévoir largement les tableaux et prier pour qu'ils ne débordent pas...

Le fait que nous voulions une initialisation particulière du tableau ne peut plus être donné simplement : nous devons écrire une boucle pour expliquer à l'ordinateur *comment* initialiser le tableau. Il nous faut même expliquer *comment* réaliser cette boucle : initialisation, test de fin, incrément... Enfin, les différentes propriétés que nous connaissons sur les types de données sont perdues : nous ne manipulons plus que des adresses (pointeurs) et des `int`, c'est-à-dire des entiers *machine*.

Cette description utilise donc un niveau adapté au fonctionnement de l'ordinateur, mais encore suffisamment abstrait pour être indépendant d'une architecture machine particulière. La dernière étape consiste à traduire le programme en *instructions machine* correspondant à l'ordinateur cible, ce que l'on fait lorsque l'on travaille en assembleur. A ce niveau, toute notion de structure disparaît : des notions aussi simples que des boucles doivent être réalisées au moyen de tests et de branchements. La description appropriée est donc l'*ordinogramme* : un graphe représentant le parcours exact de la machine. De même, toute notion de typage disparaît, puisque l'on ne fait même plus de différence entre des nombres entiers, flottants ou des pointeurs, ni même entre structures de programmes et structures de données : les seules notions restantes sont des instructions, des adresses et des mots mémoire. La notion de tableau, par exemple, devra être réalisée au moyen de l'indexation pour accéder à des mots mémoire consécutifs. Le langage n'est plus à même d'effectuer aucun contrôle, puisque l'on n'est même pas sûr que ce qu'exécutera l'ordinateur correspond au texte du programme (un programme peut se modifier lui-même).

Nous voyons donc que la descente de niveaux sémantiques est essentiellement un passage du «quoi» (expression de besoin) au «comment» (solution au besoin). Bien entendu, il n'est nécessaire de spécifier le «comment» que si le langage ne fournit pas directement l'outil adéquat. Par exemple,

[1] Tout au moins pour des variables locales classiques, comme le permet Ada. Il faudrait faire appel à l'allocateur et gérer un niveau de pointeur supplémentaire.

nous avons eu l'enchaînement : besoin : liste (OK en SETL), réalisée par des tableaux de taille variable (OK en Ada), réalisés par des tableaux de taille fixe (OK en C), réalisée par une zone mémoire indexée. Remarquer que nous nous sommes arrêtés là parce que nous avons supposé que la machine fournissait directement l'abstraction nécessaire : sur une machine plus élémentaire, nous aurions pu devoir réaliser la notion d'indexation par un calcul d'adresse explicite et une indirection. Remarquons également que la descente dans les niveaux d'abstraction s'accompagne d'une *perte d'information* : la version SETL exprime quasiment directement le besoin ; en Ada, nous ne voyons plus explicitement la notion de liste. En C, toute l'information sur les relations logiques entre les différentes données est perdue. Enfin en assembleur, les variables deviennent totalement indifférenciées. Les physiciens (et les théoriciens de l'information) diraient que l'entropie du système augmente.

Si cette information n'est plus visible, il est possible de la reconstituer, de même qu'il est possible de faire diminuer l'entropie : mais comme en physique, cela coûte beaucoup d'énergie : il faut reconstruire le comportement dynamique du système (ce qui se passe à l'exécution), puis indentifier quelle structure abstraite de plus haut niveau pourrait avoir ce comportement pour implémentation. Inutile de dire que ce processus est nettement plus difficile que la démarche inverse qui consiste à passer de la structure abstraite à son implémentation. Ce qui nous amène à une autre règle d'or :

> *Tout comportement doit être décrit au moyen de la structure de plus haut niveau disponible.*

C'est dans ce cadre que l'on peut également régler la fameuse «polémique du `GOTO`», qui fit rage dans les années 70, mais qui n'est pas encore totalement évacuée pour certains. En gros, le `GOTO` est généralement considéré comme un ordre contraire aux bons principes de programmation, mais d'aucuns argumentent que finalement, les structures de contrôle ne sont que des `GOTO` déguisés. C'est exact : en fait le rôle d'un langage de haut niveau est précisément de *cacher* les structures de bas niveau, en leur attribuant un plus haut niveau sémantique. Donc si vous travaillez en un langage de bas niveau (Le défunt FORTRAN IV, le Basic «standard»), vous devrez utiliser des `GOTO`, et il n'y a pas de honte à cela, à condition que dans une étape de conception précédente vous ayez exprimé votre algorithme au moyen de concepts de plus haut niveau.

Insistons sur un point : quel que soit le langage de programmation utilisé, toutes les étapes que nous avons décrites devront être accomplies... mais pas nécessairement par le programmeur. Le compilateur travaille en procédant de la même façon : une phase d'analyse de haut niveau, suivie d'une phase d'expansion destinée à abaisser le niveau sémantique de la description, suivie enfin d'une phase de génération de code. Le niveau d'un langage correspond donc au niveau sémantique à partir duquel un outil automatique (le compilateur) est capable de prendre le relais du programmeur. Un langage est d'autant plus haut niveau qu'il prend le relais du programmeur plus tôt, c'est-à-dire qu'un plus grand nombre de ces étapes seront accomplies automatiquement.

Automatiquement, certes, mais accomplies tout de même ! Ceci explique pourquoi il est normal que, toutes choses égales d'ailleurs, il faille plus de temps pour compiler un programme écrit dans un langage de plus haut niveau. [Ber89] rapportent qu'il faut quatre fois plus de temps pour compiler un programme Ada qu'un programme Pascal faisant la même chose ! D'ailleurs, les compilateurs simples ne se contentent-ils pas d'une ou deux passes alors que les compilateurs Ada en ont souvent près de huit ? C'est simplement que partant de plus bas, les premiers ont moins de travail à faire. Ceci ne signifie pas que le travail correspondant à l'abaissement du niveau sémantique ne doive pas être fait : simplement il doit être accompli par le programmeur au lieu d'être pris en charge par le compilateur. Pour être honnête, il faut donc ajouter au seul temps de compilation des langages de bas niveau l'effort de conception supplémentaire (et le coût des erreurs introduites) dû à ce plus bas niveau. A ce moment, les avantages du haut niveau redeviennent évidents : [Ber89] rapportent que depuis que les équipes sur lesquelles ont été effectuées leurs mesures ont abandonné Pascal pour Ada, les factures mensuelles passées en temps de compilation ont *diminué*... simplement grâce au fait que de nombreuses erreurs qui n'étaient diagnostiquées qu'à

l'exécution (et qui nécessitaient plusieurs cycles de recompilation pour être identifiées) sont maintenant «piégées» dès la première compilation.

Notons enfin que le gros risque avec les langages de bas niveau est de court-circuiter des étapes. Nous avons vu comment abaisser progressivement le niveau sémantique d'un projet. Mais l'outil final (le langage) étant en général accessible au concepteur, celui-ci tendra à passer directement de l'expression de haut niveau au codage. Cela peut marcher pour des petits projets, mais lorsque la taille du saut (sémantique) augmente... on finit par se casser la figure.

## 5.3 Apport des langages de haut niveau

Les premiers langages de programmation étaient orientés machine : on forçait en quelque sorte le programmeur à penser avec le même vide culturel que la machine. Au moins était-on sûr qu'il n'y avait pas de risque d'interprétations divergentes. En fait, c'était bien entendu extrêmement pénible, ce qui a conduit à l'apparition de langages de plus haut niveau, c'est-à-dire plus proches de la façon de penser du programmeur. Ce faisant, on a également éloigné le niveau de lecture du programmeur du niveau de lecture de l'ordinateur : on a donc accru le risque de divergence entre les deux. Quel est donc l'intérêt des langages de haut niveau ?

Il provient essentiellement de ce que ces langages permettent de communiquer à l'ordinateur une partie du «savoir» supplémentaire que possède le programmeur. Autrement dit, au lieu de forcer le programmeur à comprendre le programme comme le fait le compilateur, on a la possibilité d'indiquer à celui-ci un certain nombre d'éléments «culturels» qui rapprocheront son analyse du texte de celle effectuée par l'être humain. Ceci permettra au compilateur d'effectuer des tests beaucoup plus nombreux, et surtout situés à un niveau sémantique bien supérieur. Le programmeur devra donc prendre soin de tenter de décrire, non plus les éléments de la machine dont il a besoin, mais les éléments de son domaine de *problème*. C'est pourquoi les types prédéfinis, qui appartiennent au domaine de la machine, devront (en général) être évités. Prenons un exemple.

Dans beaucoup de programmes, et quel que soit le langage de programmation, il est fréquent de trouver des déclarations comme :

```
Compteur : Integer;
```

Nous prétendons que cette déclaration est trompeuse, non maintenable, dangereuse, non portable, inefficace, et pour tout dire, opposée aux principes même du génie logiciel ! Pourquoi ?

Elle est trompeuse et non maintenable, car elle autorise des valeurs négatives pour `Compteur`, alors qu'un compteur ne peut contenir que des valeurs positives ou nulles. Bien sûr, le lecteur humain *sait* qu'un compteur ne peut être négatif, mais cette information n'est pas fournie au compilateur. Il n'y a donc aucune garantie que la variable sera effectivement utilisée comme compteur. Le programmeur de maintenance avisé devra alors résoudre une question difficile : est-ce que le programmeur d'application a utilisé le type `Integer` par simple paresse, ou n'y a-t-il pas quelque endroit où il a mis une valeur négative dans `Compteur` pour réaliser une «grosse astuce» ? Le seul moyen de résoudre la question est d'aller inspecter toutes les utilisations de la variable.

Elle est dangereuse et non portable, parce qu'elle ne se préoccupe pas de la question de la borne supérieure. Tout type de donnée possède nécessairement une limite aux valeurs qu'il peut stocker. Il est vraisemblable que le programmeur n'a voulu qu'une variable «entière», et n'a même pas songé à cette limite, dont la valeur dépend ici de l'implémentation. Trop souvent, les programmeurs déclarent des variables du type `Integer` simplement pour éviter d'avoir à penser au problème de la borne supérieure, comme s'il s'agissait réellement d'entiers mathématiques. Un tel programme, qui fonctionne parfaitement sur une machine 32 bits, peut avoir un comportement imprévisible sur une machine 16 bits.

Elle est inefficace, parce qu'une définition plus précise aurait permis au compilateur d'utiliser la mémoire de façon plus efficace, et même de générer un meilleur code. Supposons que cette

variable serve à indexer un tableau de 10 éléments ; si elle avait été déclarée avec une contrainte correspondant à sa vraie utilisation, le compilateur aurait été capable de n'utiliser qu'un octet de mémoire (au lieu de 2 ou 4), et surtout aurait pu éliminer toutes les vérifications de débordement lorsqu'elle était utilisée pour indexer le tableau.

- Plus que pour toutes ces excellentes raisons, elle est contraire aux principes du génie logiciel parce que tout le savoir qu'avait le concepteur des propriétés de sa variable n'a pas été écrit, et ne sera donc pas transmis aux personnes chargées de la maintenance.

Qu'aurait-il dû faire ? Etudier le domaine de problème et refléter ses connaissances dans l'expression du langage, ce qui aurait donné :

```
type Valeurs_à_compter is range 0..40_000;
Compteur : Valeurs_à_compter;
```

Noter que le compilateur aurait utilisé la précision de cette déclaration pour réserver juste l'espace nécessaire : 16 bits si la machine disposait d'entiers non signés, 24 ou peut-être 32 sinon, mais de toute façon le type le plus économique permettant de répondre au besoin exprimé.

Ceci n'est pas possible avec un langage qui travaille au niveau machine ; quel type aurions-nous choisi en C, s'il nous fallait être portable ? Certainement pas `int` (et encore moins `short`), car même si l'on peut supposer qu'il soit au moins sur 16 bits sur la plupart des machines, on ne peut supposer plus, et c'est insuffisant ici. `unsigned` ? Le langage n'offre aucune garantie qu'il dispose d'une étendue plus grande que `int`. Nous aurions dû nous rabattre sur `long` (ou mieux `unsigned long`), avec le risque de réserver 64 bits par variable sur certaines machines, là où 16 auraient suffi.

### *5.3 .1 Les types de données abstraits*

Puisque nous voulons désormais travailler sur les éléments du domaine de problème, il nous faut utiliser non des types machine, mais des *abstractions* d'éléments du monde réel. Les types de données correspondants sont appelés *types de données abstraits*. Un tel type est caractérisé par un ensemble de *valeurs* et un ensemble d'*opérations* qui lui sont applicables.

Si cette notion de type de donnée abstrait est commode pour l'utilisateur, elle est fort éloignée de ce que peut traiter un ordinateur : des nombres. Par conséquent, chaque attribut et chaque opération doivent être *implémentés* au moyen d'éléments de plus bas niveau. Il existe donc deux vues différentes d'un type de donnée abstrait : la vue abstraite, ou externe, qui exprime les propriétés utilisables de l'abstraction (ce que l'on appelle dans la terminologie Ada la *spécification*) et les mécanismes utilisés pour concrétiser cette abstraction (pour Ada, l'*implémentation*). Comme pour tout élément logiciel, la solution au problème de l'implémentation d'une abstraction donnée n'est jamais unique ; cependant, toutes les solutions possibles doivent être considérées comme équivalentes du point de vue de l'utilisateur, tant qu'elles fournissent des abstractions sémantiquement équivalentes[1].

En principe, il devrait donc être possible de remplacer n'importe quelle implémentation d'une abstraction par une implémentation différente sans perturber les utilisateurs. Si une spécification dépend malencontreusement d'un mécanisme d'implémentation particulier, cela s'appelle une *surspécification* : la spécification est *trop* précise, car elle mentionne des éléments qui n'appartiennent plus à la vue abstraite. Attention : une spécification peut imposer des contraintes (en temps d'exécution, en mode de gestion de la mémoire) qui interdisent certaines implémentations : il n'y a pas surspécification tant que ces contraintes résultent du besoin extérieur, et non d'une vue *a priori* d'une méthode d'implémentation particulière.

Inversement, si un utilisateur fait appel à des propriétés d'un type de donnée abstrait qui ne font pas logiquement partie de l'abstraction, on dira qu'il *viole l'abstraction*. Dans un cas comme dans

[1] Sauf éventuellement du point de vue des performances.

l'autre, il devient impossible de remplacer une implémentation par une autre, et l'indépendance entre spécification et implémentation est perdue.

## *5.3 .2 Le langage comme outil de vérification*

Eviter toute surspécification est une tâche difficile qui requiert un grand talent d'abstraction, mais prévenir les violations d'abstraction peut être obtenu par les vérifications effectuées par le langage. Est-ce si important que le langage vérifie cela ? Et à quoi cela sert-il après tout que le langage effectue des vérifications ?

Reprenons le problème à la base. A un moment donné, l'ensemble des valeurs de toutes les variables d'un programme constitue ce que l'on appelle le *vecteur d'état*. Le plus vaste vecteur d'état comporte toutes les combinaisons de bits possibles pour toutes les variables du programme. Nous appellerons cet ensemble les états matériels. Mais seul un sous-ensemble de celui-ci est autorisé par les règles d'un langage de programmation de haut niveau, puisque le compilateur effectue (au moment de la compilation ou de l'exécution) un certain nombre de vérifications, qui empêchent certaines combinaisons de se produire. Nous appellerons ce sous-ensemble les *états autorisés*. A l'intérieur de ce sous-ensemble, un ensemble encore plus réduit est constitué des états accessibles par une exécution correcte du programme. Nous appellerons ce dernier ensemble les *états corrects*. Ces différents états sont résumés dans la figure 8.

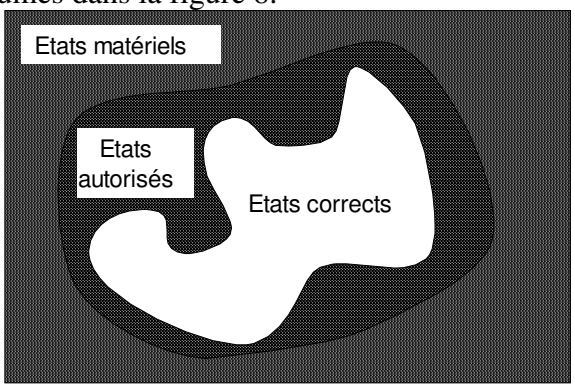


**Figure 8 :** Les différents états d'un programme

Les états matériels doivent nécessairement englober les états autorisés, autrement le langage ne serait pas compilable (sur cette machine). De même, les états autorisés doivent inclure les états corrects, autrement le programme désiré ne pourrait être écrit dans le langage. Notez cependant que les états autorisés ne peuvent coïncider exactement avec les états corrects : cela reviendrait à prouver formellement le programme. Il y a donc nécessairement des états incorrects, correspondant à la zone des états autorisés n'appartenant pas aux états corrects, qui *peuvent* être atteints en fonction des seules règles du langage ; c'est la responsabilité du programmeur de faire en sorte que ceci ne se produise jamais durant l'exécution du programme. D'une certaine façon, un «bug» est une brèche par laquelle on atteint un état incorrect.

C'est sur ce schéma que l'on comprendra le mieux la différence fondamentale de philosophie entre les langages C/C++ et Ada. C a été conçu comme une alternative à l'assembleur ; ses concepteurs l'ont même qualifié d'«assembleur portable». Comme en assembleur, l'idée était de permettre de faire faire «ce que l'on voulait» à la machine ; la crainte était que le langage empêche le programmeur de faire ce qu'il souhaitait. Le langage a donc cherché à étendre au maximum les états autorisés pour les rapprocher au maximum de l'ensemble des états matériels. C'est en ce sens que beaucoup considèrent que C est un langage très puissant : il permet de *tout* faire.

Tout, y compris des catastrophes... L'approche suivie par Ada est radicalement différente. Le but était d'avoir un langage fiable et de haut niveau pour des applications critiques. Il fallait pour cela non seulement fournir des outils permettant la définition de vues abstraites, mais également interdire l'accès aux détails de l'implémentation afin d'éliminer la plus grande partie des états

incorrects. Le but recherché était l'inverse de celui de C : il s'agissait de resserrer l'ensemble des états autorisés autour des états corrects, éliminant ainsi autant d'états incorrects que possible. Vu ainsi, on peut (et même on doit) se poser la question : à quoi cela pourrait-il servir d'autoriser des états incorrects ? La vraie question n'est donc pas pourquoi faudrait-il contraindre, mais quel pourrait être l'intérêt de ne *pas* contraindre.

Cette approche présente tout de même un risque : il peut arriver qu'à force de restreindre les états autorisés, on ampute une petite partie des états corrects. Autrement dit, le langage ne laisse pas, au nom de règles de sécurité, le programmeur faire ce qui lui est nécessaire. Certaines échappatoires (`Unchecked_Conversion` par exemple, nous y reviendrons) ont été prévues au niveau du langage, mais malgré tout il peut arriver que le langage interdise certaines pratiques qui bien qu'utiles ou même nécessaires dans certains cas, seraient trop dangereuses en général. Le programmeur devra donc *changer sa conception* pour se plier aux règles générales. Cette nécessité (qui paraît révoltante au programmeur C !) ne paraîtra choquante que dans le monde du logiciel. Toutes les autres branches de l'industrie sont coutumières du fait, et savent qu'elles doivent ajuster leurs conceptions à des règles de sécurité dont l'applicabilité est parfois discutable compte tenu des particularités du contexte, mais qui ont force de loi. Songez par exemple que le défunt Concorde était muni de doubles commandes mécaniques, dont il était de notoriété publique qu'elles étaitent quasiment inutilisables dès que l'avion avait pris une certaine vitesse... et qui ont coûté très cher dans le bilan de poids de l'appareil. Mais personne n'aurait osé prendre la responsabilité de les supprimer, *et c'est normal*, car il s'agissait d'un dispositif de sécurité fondamental.

Quel est l'avantage d'une telle approche ? Si on recherche une erreur dans un programme qui compile, on peut être assuré que les règles du langage sont respectées. Autrement dit, on peut garantir que le vecteur d'état du programme appartient toujours à l'ensemble des états *autorisés*[1]. En réduisant le nombre d'états autorisés mais incorrects, un langage rigoureux diminue la probabilité d'erreurs (moins d'états incorrects sont accessibles) et facilite la maintenance (moins d'états incorrects sont à envisager lors de la recherche d'erreurs). De plus, le compilateur pourra faire usage de cette connaissance supplémentaire pour optimiser le programme. Pour prendre un exemple concret, si l'on doit manipuler des couleurs, il est possible depuis Pascal (le langage !) de définir un type énumératif :

```
type Couleurs is (Noir,  Rouge,   Bleu, Vert,
                  Jaune, Magenta, Cyan, Blanc);
```

En faisant ainsi, on garantit qu'il n'est pas possible de multiplier deux couleurs, ni d'affecter à une couleur une valeur numérique incorrecte. Il paraît équivalent de déclarer :

```
Noir  : constant := 0;
Rouge : constant := 1;
...
```

Après tout, n'est-ce pas ce que fait le compilateur ? Et cela marchera aussi bien que le type énumératif, tout au moins *tant que personne ne tentera de violer l'abstraction*. Ce que l'on perd, c'est la vérification par le compilateur que *seules* les propriétés abstraites sont disponibles. Représenter des couleurs par des nombres est une façon d'*implémenter* la notion abstraite de couleur, ce qui est le travail du compilateur. Si vous le faites vous-même, vous autorisez l'utilisateur de l'abstraction à travailler au niveau de l'implémentation plutôt qu'au niveau abstrait, et vous ouvrez ainsi une voie vers des états incorrects.

Enfin, l'avantage peut-être le plus important de l'approche de haut niveau est le contrôle qu'elle peut exercer sur la conception. Puisque maintenant le compilateur a de l'information sur les éléments de plus haut niveau du problème, une faute de conception se traduira souvent par une incohérence au niveau du typage qui sera interceptée par le compilateur. Combien de fois avons-nous vu des programmeurs nous appeler à la rescousse pour un problème de langage, alors qu'il s'agissait en fait d'un problème de conception ! En voici un exemple typique :

[1] Sauf en cas de bug du compilateur. Ceux-ci sont suffisamment rares pour que l'on puisse faire cette hypothèse, mais c'est ce qui rend les erreurs dues aux compilateurs si difficiles à trouver.

Un étudiant débarqua un jour dans mon bureau en grognant : «Ada est vraiment un langage stupide. Regardez ça.» Il travaillait sur le simulateur d'un ordinateur un peu particulier. Il avait utilisé un tableau d'`Integer` pour représenter la mémoire. La mémoire était divisée en pages de 512 mots. Il avait donc les déclarations suivantes :

```
type Mémoire is array (0..32767) of Integer;
type Page    is array (0..511)   of Integer;

M : Mémoire;
P : Page
```

Le problème était que le compilateur refusait l'affectation suivante :

```
P := M(0..511);    -- Tranche des 512 premiers éléments de M
```

alors que la boucle suivante, qui était sémantiquement strictement équivalente, était acceptée :

```
for I in 0..511 loop
   P (I) := M (I);
end loop;
```

Evidemment, du point de vue du langage, il y avait de bonnes raisons. La première affectation s'effectuait entre une variable de type `Page` et une expression de type `Mémoire` ; comme il s'agissait de deux types différents, l'affectation était interdite. Dans le second cas, on n'affectait que des *composants*, tous du type `Integer`, et chacune des affectations était autorisée.

Je ne tentai même pas de m'aventurer dans une telle explication, car l'étudiant aurait pensé (comme le lecteur le pense actuellement) que le typage fort ne fait qu'apporter des complications au malheureux utilisateur en l'obligeant à écrire une boucle explicite. La conversation continua comme ceci :

*Moi :* Pourquoi avoir déclaré le type `Page` ?
*Etudiant* : Parce que j'en avais besoin !
*Moi* : Pensez-vous qu'une page soit une entité de *nature différente* d'une mémoire ?
*Etudiant* : Hmm... Non, c'*est* une sorte de mémoire.
*Moi* : Quelle est donc la différence ?
*Etudiant* : Une page fait toujours 512 mots, alors que la mémoire peut avoir n'importe quelle taille *a priori*.
*Moi* : Ah ! Comment peut-on exprimer ceci en Ada ?
*Etudiant* : J'y suis : une `Mémoire` est un type tableau non contraint, et `Page` est un sous-type contraint de `Mémoire` !

Il modifia son programme de la façon suivante :

```
type Mémoire is array (Natural range <>) of Integer;
subtype Page is Mémoire (0..511);

M : Mémoire(0..32767);
P : Page;
```

et dès lors, son affectation fonctionna sans problème, puisque `P` et `M` appartenaient au même type. Que pouvons-nous conclure de cet exemple ? Le programme de l'étudiant comportait une faute de conception, puisque l'utilisation qui était faite des types ne correspondait pas aux abstractions souhaitées, ce qui avait provoqué l'erreur de compilation. Le mouvement naturel de l'étudiant avait été de s'en prendre au langage et de chercher un moyen de tourner le contrôle. Il y était arrivé en ne travaillant pas globalement sur les objets, mais au niveau des composants. Remarquez que cela était possible parce que le type `Mémoire` était visible : s'il s'était agi d'un type privé, il n'aurait pas pu s'en sortir aussi aisément. En fait, il était descendu d'un niveau d'abstraction, puisqu'il ne travaillait plus globalement sur la `Mémoire`, mais sur la façon dont cette mémoire était représentée. A ce plus bas niveau, la sémantique était plus pauvre, et il avait pu violer l'abstraction.

Cet exemple montre bien comment un langage très strict peut piéger les fautes de conception ; si une telle situation était apparue en C++ (en C, aucun contrôle de ce type n'est possible), il est

vraisemblable que l'étudiant se serait contenté d'un vigoureux forçage de type (*type cast*) pour obliger la compilation à passer... et cacher le problème de conception. Dans le contexte du génie logiciel, où fiabilité et facilité de maintenance sont primordiaux, le langage de programmation doit fournir des contrôles rigoureux pour éliminer un maximum d'états incorrects dès l'étape de compilation. On peut résumer ceci ainsi :

> *Ce qui fait la valeur d'un langage de programmation, ce n'est pas ce qu'il autorise, c'est ce qu'il interdit.*

### *5.3 .3 Autres formes de protection*

La programmation est un domaine plein de pièges subtils, dont certains requièrent une grande attention pour être évités... à moins que le langage ne prenne en charge les problèmes. Si les types abstraits sont l'outil de base de protection du programmeur, il existe d'autres cas où une formulation de plus haut niveau permet d'obtenir une meilleure sécurité.

Un exemple caractéristique est celui des boucles en C. Prenez un programmeur C moyen (ou même expérimenté), et demandez-lui comment il écrirait l'équivalent de la boucle Ada :

```
for I in A .. B loop
   ...
end loop;
```

Il vous répondra immédiatement :

```
for (I = A; I <= B; I++) {
   ...
}
```

Demandez-lui alors ce qui se passe si, par exemple, `I` est un octet (ou plus exactement un `char`, puisqu'en C les caractères sont des nombres !), que `A` vaut 0 et `B` vaut 255. Ce cas de figure n'a rien d'exceptionnel : il s'agit d'une simple boucle sur l'ensemble des caractères. Horreur ! Lorsque I atteint 255, il repasse à 0, car il n'y a pas de notion de débordement en C[1], et la boucle ne s'arrête jamais ! Ce qui est surprenant (et inquiétant), c'est que nous avons tenté cette expérience avec de nombreux programmeurs C, et qu'*aucun* n'a jamais pensé au problème ! On est donc en droit de penser que tous les programmes C marchent par hasard, parce qu'on ne s'aventure jamais trop près des cas limites... Bien entendu, des tests de la suite de validation Ada assurent que la boucle produite par le compilateur marche toujours correctement dans ce cas de figure. Le programmeur Ada est donc protégé par le langage d'une erreur grave, sans même avoir à s'en préoccuper.

Cet exemple montre bien que le langage peut effectivement fournir un degré de sécurité supplémentaire qui n'a rien à voir avec un quelconque problème de conception. Remarquons qu'encore une fois, ce résultat est obtenu par le fait que la description Ada se situe un niveau d'abstraction au-dessus de la description C : le programmeur décrit ce qu'il veut obtenir (une boucle de 0 à 255) et non comment l'obtenir (par une affectation, un test et un incrément – algorithme dont nous venons de montrer qu'il était faux)[2].

## 5.4 Et l'efficacité ?

Si l'utilisation de fonctionnalités de haut niveau améliore indiscutablement la lisibilité, la sécurité et la maintenabilité, l'on est en droit de se demander si cela n'a pas un effet adverse sur l'efficacité.

---

[1] En toute rigueur, la norme C n'interdit pas de «planter» le programme dans ce cas, mais la plupart des compilateurs ne le font pas. Inutile de penser à un traitement d'exception (même en C++, qui a pourtant les exceptions, les débordements ne sont pas traités).

[2] Signalons au lecteur qui chercherait la solution correcte en C qu'il n'est pas possible de coder une boucle **`for`** (au sens d'Ada) au moyen d'une simple boucle **`while`** (dont le **`for`** C n'est qu'une variante). Il faut nécessairement un **`exit`** (**`break`** en C).

En fait, l'efficacité est souvent invoquée comme raison d'utiliser des langages de plus bas niveau. Voyons donc plus précisément ce problème.

Tout d'abord, il faut savoir de quelle efficacité l'on parle. Il existe souvent des contraintes temporelles au cahier des charges : celles-là doivent être impérativement respectées, car un programme qui ne répond pas à son cahier des charges est un programme *faux*. La question de l'efficacité se pose seulement après : si le programme vérifie largement ses contraintes, comment va-t-on utiliser la marge supplémentaire ? Les machines utilisées au début de l'informatique étaient incroyablement lentes par rapport aux machines (mêmes individuelles) actuelles : on disait alors que le bon programmeur était celui dont les programmes allaient le plus vite possible. A l'époque, les programmes étaient de taille plus modeste qu'aujourd'hui, et l'on se souciait peu de la maintenance... Cette mentalité n'a plus lieu d'être aujourd'hui : un bon programme, c'est d'abord un programme évolutif, fiable et facile à maintenir. De plus, la puissance des machines rend acceptables des solutions autrefois impossibles. Ceci ne signifie pas que l'efficacité ne doive pas être recherchée, mais seulement que, encore une fois, il faut trouver le bon compromis, car ce que l'on gagne en efficacité, on le perd sur d'autres aspects, tels que lisibilité et portabilité. Par exemple, imaginons un programme interactif (comme une interrogation de base de données). S'il donne la réponse au bout de dix secondes, c'est inacceptable (et vraisemblablement trop pour les contraintes du cahier des charges) : il faut faire quelque chose. Si on améliore les performances pour que les réponses parviennent en moins d'une seconde, c'est bien ; l'attente est sensible pour l'utilisateur, mais acceptable. Si l'on fait passer ce temps à un dixième de seconde, c'est parfait : la réponse semble instantanée. Si l'on poursuit dans cette voie pour obtenir un temps de réponse d'un centième de seconde, *on perd son temps* : l'utilisateur ne sera pas sensible à la différence, et on risque de le payer cher sur d'autres aspects.

D'autre part, il n'est pas du tout évident qu'une description de plus haut niveau se traduise nécessairement par une perte d'efficacité ; au contraire, le compilateur est capable d'utiliser le supplément d'information pour mieux optimiser le code. Considérons cet exemple :

```
   I   : Integer; -- Ce qu'il ne faut pas faire !
   S1  : String(1..10);
   S2  : String(1..10);
begin
   I       := Calcul_compliqué;        -- (1)
   S1(I)   := S2(I);                   -- (2)
```

Il n'y a pas de vérification au moment de l'affectation à `I` en (1), mais comme l'on ne sait pas *a priori* quelle est la valeur retournée, il faut vérifier à chaque utilisation (deux fois en (2) ) que la valeur est correcte. Si nous informons le compilateur de notre intention d'utiliser cette variable pour indexer des chaînes comme ceci :

```
subtype Index is Integer range 1..10;
   I   : Index;
   S1  : String (Index);
   S2  : String (Index);
begin
   I       := Calcul_compliqué;        (1)
   S1(I)   := S2(I);                   (2)
```

alors le compilateur effectue une vérification lors de l'affectation en (1), mais il n'est plus nécessaire d'en effectuer lors des utilisations en (2). Comme on utilise les variables plus souvent qu'on ne les modifie, cette approche est généralement préférable. C'est tellement vrai que beaucoup de programmeurs sont déçus lorsqu'ils tentent d'accélérer leurs programmes en les compilant «sans vérification» : le gain excède rarement quelques pour-cents. Ceci signifie simplement que l'optimiseur a été capable d'éliminer de lui-même tous les tests redondants... et donc que le (petit) gain d'efficacité se fait certainement au détriment des tests réellement importants, donc avec un impact maximal sur la fiabilité ! Conséquence : en général, en Ada, on laisse toutes les vérifications *même dans la version finale, commercialisée, du programme*.

Il faut savoir que les techniques modernes d'optimisation permettent d'obtenir des résultats surprenants. Nous avons ainsi connu un programmeur qui avait écrit un programme de façon «ignoble», persuadé que cela irait plus vite. Il reçut du compilateur le message suivant :

```
WARNING: frame of control too complicated; optimizer gives up.[1]
```

Du coup, le programme allait moins vite que s'il avait écrit son code proprement et laissé l'optimiseur faire son travail... A travers cet exemple, on notera la *différence d'état d'esprit* : le rôle du compilateur C est de fournir une traduction directe vers les instructions machine, l'optimisation étant du ressort du programmeur. Du coup, un compilateur C, même relativement simple, produira un code acceptable – d'où la réputation d'efficacité traditionnellement attachée au C. Inversement, le rôle du compilateur Ada est de libérer (autant que possible) le programmeur des contraintes de bas niveau ; un compilateur sans optimiseur performant serait catastrophique, mais le langage a été conçu pour permettre des optimisations très perfectionnées. Il est donc possible d'optimiser beaucoup plus du code Ada que du code C (cf. [Tar93] pour un exemple de programme initialement développé en C qui n'a pu tenir ses performances qu'après un recodage en Ada).

Enfin, l'on ne saurait trop rappeler que la recherche d'efficacité doit se faire d'abord par la recherche d'algorithmes performants. Les bénéfices que l'on peut en retirer sont de plusieurs ordres de grandeur supérieurs à ce qui peut être obtenu par des «astuces» de codage. Le programmeur doit toujours garder à l'esprit la règle des 90/10 : un programme passe 90% de son temps dans 10% de son code ; or ces fameux 10% sont en général très difficiles à identifier. La seule constante que nous ayons rencontrée dans des projets où se posait un problème d'efficacité est que le point critique n'était *jamais* là où le supposait le programmeur[2]. Il est indispensable de disposer d'outils de mesure lors de toute recherche d'amélioration des performances.

En conclusion, nous dirons que l'efficacité est importante, mais que ce n'est plus le seul critère à prendre en compte pour juger de la qualité d'un programme, et que les principes du génie logiciel continuent à s'appliquer même pour l'écriture de programmes critiques sur le plan des performances. Quand l'on voit la fréquence avec laquelle l'efficacité sert de justification abusive à des pratiques peu recommandables, on ne peut que conclure par une paraphrase :

*Efficacité, que de bugs on commet en ton nom !*

## 5.5 Conclusion

Le typage fort et la vérification par le langage des abstractions sont des dispositifs de sécurité indispensables : ils ne peuvent être pris à la légère. Cette démarche est universellement adoptée dans d'autres branches de l'industrie. Par exemple, une machine dangereuse comme un massicot ne peut fonctionner que si l'opérateur appuie simultanément sur deux boutons, disposés de telle façon qu'il soit obligé d'utiliser ses deux mains : on est ainsi assuré qu'il n'y a pas une main qui traîne sous la lame. On pourrait penser que l'opérateur d'un massicot a une perception beaucoup plus directe du risque qu'il y aurait à laisser une main dans la machine que le programmeur d'utiliser un forçage de type ; il est cependant connu qu'en l'absence de dispositif de sécurité, des accidents se produisent. Il n'y a pas de raison de supposer que les choses se passeraient différemment en programmation. Remarquons au passage que l'utilisation de dispositifs de sécurité dans l'industrie ennuie les opérateurs et diminue bien souvent la productivité ; mais l'on considérerait comme inacceptable de sacrifier la sécurité à de tels impératifs.

Donner au développement de logiciel un caractère industriel demande que l'on reconnaisse la nécessité d'un contrôle plus pointu, et que la sécurité des systèmes complexes ne peut pas ne dépendre que du talent des individus qui les conçoivent. Nous conclurons en illustrant cette

[1] AVERTISSEMENT : Structure de contrôle trop compliquée; l'optimiseur abandonne.

[2] Nous avons connu ainsi un compilateur Pascal qui passait l'essentiel de sont temps... dans la boucle de lecture de caractères. La réécriture en assembleur de la seule procédure de lecture d'un caractère a permis de *doubler* la vitesse du compilateur.

différence fondamentale dans la perception du rôle du programmeur par le rapprochement de deux citations tirées respectivement d'introductions aux langages C et Ada :

> *C a été conçu dans l'idée que le programmeur est quelqu'un de raisonnable et qui sait ce qu'il fait.*
>
> *Ada a été conçu en tenant compte du fait que le programmeur est un être humain.*

## 5.6 Exercices

1. En C++, il est possible de définir des classes fournissant la notion de tableau avec contrôle de débordement. Quels en sont les inconvénients par rapport à des tableaux vérifiés par le compilateur ? Discuter sur le plan des principes du génie logiciel et sur le plan des optimisations possibles du code généré.
2. Rechercher dans les langages autres qu'Ada les restrictions qui sont d'ordre méthodologique et celles dues à la technique du compilateur. Que peut-on en conclure ?
3. Nous avons mis en garde le lecteur contre l'utilisation du type `Integer`. Ces arguments sont-ils applicables au type `String` ? Au type `Boolean` ?
4. Ecrire au moyen d'une boucle **`loop`** simple l'équivalent exact de la boucle **`for`** d'Ada. Attention, cet exercice est plus difficile qu'il n'y paraît !

# 6
# Liaison entre méthode et langage

Lorsque l'on doit intervenir sur un programme existant pour corriger des erreurs, lui apporter des améliorations ou lui faire subir une révision majeure, on est amené à intervenir sur le code, bien sûr, mais à l'intérieur d'un cadre qui provient de la méthode. Comme toujours, nous considérons que celui qui effectue la maintenance n'est pas le concepteur initial. Il faut donc commencer par comprendre la philosophie générale de la conception, puis déterminer comment l'évolution pourra être effectuée sans perturber la structure générale, ou, si ce n'est pas possible, faire évoluer la conception elle-même. L'énorme différence par rapport à ce qui se passe en conception initiale est qu'à ce moment, le code existe déjà. Il importe donc d'être capable de déterminer l'impact de toute modification de la conception sur le code. Ce problème porte le nom général de traçabilité : être capable de «suivre à la trace» (dans les deux sens) les liens qui existent entre la conception et le code.

## 6.1 Le problème de la documentation

L'idée la plus naturelle est de considérer que la documentation est là justement pour assurer cette traçabilité. Depuis des années, la question de la documentation des conceptions a été centrale à la démarche de rationalisation de la production de logiciel... et toujours aussi difficile à obtenir :

> *La documentation est l'huile de ricin de la programmation : cela doit bien servir à quelque chose, puisque les chefs de projets insistent tant pour en avoir....*

Nous avons vu que la bonne compréhension *par un être humain* d'un programme s'appuyait sur une part «culturelle», c'est-à-dire sur des éléments ne figurant *pas* dans le texte du programme. Une bonne partie de la difficulté de reprendre le programme de quelqu'un d'autre tient à ce phénomène : la «culture» d'un programmeur est différente de celle d'un autre. Même si l'on reprend son propre programme quelques mois après l'avoir écrit, il faut un certain temps avant de «reconstituer le contexte» nécessaire à sa compréhension. On comprend mieux alors le rôle de la documentation : elle permet de transmettre cette information nécessaire à la compréhension du programme qui ne se trouve pas dans le texte du programme. Cela explique aussi pourquoi les programmeurs sont généralement réticents à l'écrire, et pourquoi elle est généralement si mal faite : elle doit mettre noir sur blanc ce qui paraît absolument évident à son auteur... mais pas forcément à quelqu'un d'autre ; le programmeur aura donc du mal à identifier ce qui posera problème à un futur mainteneur doté d'une autre «culture», et risquera d'insister lourdement sur des détails inutiles, tout en laissant de côté des points fondamentaux. Nous avons été ainsi amené à relire des spécifications pour un système de menus. La documentation fournissait abondance de détails sur toutes les interactions possibles, parlant par exemple de saisie d'éléments de menu par l'utilisateur, chose qui nous a paru totalement contradictoire avec la notion même de menu... jusqu'au moment où nous avons compris que ce que le rédacteur avait appelé *menu* était ce que nous appelions *masque de saisie*. Le programmeur avait tout simplement oublié de décrire la nature même de ce dont il parlait.

La documentation souffre d'un autre problème : même si elle existe, il convient de la mettre à jour au fur et à mesure de l'évolution du projet ou des modifications dues à la maintenance. Un programmeur de maintenance consciencieux devrait rajouter à la documentation tous les points qu'il aurait souhaité y trouver et qu'il a dû reconstituer *à partir du programme* à grand-peine... c'est rarement le cas.

Ce problème est rendu encore plus aigu par le fait que la plupart des méthodes confondent démarche de conception et documentation. Lors de la conception, un certain nombre de documents sont produits qui expriment l'état d'avancement du projet et de la réflexion des concepteurs. Concrètement, une méthode se traduit de façon visible par la production de ces documents. Pour les concepteurs, cette documentation devient en quelque sorte obsolète une fois le projet entré en phase de codage, et les programmeurs éprouvent rarement le besoin de la remettre à jour en cas de modification apparaissant tard dans le processus de développement. La documentation nécessaire par la suite doit se donner pour but de permettre à une personne nouvelle dans un projet de comprendre sa structure et les décisions de conception qui ont été prises ; il n'y a aucune raison *a priori* que celle-ci coïncide avec les documents de la méthode, qui reflètent plutôt l'historique de la conception. Mais comme cette seconde documentation existe rarement, la première en tient lieu.

La documentation est donc rarement complète, appropriée et à jour. C'est un point faible du développement logiciel, mais elle est indispensable pour porter l'information qui ne peut s'exprimer directement dans le texte du programme. Si l'on pouvait exprimer *tout le contexte culturel du programmeur* dans le programme lui-même, et si le langage était d'assez haut niveau pour correspondre directement aux éléments de la conception, alors la documentation de programmation deviendrait inutile.

## 6.2 Vers l'autodocumentation

Encore une fois, ce sont les éléments tellement «évidents» (pour le concepteur initial) qu'ils ne figurent nulle part qui sont la cause d'une grande partie des difficultés de maintenance. Par conséquent,

*Tout le savoir du programmeur doit être exprimé dans le texte du programme.*

De quelle façon peut-on formuler ce savoir ? La première idée qui vient à l'esprit est d'utiliser dans ce but les commentaires. Nous parlons ici bien entendu des commentaires algorithmiques ; les en-têtes de modules (auteur, historique, etc.) sont indispensables et jouent un rôle différent. Hélas, ils souffrent des mêmes défauts que la documentation séparée, hormis le fait qu'ils figurent textuellement dans le corps de programme, ce qui facilite leur mise à jour simultanée lors de modifications ; en particulier, il est possible de modifier le programme sans mettre à jour les commentaires correspondants. En cas de désaccord, le programmeur croira toujours le code contre le commentaire. [Lan91] conseille même, lorsque l'on a à intervenir dans un code écrit par quelqu'un d'autre, d'*ignorer* systématiquement tous les commentaires : en effet, si l'auteur a fait une erreur, le commentaire risque d'induire le lecteur dans la même erreur ! On peut même dire qu'un commentaire est un aveu de faiblesse de la part du programmeur : s'il éprouve le besoin de clarifier les choses, c'est qu'il n'a pas écrit son programme de façon qu'il soit directement compréhensible.

*Le commentaire est la plus mauvaise forme d'expression du savoir du programmeur.*

Les possibilités de typage très rigoureux d'Ada ouvrent une nouvelle voie : exprimer une partie du savoir sous une forme compilable, donc vérifiable par le compilateur. Supposons par exemple que nous voulions un type destiné à compter quelque chose, et considérons les déclarations suivantes :

```
type    Compte_1 is new Integer   range 0..1000;
type    Compte_2 is new Natural   range 0..1000;
type    Compte_3 is new Comptable range 0..1000;
subtype Compte_4 is     Comptable range 0..1000;
```

Selon toute vraisemblance, le code généré par ces quatre déclarations sera exactement le même ; il n'est donc pas question ici de considérer des différences d'efficacité. La déclaration de `Compte_1` ne nous apporte aucune information supplémentaire. `Compte_2` nous montre que le programmeur voit effectivement son type comme un compteur, puisqu'il le dérive d'un type qui ne peut intrinsèquement pas être négatif. `Compte_3` nous apporte une information supplémentaire : c'est un type dérivé de `Comptable` ; il porte donc une dépendance conceptuelle à ce type (en particulier, ceci nous informe – et le compilateur vérifiera – que `Compte_3` ne peut pas s'étendre au-delà de `Comptable`). Il s'agit cependant d'un type de nature différente, puisque c'est un type dérivé ; alors qu'avec `Compte_4`, il ne s'agit que d'un sous-type, donc d'entités de même nature que `Comptable`.

On voit bien sur cet exemple que le fait d'avoir plusieurs façons d'exprimer la même chose devient une aide, car cela permet de transmettre un renseignement supplémentaire au lecteur : s'il y a un choix, le fait d'adopter une solution plutôt qu'une autre est porteur d'information. Plus les possibilités de typage du langage seront riches, plus il y aura de choix d'implémentations possibles, donc d'information transmise à la fois au compilateur (qui pourra faire des vérifications plus précises) et au lecteur. Ceci fonctionne remarquablement bien... à condition que le programmeur ait effectivement utilisé toutes les possibilités du typage.

Si l'on adopte cette façon de faire, il ne doit plus rester dans un programme que deux formes de commentaires : les en-têtes de module (de format standardisé, selon les normes du projet), et l'expression d'invariants, éléments supposés toujours vrais à un endroit du programme. Et encore, ces derniers peuvent faire l'objet d'une expression dans le langage (et donc d'un contrôle automatique). Il suffit de disposer de la procédure suivante (on peut en faire des versions plus perfectionnées) :

```
procedure Assert (Condition : Boolean) is
begin
   if not Condition then
       raise Program_Error;
   end if;
end Assert;
```

On peut alors exprimer les invariants sous la forme :

```
Assert( Taille (Pile) >= 2 );
```

Si l'invariant n'est pas satisfait, il y aura levée de l'exception `Program_Error` : encore une fois, les hypothèses du programmeur peuvent être vérifiées automatiquement par le langage.

En résumé, les possibilités de typage d'Ada permettent de transmettre beaucoup plus d'information au lecteur que d'autres langages. Ce n'est bien entendu pas une garantie ; mais en pratique, on s'aperçoit rapidement, lorsque l'on reprend un programme écrit par quelqu'un d'autre, à quel point l'on peut faire confiance aux déclarations. Et de toutes façons, la situation ne peut être pire qu'avec les autres langages où ce type de documentation n'est pas possible. Cela demande plus de travail au programmeur ? Certes, mais pas plus que de mettre des commentaires, pour un niveau de sécurité bien meilleur ; et comme le remarquait [Lan91] :

> *Un programmeur à l'esprit ordonné écrit des instructions limpides. un programmeur à l'esprit embrouillé fait d'obscurs commentaires.*

## 6.3 Tu ne coderas point avant d'avoir conçu

Une fois que nous avons admis que notre connaissance des objets que nous manipulons doit s'exprimer dans le langage, nous pouvons nous poser la question : ne peut-on aller plus loin ? Ne peut-on également utiliser Ada pour exprimer nos décisions de conception, avant même d'atteindre la phase de codage ? Nous touchons là à un principe sacro-saint : on ne doit pas commencer à écrire du code tant que les phases de conception ne sont pas totalement terminées. Mais quelle est l'origine de ce principe ? Revenons d'abord sur la démarche de conception elle-même.

Au départ, on trouve un problème particulier à résoudre, dans le cadre du projet. Mais ce problème n'est souvent qu'un cas particulier d'un besoin plus général, qu'il nous faut identifier. Enfin, nous devons trouver une implémentation répondant au besoin.

A l'aube de l'humanité (c'est-à-dire avant l'apparition d'Ada), le langage de programmation n'était considéré que comme une suite d'instructions données à une machine. Autrement dit, il ne permettait d'exprimer que la troisième étape. Le programmeur en revanche n'était préoccupé que de la première : trouver une solution à son problème. En codant directement ce qui lui paraissait une solution, il court-circuitait totalement la deuxième partie : la vraie réflexion sur la *nature* du problème.

Prenons par exemple un dispositif destiné à faciliter l'accordage d'un piano. Après avoir mesuré la fréquence d'une note jouée, il faut rechercher la note théorique la plus proche, et indiquer l'écart entre la fréquence mesurée et la fréquence théorique de la note (*besoin particulier*). Ce besoin n'est en fait qu'un cas particulier du *problème général* de situer une valeur par rapport à une fonction tabulée. La solution à un problème d'informatique n'est jamais unique, par conséquent il existera de nombreuses façons différentes d'implémenter cette notion abstraite. Dans notre exemple, il pourra exister différentes formes d'organisations de la table des valeurs et plusieurs algorithmes de recherche possibles (les *implémentations*). Ici, le programmeur (qui travaillait en Pascal, mais venait de FORTRAN) n'avait songé à utiliser que l'outil normal à tout faire de FORTRAN : la boucle `DO` (ou **`for`** en Pascal). Il avait donc codé un algorithme de parcours séquentiel de la table, et trouvait son code insuffisamment performant (il fallait reconnaître les notes en temps réel). La table des fréquences des notes étant naturellement triée, il fallait bien sûr utiliser une recherche dichotomique. L'algorithme trouvait ainsi la bonne note (parmi 128) en 7 comparaisons (au pire) au lieu de 64 (en moyenne) avec la recherche séquentielle. Soit une accélération de presque un facteur 10...[1]

Le passage trop rapide du problème particulier au codage risque donc de faire passer à côté d'autres solutions plus performantes, ou bien de faire adopter une solution à court terme impossible à faire évoluer. La règle habituelle, qui interdit au programmeur de coder avant d'avoir formalisé son problème, sert à l'obliger à passer par cette deuxième étape de reformulation. Le point fondamental n'est donc pas de concevoir avant de coder : c'est de formaliser le problème avant d'adopter une implémentation particulière.

Faute de passer par cette étape, il n'est plus possible de faire la différence entre un concept et son implémentation. Et comme chaque couche de logiciel s'appuie sur d'autres couches plus profondes, des dépendances *transitives* aux implémentations vont s'établir à travers tous les niveaux, conduisant à de véritables *fuites d'abstraction*. Un exemple typique de ceci est un problème auquel se trouvent confrontés les développeurs utilisant DBase III sur PC. La séquence de code pour faire passer une imprimante en mode double largeur se termine par l'envoi du caractère NUL. Or, le mode d'emploi de DBase III spécifie bien «qu'il n'est pas possible d'envoyer un caractère NUL sur l'imprimante». La raison en est évidente à quiconque a pratiqué les langages de programmation : DBase est écrit en C, et c'est l'habitude en C d'utiliser le caractère NUL comme terminateur de chaîne, ce qui interdit de le faire figurer *dans* une chaîne. Il n'empêche qu'il n'est pas possible d'écrire en double largeur dans une application DBase *à cause d'un choix de représentation d'une structure de donnée particulière dans le langage qui a servi à écrire le compilateur DBase !* L'origine du problème se trouve trois niveaux d'abstraction plus bas, mais C est typiquement un langage qui «fuit» : le besoin abstrait est celui d'une chaîne de caractères, mais le comportement des abstractions est entièrement gouverné par le choix de représentation sous-jacent : impossible d'ignorer qu'une chaîne de caractères est en fait un pointeur sur un octet, ni qu'un tableau n'est en fait que l'adresse de son premier élément[2].

[1] Pour que l'histoire soit complète, notons que ce programmeur était venu nous voir à l'origine pour savoir si un **case** était plus performant qu'un **if**. Bien sûr, on ne peut espérer mieux à ce niveau qu'un gain de quelques pour-cents de performance dans le meilleur des cas. Bel exemple de ce que nous avons dit à propos de la recherche d'efficacité...

[2] Signalons au passage à ceux qui ne voient pas le problème ici qu'en Ada, la valeur interne d'une variable tableau n'est généralement *pas* l'adresse de son premier élément.

## 6.4 Le parcours horizontal du V de développement

Les différentes méthodes de conception recouvrent différentes façons de formaliser les problèmes. Or Ada nous offre, au moyen des spécifications indépendantes des implémentations, la possibilité d'exprimer la formulation abstraite *sous une forme compilable*, c'est-à-dire vérifiable (dans une certaine mesure) au moyen d'un outil automatique, le compilateur. Cette possibilité a un effet considérable sur tout le processus de développement.

On représente habituellement les étapes de la conception au moyen du «V» de développement, qui représente le modèle dit de la «chute d'eau» (Fig. 9).

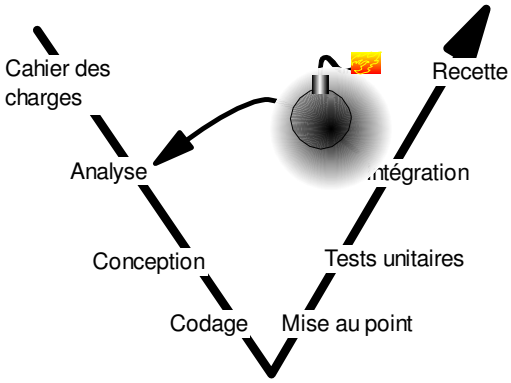


**Figure 9 :** Le modèle de la chute d'eau

On part du cahier des charges, dont on tire l'analyse générale, puis la conception détaillée et le codage. On ne passe à l'étape suivante qu'après avoir validé l'étape précédente. Une fois le codage effectué, on va vérifier en remontant en sens inverse : au codage correspond la mise au point, à la conception détaillée correspondent les test unitaires, à la conception générale correspond l'intégration, et enfin au cahier des charges correspond la recette.

Le principal problème de ce modèle est qu'il est extrêmement sensible aux erreurs. Si l'on a commis une faute au niveau de la conception générale, elle ne sera trouvée qu'au moment de l'intégration, et sa correction nécessitera une nouvelle itération à travers tout le cycle de développement (d'où la bombe !). On a tenté de pallier cette difficulté en multipliant les niveaux de contrôle, mais il semble utopique d'espérer éliminer totalement la possibilité d'erreurs.

En revanche, si nous considérons les différentes étapes de conception comme des vues de plus en plus concrètes de l'expression d'un problème, nous pouvons à chaque étape formaliser la conception au moyen de spécifications Ada. Ces spécifications sont compilées, et donc vérifiées. On n'aborde les étapes ultérieures qu'après avoir vérifié que les différents éléments de plus haut niveau peuvent bien s'assembler de la façon souhaitée. On va donc en quelque sorte effectuer l'intégration avant même d'aborder la conception détaillée ! Ceci aboutit à ce que nous appelons le *parcours horizontal du «V» de développement*, représenté sur la figure 10. Nous verrons, dans la troisième partie de cet ouvrage, comment cette façon de faire peut être systématisée pour développer des applications par *maquettage* progressif.

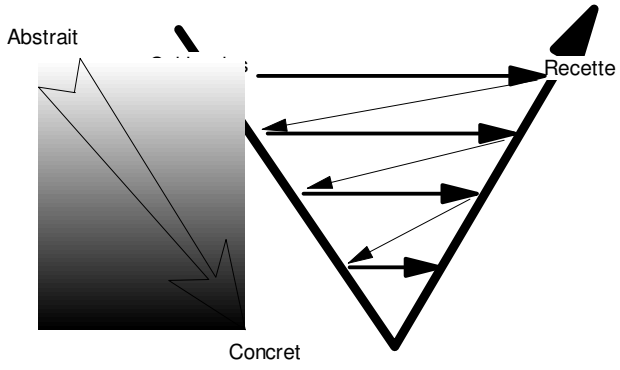


**Figure 10 :** Le parcours horizontal du «V» de développement

Ces considérations peuvent paraître théoriques : il n'en est rien, et l'une des remarques qui revient le plus souvent lorsqu'on interroge des responsables de projets qui sont récemment passés à Ada est la mention d'un véritable effondrement des temps d'intégration. Ce n'est pas étonnant : le langage est un gardien tellement vigilant qu'il empêche la construction de pièces qui ne pourraient s'assembler par la suite.

## 6.5 La nouvelle documentation de maintenance

A partir du moment où nous disposons d'un langage suffisamment puissant pour permettre l'expression directe et vérifiée de nos conceptions, la documentation traditionnelle est-elle appelée à disparaître ? Certainement pas, mais son rôle va être modifié.

Tout d'abord, la documentation de *conception* restera, car il importe de garder l'historique des choix principaux qui ont conduit à la structure actuelle du projet. La documentation réellement mise en cause est la documentation de codage. Elle ne doit plus être une description des algorithmes effectivement choisis, puisque ceux-ci sont mieux décrits par le code, ou alors se situer à un niveau nettement supérieur (description au moyen de langages formels) : elle doit essentiellement servir à tracer les *raisons des choix* qui ont abouti à la sélection de telle ou telle politique d'implémentation.

On ne le répétera jamais assez : la solution à un problème d'informatique n'est jamais unique. Une excellent habitude consiste d'ailleurs, lorsqu'un choix d'implémentation paraît évident, à systématiquement chercher une deuxième possibilité, quitte à la réfuter immédiatement : on est ainsi assuré d'avoir fait un choix délibéré, et non un choix *implicite*, c'est-à-dire provenant simplement d'un manque de réflexion sur le problème.

Il arrive également fréquemment que l'on s'aperçoive après coup qu'un choix d'implémentation n'était pas le bon, et que l'on doive revenir sur une décision antérieure, parce que l'on n'avait pas envisagé certaines difficultés. Le rôle de la documentation de bas niveau va être essentiellement de garder la trace de ces choix, des raisons qui les ont motivés, et en cas de retour arrière, des raisons du choix initial, des difficultés rencontrées, et des motivations de la remise en cause. Ceci apportera au lecteur ultérieur une meilleure compréhension du problème, et surtout lui évitera de refaire lors d'une opération de maintenance les erreurs commises au moment du développement. Combien de fois n'avons-nous pas eu l'impression que le concepteur initial était passé à côté d'une solution «évidente», alors qu'en fait celle-ci ne pouvait fonctionner pour des raisons qui n'apparaissaient que bien plus tard !

En résumé, on peut dire que le rôle de la documentation de maintenance n'est plus de décrire la solution adoptée, mais les raisons et les différents compromis qui ont conduit à préférer cette solution à d'autres. En particulier, le développeur consciencieux mentionnera les évolutions possibles qui peuvent amener à reconsidérer les choix : par exemple, une solution a pu être préférée à une autre pour des raisons d'efficacité, mais l'arrivée d'un nouveau matériel plus performant peut remettre en cause ce choix.

## 6.6 Exercices

1. Reprendre la documentation de maintenance d'un logiciel réel et la critiquer en fonction des critères de ce chapitre. Décrit-elle les principes ou les détails de la solution ? Un nouveau venu y trouverait-il rapidement les éléments lui permettant de comprendre la structure du projet ? Les informations n'auraient-elles pas pu s'exprimer dans le langage ? Etc.
2. Expliquer pourquoi il serait bon d'écrire le mode d'emploi «utilisateur» d'un programme avant d'écrire le programme.

# Deuxième partie

# Méthodes avec Ada

Nous avons vu dans la première partie le rôle important joué par les langages du point de vue méthodologique. Dans cette deuxième partie, nous passons en revue les différentes grandes classes de méthodes de conception, en montrant comment Ada leur fournit un soutien *actif*, et ceci quelle que soit la méthode.

Car là est bien l'enjeu et l'ambition d'Ada : être le langage qui soutient *toutes les méthodes*.

# 7
# Les méthodes structurées

## 7.1 Principes de la méthode

La programmation structurée fut historiquement la première formalisation de l'analyse par décomposition descendante. Le principe de cette méthode est simple : on considère qu'un programme est constitué d'une suite d'*actions* élémentaires. On décompose donc au plus haut niveau le problème à résoudre en actions individuelles que l'on suppose résolues ; on analyse ensuite chacune de ces actions en la décomposant sous forme d'actions de plus bas niveau, et ainsi de suite jusqu'à obtenir des actions considérées comme triviales et susceptibles de s'écrire directement dans le langage de programmation. La programmation structurée peut donc être définie comme une méthode dont le critère de décomposition horizontale est l'*enchaînement des actions*, et le critère de décomposition verticale la *généralité des actions*.

Le langage dont la philosophie se rapproche le plus de la programmation structurée est Pascal. On peut voir l'arbre de décomposition de la programmation structurée comme l'arbre d'imbrication des procédures d'un programme Pascal : un programme principal, qui se décompose en procédures de premier niveau, elles-mêmes décomposées en procédures de deuxième niveau, etc.

Ada, possédant un sous-ensemble équivalent à Pascal, permet bien entendu cette approche des problèmes. Il lui apporte cependant un perfectionnement important : il est possible d'imbriquer logiquement des modules, tout en conservant la possibilité de les compiler séparément. Le Pascal standard ne possède pas de mécanisme de compilation séparée. Les Pascal «étendus» (tels que Turbo Pascal par exemple) possèdent de tels mécanismes, mais à condition de ne pas conserver la notion d'imbrication. Le mécanisme des corps séparés permet, comme nous allons le voir dans l'exemple suivant, de compiler séparément des modules tout en conservant l'imbrication logique.

## 7.2 Exemple en programmation structurée

Supposons que nous voulions imprimer un triangle de Pascal (le mathématicien, pas le langage !). Il s'agit d'un triangle de nombres, se présentant comme à la figure 11.

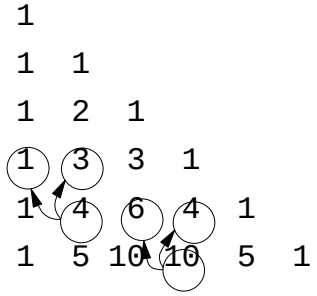


**Figure 11 :** Triangle de Pascal

Chaque élément d'une ligne s'obtient en faisant la somme de l'élément situé au-dessus de lui et de l'élément situé au-dessus et à gauche, le dernier élément à droite (qui n'a pas d'élément supérieur) étant toujours un 1. Pour résoudre le problème global (imprimer le triangle), il suffit de savoir résoudre deux sous-problèmes : calculer une ligne, et l'imprimer. Nous avons également besoin d'une structure de données pour communiquer entre ces deux étapes : la `Ligne`. Le programme principal peut s'écrire :

```
procedure Triangle_Pascal is
   Max : constant := 10;
   Ligne: array(1..Max) of INTEGER;
   procedure Calculer_Ligne (Numéro : INTEGER) is separate;
   procedure Imprimer_Ligne (Numéro : INTEGER) is separate;
begin
   for I in 1..Max loop
      Calculer_Ligne (Numéro => I);
      Imprimer_Ligne (Numéro => I);
   end loop;
end Triangle_Pascal;
```

Nous pouvons compiler ce programme pour vérifier qu'il ne contient pas d'incohérence à ce niveau de décomposition, alors que nous n'avons encore pris aucune décision de conception en ce qui concerne les niveaux inférieurs.

> La clause **is separate** permet de compiler séparément un corps de sous-programme, tout en gardant toutes les propriétés (notamment la visibilité et le contrôle des types) qu'il aurait s'il se trouvait effectivement à la place de la clause.

Ensuite, nous devons descendre d'un niveau en nous préoccupant de savoir *comment* nous allons réaliser les procédures `Calculer_ligne` et `Imprimer_ligne`. Comme les lignes sont calculées dans l'ordre, lorsque `Calculer_ligne` est appelé la variable `Ligne` contient la ligne précédente (d'ordre `Numéro-1`). Nous pouvons utiliser cette information pour calculer la ligne d'ordre `N`, à condition de calculer *de droite à gauche*, afin de ne pas «écraser» des valeurs dont nous avons encore besoin. Le premier élément valant toujours 1, il est inutile de le recalculer, et nous savons que le dernier vaut également toujours 1. Nous pouvons donc écrire :

```
separate (Triangle_Pascal)
procedure Calculer_Ligne(Numéro : Integer) is
begin
   Ligne (Numéro) := 1;
   for I in reverse 2..Numéro-1 loop
      Ligne (I) := Ligne (I) + Ligne(I-1);
   end loop;
end Calculer_ligne;
```

Au niveau d'abstraction supérieur, nous n'avions pas prévu de traitement spécial pour les premières lignes ; nous devons donc vérifier ce qui se passe, pour éventuellement faire un cas spécial. Si `Numéro` vaut 1, la procédure met la valeur 1 dans `Ligne(1)`, et la boucle n'est pas effectuée (`Numéro-1` vaut 0, qui est plus petit que 2). Le traitement est donc correct. Si `Numéro` vaut 2, le premier élément est intouché, on met 1 dans le deuxième élément, et la boucle n'est pas effectuée : le traitement est encore correct. Enfin, à partir de 3, l'algorithme normal s'applique. Il n'y a donc pas à faire de cas particulier, ce qui n'était pas évident au départ.

Pour imprimer, nous pouvons représenter le comportement souhaité de la façon suivante : écrire la ligne d'ordre `I` revient à écrire les `I` premiers éléments, suivis d'un retour à la ligne. Noter qu'à ce niveau, nous ne savons pas *comment* écrire un élément : nous reportons ce problème à l'étape suivante. Nous pouvons donc écrire :

```
with Ada.Text_IO;
separate (Triangle_Pascal)
procedure Imprimer_Ligne (Numéro : Integer) is
   procedure Imprimer (Elem : Integer) is separate;
   use Ada.Text_IO;
begin
   for I in 1 .. Numéro loop
      Imprimer (Ligne (I));
   end loop;
   New_Line;
end Imprimer_Ligne;
```

Il ne nous reste plus qu'à analyser comment imprimer un élément simple. Ada fournit des entrées-sorties prédéfinies sur le type `Integer` dans le paquetage `Ada.Integer_Text_IO`.

> Ce paquetage n'est en fait qu'une pré-instanciation du paquetage générique `Text_IO.Integer_IO`, qui permet d'avoir des entrées-sorties sur n'importe quel type entier. Il n'était pas obligatoire en Ada 83.

Nous allons donc utiliser ce composant :

```
with Ada.Integer_Text_IO;
separate (Triangle_Pascal.Imprimer_Ligne)
procedure Imprimer (Elem : Integer) is
   use Ada.Integer_Text_IO;
begin
   Put (Elem, Width =>2);
end Imprimer;
```

Bien sûr, cet exemple est un peu exagérément détaillé, car son but est de montrer la démarche : *pour imprimer un triangle de Pascal*, on suppose que l'on sait calculer et imprimer une ligne. *Pour imprimer une ligne*, on suppose que l'on sait imprimer un élément. Chaque étape s'appuie sur des problèmes plus spécifiques, que l'on suppose résolus, puis que l'on implémente. Inversement, chaque étape de la résolution s'appuie sur une vue de plus en plus précise, mais de plus en plus étroite du problème : `Imprimer_ligne` ne traite que de l'impression, sans se préoccuper de la façon dont la ligne a été calculée. `Imprimer` s'occupe uniquement de l'impression d'un élément, sans savoir où ni pourquoi on le lui demande. Enfin, chaque niveau s'exécute séquentiellement dans le temps, et sait dans quel ordre il est appelé : c'est ce qui permet à `Calculer_ligne` d'utiliser le résultat du calcul précédent.

## 7.3 Critique de la méthode

Lors de son émergence, la programmation structurée représentait un grand progrès par rapport aux méthodes, ou plutôt à l'absence de méthodes, existantes. Cependant, avec les progrès de l'informatique et l'extension du domaine des problèmes susceptibles d'être résolus de façon informatique, elle bute sur un certain nombre de difficultés que nous allons résumer maintenant.

Cette méthode conduit à une topologie de programme purement arborescente, comme illustrée par la figure 12. Chaque module est fait spécifiquement en fonction des exigences du niveau supérieur : ceci conduit à une architecture de programme très monolithique, semblable à un puzzle où chaque morceau est destiné à entrer à un endroit précis pour lequel il a été conçu. Il y a donc un couplage *spatial* entre les modules : comme un module «voit» tous les éléments qui lui sont supérieurs, il n'est pas possible de recopier un module dans une autre branche du programme. De plus, le développeur sait *quand*, dans le déroulement du programme, son module sera appelé : il peut donc faire des suppositions sur l'état des variables, ce que fera le prochain module appelé, etc.[1] Ceci entraîne également un couplage *temporel* des modules : le même sous-programme, appelé depuis un endroit qui n'était pas celui initialement prévu, risque de ne plus fonctionner normalement.

[1] Qui n'a jamais entendu des remarques du genre : «Pas la peine de remettre à jour la variable X, elle est écrasée par le prochain module» ?

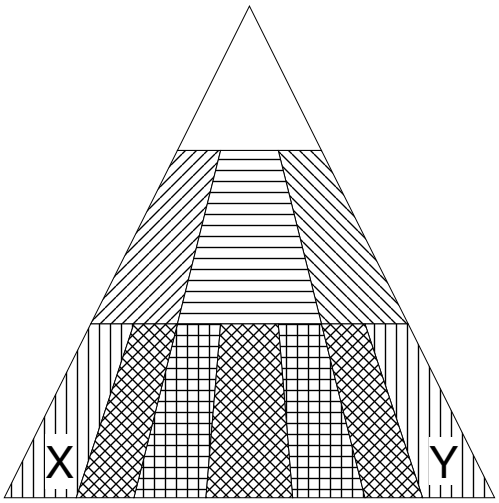


**Figure 12 :** Topologie de programme en programmation structurée

Une autre conséquence de l'aspect monolithique des programmes ainsi développés est qu'il est très difficile d'autoriser les compilations séparées. Pascal en standard ne les permet pas. Si de nombreuses extensions de Pascal permettent de le faire, c'est toujours en répétant dans chaque module les déclarations globales qu'il est censé voir ; une modification d'un de ces objets globaux doit être répercutée dans tous les modules qui l'utilisent, et le langage ne garantit absolument pas la cohérence des différentes déclarations[1]. Tout ceci entraîne que les modules développés pour une application peuvent difficilement être réutilisés dans un contexte différent. Même à l'intérieur d'une même application, on trouve souvent des fonctionnalités très voisines, mais légèrement différentes, ce qui conduit à une duplication de codes quasi identiques. Ceci est d'ailleurs une conséquence inévitable de la hiérarchisation. C'est ce que nous allons montrer maintenant.

Il arrive souvent qu'une fonctionnalité identique se retrouve à différents endroits d'un même programme ; on peut imaginer par exemple que les modules marqués X et Y sur la figure 12 aient tous deux besoin d'effectuer des lectures sur le terminal, et que l'on souhaite passer par une procédure permettant d'éditer la ligne par exemple. Dans la décomposition strictement arborescente de la programmation structurée, ceci s'exprimerait par le fait que deux modules, *a priori* totalement indépendants, expriment tous deux dans le cours de leurs actions la sous-action «lire une ligne». En suivant strictement la méthode, il faudrait écrire le même sous-programme deux fois dans deux modules distincts. C'est évidemment peu souhaitable ; ce n'est pas tant la duplication de l'écriture qui est gênante (un bon éditeur de texte le fait en quelques touches) ni même l'espace mémoire gaspillé par la duplication du code (les machines ont des espaces mémoire importants de nos jours) : le problème réel vient de la maintenance. Si une modification doit être faite dans un de ces modules dupliqués, il faut la répercuter dans toutes les copies, et le risque est grand d'en oublier. On peut ainsi voir des erreurs corrigées à un endroit réapparaître soudainement à un autre ; on risque également des problèmes d'incohérence : imaginez un programme où le caractère d'effacement serait Del par endroits et Backspace à d'autres !

La solution généralement adoptée consiste à remonter la fonctionnalité dans l'arbre de décomposition jusqu'à un endroit où elle sera visible de tous les modules qui l'utilisent, c'est-à-dire presque toujours jusqu'à la racine. *Le module se situe alors à un niveau global qui ne correspond plus à la décomposition logique*. Le même phénomène se produit pour les structures de données, lorsque deux parties du programme doivent à travailler sur une même variable. Les règles de visibilité exigent alors que la variable soit «remontée» dans l'arbre au moins jusqu'au nœud commun aux deux branches concernées.

[1] Le même reproche peut d'ailleurs être adressé à C, à FORTRAN (avec les COMMON), etc.

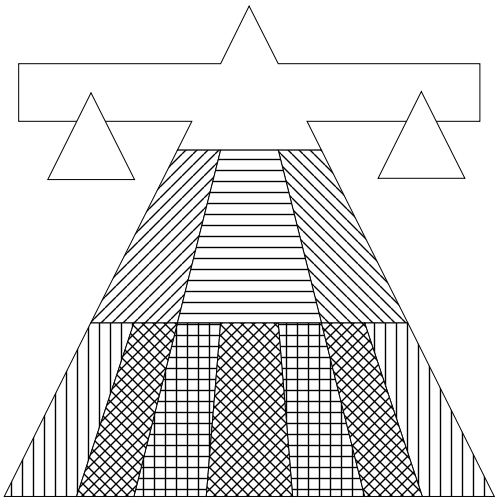

**Figure 13 :** Dégénérescence de la topologie de programme en programmation structurée

On s'aperçoit donc qu'en pratique, la belle arborescence théorique décrite par la méthode dégénère pour aboutir à une topologie illustrée par la figure 13. On a alors des arborescences partielles, avec un regroupement important de fonctions et de structures de données au niveau de la racine. Remarquons qu'il s'agit d'une première entorse aux principes de la programmation structurée : des entités sont déclarées à un niveau où elles ne correspondent pas à une action élémentaire du niveau immédiatement supérieur.

Ce regroupement a une conséquence extrêmement néfaste : les éléments ainsi propulsés au niveau global sont alors visibles non seulement depuis les modules concernés, mais également depuis l'ensemble du programme. Si par exemple une variable ne doit être (logiquement) modifiée qu'en passant par l'intermédiaire d'une procédure de contrôle, on n'a plus aucune garantie qu'un programmeur «astucieux» n'a pas trouvé plus commode d'accéder à la variable directement. Il devient donc extrêmement difficile de contrôler quelle partie du programme utilise quelle autre partie, puisque toutes sont visibles entre elles. On a bien décomposé le programme en unités de tailles suffisamment petites pour être gérables, mais on a établi des liens de dépendance qui croissent rapidement, jusqu'à dépasser ce que l'on peut gérer ; on ne sait plus alors très bien qui modifie quoi ou qui appelle qui dans le programme.

Notons pour terminer que la définition même de la programmation structurée suppose l'ordonnancement séquentiel des actions. Il n'est donc pas possible de l'utiliser pour définir des systèmes parallèles. On peut pallier cet inconvénient en définissant un système parallèle comme un ensemble de *processus*, individuellement séquentiels, mais s'exécutant en parallèle. La programmation structurée peut alors s'appliquer à l'analyse de chacun des processus, mais bien sûr l'analyse du comportement global requiert une méthode appropriée au parallélisme.

Ce problème de l'analyse des systèmes parallèles à été longtemps ignoré, ou tout au moins confiné à quelques domaines particuliers, car les langages courants n'offraient pas de possibilité de programmation parallèle, ce qui n'encourageait pas les développeurs de systèmes d'exploitation à développer le parallélisme, ce qui n'encourageait pas le développement de langages parallèles, etc. La situation change aujourd'hui, car la demande de toujours plus de puissance de calcul ne pourra être satisfaite qu'avec des machines parallèles, ce qui a amené l'introduction de processus légers (*threads*) au niveau des systèmes d'exploitation... ainsi bien entendu que l'apparition d'un langage de programmation industriel complet offrant, au même titre que ses autres fonctionnalités, un modèle simple de programmation parallèle[1].

[1] Ada, pour ceux qui ne l'auraient pas reconnu !

## 7.4 Ada et la programmation structurée

Pascal avait été conçu spécifiquement pour mettre en œuvre les principes de la programmation structurée. Ada a poursuivi dans cette voie, tout en ajoutant les éléments nécessaires à un langage d'envergure industrielle.

### *7.4 .1 Utilisation des sous-programmes*

Ainsi que nous l'avons vu, Ada permet comme Pascal de traduire la hiérarchie des actions résultant de la programmation structurée sous forme d'imbrication de procédures, mais offre de plus la possibilité de compiler séparément les corps. On peut certes s'affranchir de la hiérarchie stricte en regroupant certains sous-programmes en paquetages ; mais la hiérarchisation est toujours possible.

On a prétendu [Cla80] que cette possibilité d'imbrication était inutile, voir même nocive. Les langages comme C/C++ n'autorisent d'ailleurs pas l'imbrication de sous-programmes. Il est en effet toujours possible de mettre deux sous-programmes «à côté» l'un de l'autre, au lieu de les imbriquer, avec sensiblement les mêmes fonctionnalités :

```
-- Sous-programmes
-- imbriqués

procedure A is
   procedure B is
      ...
   end B;
begin
   B;
end A;
```

```
-- Sous-programmes
-- non imbriqués

procedure B is
    ...
end B;
procedure A is
begin
      B;
end A;
```

La différence vient encore de la maintenance : si un sous-programme semble appelé à tort et qu'il est déclaré au niveau le plus externe, on doit jeter la suspicion sur l'ensemble du programme ; si au contraire il est déclaré à l'intérieur d'un autre sous-programme, l'appel incorrect a forcément lieu depuis l'intérieur de cet autre sous-programme, réduisant considérablement le champ d'investigation nécessaire pour retrouver l'erreur.

Ada permet donc de traduire la structure de la conception directement dans la structure du programme, tout en conservant la possibilité de compilation séparée. Une difficulté est que si l'on modifie une couche haute, il faudra recompiler tous les éléments (séparés) de plus bas niveau ; mais ceci est un problème inhérent à la *méthode*, où une modification des couches hautes a des conséquences sur toutes les couches inférieures ; le langage ne fait que répercuter les propriétés de la méthode.

### *7.4 .2 Utilisation des paquetages*

En programmation structurée, les paquetages servent essentiellement à regrouper soit des structures de données, soit des structures de programmes. On trouvera donc, selon la classification établie par Booch, essentiellement des «collections de données» et des «collections de sous-programmes». Les principes de la programmation structurée impliquant la visibilité des données, on trouvera rarement des parties privées dans les paquetages utilisés avec cette méthode.

#### a) Collection de données

Une collection de données est un simple ensemble de données globales utilisées par tout le système, à l'exclusion de toute opération. En Ada, une collection de données se présentera sous la forme d'une spécification de paquetage ne comportant que des déclarations de types, de constantes, de variables ou d'exceptions. Il n'y aura généralement pas de corps de paquetage, sauf peut-être pour permettre l'initialisation des variables. Les collections de données globales existent depuis

longtemps dans les langages de programmation (`COMMON` FORTRAN). Ada offre cependant un niveau de sécurité supplémentaire, car le compilateur gère tous les contrôles de types même entre unités compilées séparément ; il n'est donc plus possible de «tricher» sur les types par le biais de ce type d'unité.

Dans les méthodes de programmation structurée, on trouve souvent la notion de dictionnaire de données globales, réalisé au moyen de paquetages de type «collection de données». Bien qu'on considère généralement qu'il n'existe qu'un seul dictionnaire de données, on peut l'implémenter au moyen de plusieurs paquetages, chaque paquetage regroupant des données ayant un lien entre elles. On diminue ainsi les recompilations en cas de modification du dictionnaire de données, mais surtout le graphe de dépendances des **`with`** fournit une indication précise sur les données utilisées par chaque module.

### b) Collection de sous-programmes

A l'opposé de la collection de données, on trouve la collection de sous-programmes. Ceci correspond à la notion habituelle de *bibliothèque* : bibliothèque mathématique, graphique, scientifique... Une telle collection s'exprime en Ada sous la forme d'un paquetage dont la spécification ne comporte que des sous-programmes (procédures et fonctions). Le point important est que le paquetage ne comporte aucun état rémanent entre deux appels de ses sous-programmes. On s'interdit donc toute modification de variable globale par un sous-programme, et l'on garantit que les valeurs renvoyées ne dépendent que des paramètres et non de l'ordre d'appel des sous-programmes. Les sous-programmes sont donc indépendants les uns des autres, et sans mémoire.

Le pragma `Pure` permet de faire vérifier par le compilateur que ces conditions sont bien respectées.

## *7.4 .3 Utilisation des exceptions*

Les canons de la programmation structurée ne connaissent pas la notion de déroutement ; l'utilisation des exceptions sera donc généralement limitée si l'on applique strictement la méthode. D'ailleurs, les règles de programmation associées aux méthodologies en programmation structurée (surtout pour les systèmes à haute sécurité) déconseillent ou interdisent l'utilisation des exceptions. L'avantage de la programmation structurée est qu'elle permet de suivre directement l'exécution du programme, propriété qui n'est plus vérifiée en présence d'exceptions. Malgré tout, il faut bien définir une politique d'erreur (nous en discuterons dans la quatrième partie) ; mais on utilisera plutôt une technique par codes de retour.

On ne peut cependant ignorer totalement les exceptions : d'une part, les composants logiciels n'ont pas d'autre solution que de les utiliser en cas d'impossibilité de rendre le service demandé ; d'autre part, une anomalie du programme peut conduire à la levée d'une exception prédéfinie. On évitera donc en général de lever des exceptions, mais on devra prévoir *quand même* des traite-exceptions de sécurité. En cas de levée d'exception imprévue, on appellera la procédure de gestion normale des situations erronées.

## *7.4 .4 Utilisation des génériques*

Nous avons vu que la réutilisation de code en programmation structurée impliquait soit de le dupliquer, ce qui posait des problèmes de maintenance, soit de le remonter à un niveau global, ce qui violait la décomposition logique. Les génériques d'Ada offrent une solution élégante à ce problème. Supposons par exemple que nous ayons besoin de la procédure `Lire_Ligne` à plusieurs endroits du programme ; cette procédure a pour mission de lire une ligne au terminal (en la

complétant avec des espaces, éventuellement dans une fenêtre, avec des fonctions d'édition...) et de la mettre dans une certaine variable de la procédure englobante. Initialement, le programmeur avait écrit :

```
procedure Utilisatrice is
   Mon_Tampon : String(1..80)
   procedure Lire_Ligne is
       Longueur : Natural;
   begin
       Get (Mon_Tampon, Longueur);
       Mon_Tampon(Longueur+1 .. Mon_Tampon'Last):=
                                                 others => ' ');
   end Lire_Ligne;
begin
   ...
end Utilisatrice;
```

Lorsqu'apparaît le besoin de réutiliser cette procédure dans un autre contexte, le programmeur ne duplique pas le code, mais le transforme en générique :

```
generic
   Tampon : in out String;
procedure Lire_Ligne_Générique is
   Longueur : Natural;
begin
   Get (Tampon, Longueur);
   Tampon(Longueur+1 .. Tampon'Last) := (others => ' ');
end Lire_Ligne_Générique;
```

Il peut ensuite, dans l'ancienne procédure ainsi que dans toutes celles qui en ont besoin, écrire :

```
procedure Utilisatrice is
   Mon_Tampon : String(1..80)
   procedure Lire_Ligne is
       new Lire_Ligne_Générique (Mon_Tampon);
begin
   ...
end Utilisatrice;
```

Ainsi, chaque procédure utilisatrice dispose de sa propre procédure locale, respectant la structure logique, mais le code n'est écrit qu'une fois, et toute modification de maintenance qui lui serait apportée sera automatiquement reportée dans toutes les instanciations. Les génériques permettent donc d'apporter à la programmation structurée une forme de réutilisation qui ne perturbe pas la structure totalement hiérarchique de l'application.

## *7.4 .5 Utilisation du parallélisme*

Nous avons vu que les méthodes en programmation structurée se prêtent mal au développement de systèmes parallèles. Il faut cependant parfois définir des activités parallèles dans des systèmes développés en programmation structurée. Dans ce cas, on décompose le système en tâches indépendantes dès les premières étapes de la conception, ce qui permet ensuite de poursuivre l'analyse de chacune d'elles en programmation structurée «normale».

On trouvera donc surtout des tâches Ada définies directement dans des paquetages de bibliothèque, se comportant essentiellement comme des programmes principaux multiples. En Ada 95, on pourra également utiliser le modèle d'exécution distribuée qui permet effectivement d'avoir plusieurs programmes principaux faiblement couplés.

Notons encore que pour un système développé en programmation structurée ayant besoin de parallélisme, il est particulièrement utile de disposer d'un langage offrant un mécanisme de tâches intrinsèque : le surcroît de complexité est alors négligeable. Autrement, il faut soit s'appuyer sur les primitives d'un exécutif temps réel particulier, mais on y perd la portabilité, soit écrire soi-même son propre système de gestion des tâches, mais l'effort supplémentaire est loin d'être négligeable, et il risque souvent d'imposer des contraintes à travers toutes les couches du logiciel. C'est ainsi que

les normes de programmation de systèmes graphiques tels que Windows ou X Window demandent à toutes les applications utilisatrices de se bloquer volontairement périodiquement, pour permettre au système de «reprendre la main».

## 7.5 Exercices

1. Analyser en programmation structurée un système d'affichage de la vitesse d'un véhicule. On suppose que l'on dispose d'un dispositif physique qui fournit une impulsion à chaque tour de roue.
2. Analyser en programmation structurée un programme qui compte le nombre de lettres, de mots et de phrases d'un fichier texte.
3. Expliquer pourquoi la programmation structurée est très utilisée pour les systèmes à fortes contraintes temps réel et/ou de haute sécurité.

# 8

# Les méthodes orientées objet

## 8.1 Principes des méthodes orientées objet

Les méthodes orientées objet sont nées du désir de remédier aux problèmes de la programmation structurée, sans en perdre bien entendu les avantages. Le principal mérite de la programmation structurée vient de ce qu'elle procure une méthode rationnelle, progressive, de décomposition des programmes. Celle-ci s'effectue de façon descendante, ce qui permet de mettre en place la structure générale sans se soucier des détails d'implémentation. Mais son problème essentiel vient du *couplage* des différents éléments de la décomposition entre eux. Ce que l'on cherche donc, c'est une méthode *descendante* (par raffinements successifs), mais produisant des modules à *faible couplage*. De plus, les modules doivent représenter chacun une et une seule entité logique, c'est-à-dire qu'ils doivent être à *forte cohésion* (il ne doit pas être possible de diviser un module en deux tout en conservant le faible couplage).

### *8.1 .1 Définition*

Pour remédier aux défauts de la programmation structurée, il convenait de remettre en cause son fondement même : la décomposition en actions. L'origine de cette réflexion se trouve dans un article de Parnas [Par71] intitulé «A propos des critères à utiliser pour décomposer un système en modules». Il s'agissait bien d'une remise en cause des critères de décomposition.

> *Nous proposons d'appeler «orientée objet» toute méthode dont le critère de décomposition horizontale est l'objet, en tant qu'abstraction d'un objet du monde réel.*

Mais comment définit-on ce qu'est une abstraction ? Prenons un exemple. Il est vraisemblable que vous qui lisez ces lignes possédez une voiture. Vous venez de lire la phrase précédente, et vous n'avez eu aucune peine à saisir sa signification. Pourtant possédez-vous vraiment une «voiture» ? Non, vous possédez un certain ensemble de tôles et de plastique que vous identifiez comme appartenant à une classe appelée «voiture» ; vous possédez donc un objet qui est une voiture, mais beaucoup d'autres objets ayant peu de rapport avec votre chère auto sont également des voitures. Le terme de voiture est une *abstraction*, qui permet de regrouper un certain nombre d'entités différentes.

Comment donc reconnaître si un objet donné est une voiture ou non ? Pour avoir droit à ce titre, il faut que l'objet en question ait quatre roues, des sièges et un moteur. Nous pouvons donc définir un certain nombre de valeurs qui sont caractéristiques de la classe voiture. Mais ceci ne suffit pas à la caractériser : un avion pourrait très bien posséder les caractéristiques précédentes. Il faut en plus spécifier que la voiture sert à transporter des gens sur la route. Nous devons donc définir, en plus des valeurs, un certain nombre d'actions que la voiture peut effectuer ou subir.

On peut donc définir une abstraction comme une *réduction d'une classe d'objets à un ensemble de valeurs et d'opérations communes à tous les objets de la classe, et permettant de la définir entièrement*.

Le point important de cette définition est la *réduction*. D'ailleurs, le terme *abstraire* ne signifie-t-il pas étymologiquement «retirer de» ? Par exemple, une propriété d'une voiture est la composition chimique de l'acier dans lequel a été fabriqué le vilebrequin. C'est une propriété *inutile* pour la vue de la voiture qu'a le conducteur moyen (mais pas pour l'ingénieur qui a conçu la voiture). L'abstraction va donc consister à extraire de l'énorme masse de propriétés caractérisant les objets un *petit nombre* d'entre elles qui vont être suffisantes pour l'idée que l'on se fait de l'objet à un moment donné. C'est donc essentiellement ce phénomène de réduction, de *projection* de l'objet réel dans l'espace de problème, qui va caractériser le phénomène d'abstraction.

Avec cette technique, la définition des objets s'appuie sur les propriétés naturelles de l'abstraction du monde réel. Un programme n'est donc plus une *suite d'instructions à faire exécuter par une machine*, mais une *description d'un certain modèle du monde réel*. Il s'ensuit une réorganisation complète du logiciel : si les instructions élémentaires se retrouvent quasi identiques quelle que soit la méthode de décomposition choisie, elles ne sont pas regroupées en modules de la même façon. Il n'est pas étonnant que l'ancêtre de tous les langages à objets, Simula, ait été un langage de simulation : dans quel autre contexte aurait-on plus besoin de représenter les entités du monde réel ?

De cette différence fondamentale découlent des propriétés partagées par toutes les approches objet :

L'*encapsulation*. L'objet n'est ni une structure de programme, ni une structure de données, mais regroupe en une même entité à la fois les attributs (données) et les opérations[1] (sous-programmes) associés. De plus, la structure interne (implémentation) doit pouvoir être cachée afin de garantir l'indépendance de la vue externe (abstraite) par rapport à la vue interne.

L'*indépendance temporelle*. Les objets étant définis par eux-mêmes, on peut définir le comportement d'un objet indépendamment du contexte dans lequel il est appelé.

- L'indépendance spatiale, ou *localisation*. *Tous* les aspects relatifs à une même entité sont physiquement dans le même module. Il est donc plus facile de faire des composants autonomes.

## *8.1 .2 Les objets informatiques*

Un objet informatique sera donc une entité de modularisation qui rassemblera les structures de données et les opérations pertinentes pour l'utilisation qui est faite de l'objet *du point de vue informatique*. Différentes façons d'utiliser la notion d'objet sont possibles, que nous allons maintenant passer en revue. Bien qu'il y ait toujours possibilité d'exceptions, l'expérience montre que les «bons objets» correspondent à ces catégories. On vérifiera donc lors de la conception que ce que l'on fait correspond aux critères ci-dessous ; sinon, il faut étudier s'il n'y a pas une faute de conception.

### a) Machine abstraite

On peut vouloir travailler simplement avec des représentations d'objets individuels du monde réel. On appelle de tels objets des *machines abstraites* (ou *machines à états abstraits*), car ce sont des entités autonomes possédant un état bien défini et des opérations modifiant cet état ; historiquement, on les a d'abord considérés comme des sortes d'*automates*, d'où leur nom. Les valeurs renvoyées par les opérations pourront dépendre de l'état courant, donc de l'historique des

[1] Appelées *méthodes* dans le jargon orienté objet.

appels, ce qui différencie les machines abstraites des collections de sous-programmes utilisées en programmation structurée.

On réalise en Ada une machine abstraite au moyen d'un paquetage ne comportant que des sous-programmes dans sa spécification (ainsi que, parfois, quelques types annexes nécessaires aux paramètres des sous-programmes) ; des variables cachées dans le corps de paquetage gardent trace de l'état courant. Par exemple, l'abstraction d'un moteur vue par le système de contrôle de régime pourrait être :

```
package Le_Moteur is
   procedure Allumer;
   procedure Couper;

   type Régime is range 0 .. 6_000;
   procedure Régler_Moteur (A : Régime);
   function Régime_Courant return Régime;
end Le_Moteur;
```

Le corps de ce paquetage possède des variables permanentes pour conserver la position de l'accélérateur, le réglage de l'avance à l'allumage, etc. Tous ces détails de *réalisation* ne sont pas visibles de l'extérieur.

En principe, chaque machine est unique ; on peut cependant utiliser des génériques pour obtenir plusieurs objets identiques.

### b) Type de donnée abstrait

Plutôt que des objets isolés, on veut souvent créer de nombreux objets similaires. Il faut dans ce cas définir un *type* qui servira de modèle (on parle parfois de *moule*) pour définir des objets identiques. On dit que ce type est *abstrait*, car seules les propriétés correspondant à la modélisation de l'objet du monde réel doivent être accessibles ; les contraintes de réalisation (notamment dues au langage de programmation) doivent rester cachées.

On réalise en Ada un type de donnée abstrait au moyen d'un paquetage dont la spécification comporte un seul type principal, normalement privé ou limité privé, et un ensemble d'opérations portant sur ce type. Il peut également y avoir, comme pour la machine à états abstraits, des types auxiliaires. Aucun état n'est conservé dans le paquetage ; toute l'information rémanente caractérisant l'état de l'objet doit être conservée dans les champs que comporte le type. Par exemple, nous pouvons représenter ainsi des compteurs simples :

```
package Gestion_Compteurs is
   type Compteur is private;

   type Comptable is range 0 .. 999_999;
   procedure Incrémenter    (Le_Compteur : Compteur);
   procedure Remise_A_Zéro  (Du_Compteur : Compteur);
   function Valeur_Courante (Du_Compteur : Compteur)
     return Comptable;

private
   type Compteur is
     record
       Valeur_Affichée : Comptable := 0;
     end record;
end Gestion_Compteurs;
```

Remarquer que ceci ne crée aucun objet : il faut déclarer des variables appartenant au type. Si l'on veut utiliser de tels compteurs dans une pompe à essence, on déclarera par exemple :

```
Compteur_Argent : Compteur;
Compteur_Volume : Compteur;
```

### c) Gestionnaires de données

On a souvent besoin d'objets bien connus en informatique, mais qui n'ont que peu de rapport avec des objets du monde réel : piles, listes, files, fichiers, etc. Le point commun à toutes ces structures est qu'elles n'ont aucune utilité par elles-mêmes : elles ne servent qu'à organiser, stocker ou gérer d'autres entités. Pour cette raison, nous préférons en faire une catégorie bien différenciée des autres objets, bien que leur réalisation puisse se faire soit sous forme de machine abstraite, soit sous forme de type de donnée abstrait. Nous appellerons ces entités des *gestionnaires de données*, le terme *structure de données* étant utilisé dans trop de contextes pour ne pas être ambigu. Ces gestionnaires sont normalement pourvus d'opérations de type «itérateur», comme expliqué au prochain paragraphe, et correspondent aux objets «à itérateur» de la classification de Booch [Boo87].

### d) Classification des opérations sur objets

L'interface entre les propriétés de l'objet et le monde extérieur passe normalement par des appels de sous-programmes. Quelques règles sont à respecter pour obtenir de «bons» objets.

Le premier principe est que chacun d'entre eux doit faire *une chose et une seule*. De plus, de même que les composants appartiennent normalement à l'une des catégories ci-dessus, les opérations (= sous-programmes fournis) appartiennent normalement à l'une des catégories ci-dessous ; toute déviation doit faire penser à l'éventualité d'une faute de conception.

*Sélecteurs*

Les sélecteurs sont des sous-programmes permettant d'interroger l'état ou les valeurs d'un objet. Ils seront souvent implémentés sous forme de fonctions. Ils ne modifient en aucun cas l'état de l'objet.

*Constructeurs*

Les constructeurs servent au contraire à modifier l'état d'un objet, soit pour l'initialiser, soit pour fournir une nouvelle valeur à une grandeur caractéristique, soit pour faire passer l'objet d'un état dans un autre (cas des automates).

*Itérateurs*

Les itérateurs sont des opérations un peu particulières, associées aux gestionnaires de données. Ils permettent de parcourir une structure en obtenant successivement tous ses éléments. On distingue les itérateurs *externes* et les itérateurs *internes*.

Un itérateur externe est un ensemble de sous-programmes, comportant :

  Une initialisation, donnant les caractéristiques des éléments à aller chercher dans la structure (éventuellement tous). Cette initialisation, selon les cas, se contente d'initialiser l'itérateur, ou fournit également la première valeur.

  Un moyen de rendre l'élément suivant de la structure actif. Il peut fournir directement l'élément suivant, ou simplement faire progresser l'itérateur. Il lève une exception s'il est appelé alors qu'il n'y a plus d'élément dans la structure.

  Un moyen de récupérer la valeur courante. Celui-ci peut être une fonctionnalité séparée, ou faire partie d'une des opérations précédentes.

- Un prédicat de fin de structure permettant d'arrêter les itérations.

L'itérateur externe le plus connu est celui traditionnellement utilisé pour accéder aux fichiers : `Open` constitue la fonction d'initialisation, `Get` le passage à l'élément suivant (avec fourniture de l'élément courant) et `End_Of_File` le prédicat de fin de structure. Des itérateurs similaires se retrouvent dans les bases de données.

Un itérateur interne vise à éviter d'extraire les valeurs de la structure, mais au contraire à effectuer un certain traitement *in situ*. On devra donc fournir une procédure qui sera appliquée

automatiquement à toutes les valeurs contenues dans le gestionnaire de données. Ceci s'exprime typiquement comme :

```
generic
   with procedure Traitement   (Elément : in out Donnée);
procedure Itérer;
```

Les itérateurs externes sont des briques de base simples, permettant de satisfaire à peu près tous les besoins... à condition d'écrire à chaque fois l'algorithme. Les itérateurs internes sont des entités de plus haut niveau, plus simples à utiliser (le parcours de la structure est automatique), mais la réalisation de certaines opérations telles que la jointure peut se révéler difficile.

Quelle que soit la forme d'itérateur utilisé, l'ordre dans lequel sont fournis les éléments variera selon la structure de donnée : ce sera l'ordre dans lequel les éléments ont été entrés (ou l'ordre inverse) pour une liste linéaire, un ordre croissant pour une table triée, ou même un ordre aléatoire dans le cas d'une table hash-codée. Cet ordre doit bien entendu être documenté dans la description de la structure de données.

### *8.1 .3 Objets actifs*

Le parallélisme d'Ada permet de définir des objets *actifs*, c'est-à-dire évoluant en parallèle. Ceci s'obtient en associant une tâche à une structure de donnée ; plusieurs façons de faire sont possibles, selon le degré de couplage souhaité entre la tâche et la structure de donnée.

Une première solution consiste à attacher manuellement une tâche à un objet. On peut pour cela utiliser un pointeur désignant l'objet comme discriminant de la tâche. Imaginons par exemple que nous voulions représenter un récipient qui fuit : toutes les 10 secondes, il perd un litre d'eau. Nous lui associons un «moniteur de fuite» qui diminue périodiquement la quantité restante :

```
package Compteur_fuyant is
   type Volume is delta 0.01 range 0.0 .. 100.0;
   type Compteur is
      record
        Contenu : Volume := 0.0;
      end record;

   task type Moniteur (Compteur_Associé: access Compteur);
end Compteur_Fuyant;

package body Compteur_fuyant is
   task body Moniteur is
   begin
      loop
        delay 60.0;
        if Compteur_Associé.Contenu > 1.0 then
          Compteur_Associé.Contenu :=
                              Compteur_associé.Contenu - 1.0;
        else
          Compteur_Associé.Contenu := 0.0;
        end if;
      end loop;
   end Moniteur;
end Compteur_Fuyant;

with Compteur_Fuyant; use Compteur_Fuyant;
procedure Test_Compteur is
    Mon_Compteur : aliased Compteur := (Contenu => 100.0);
    Ma_Tâche : Moniteur (Mon_Compteur'Access);
begin
    ...
end Test_Compteur;
```

Avec cette solution la tâche «moniteur» est relativement extérieure à l'objet «compteur» : rien ne nous empêche de ne *pas* associer de tâche à l'objet, et d'avoir un compteur qui ne fuit pas. Cela peut

être le comportement désiré. Si inversement nous voulons être sûrs que tous les compteurs fuient, il faut associer la tâche plus étroitement au compteur. C'est ce que permettent les *autopointeurs* :

```
package Compteur_fuyant is
   type Volume is delta 0.01 range 0.0 .. 100.0;

   type Compteur;
   task type Moniteur (Compteur_associé: access Compteur);

   type Compteur is limited
      record
         Contenu     : Volume := 0.0;
         Le_Moniteur : Moniteur (Compteur'ACCESS);
      end record;

end Compteur_Fuyant;
```

Remarquer que c'est le nom du *type* `Compteur` qui est utilisé ici comme préfixe de l'attribut `'Access`. Tout objet déclaré du type `Compteur` contiendra une tâche `Moniteur`, dont le discriminant désignera automatiquement l'objet `Compteur` dont il fait partie ; cette disposition est illustrée en figure 14. Ces autopointeurs ne sont autorisés que dans des types limités : en effet, l'affectation ne pourrait conserver la propriété de l'autopointeur de désigner sa propre structure englobante.

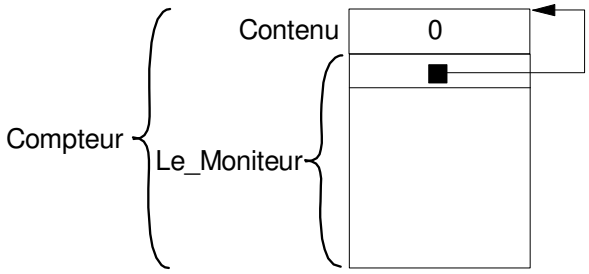


**Figure 14 :** Représentation d'un autopointeur

Le type `Compteur` utilise le mot clé **limited** dans sa déclaration d'article. Un tel type est appelé «intrinsèquement limité» : l'affectation est interdite même là où l'on a la visibilité sur le type complet, et le langage garantit que tous les passages se feront par adresse ; il n'y a donc *jamais* duplication.
Les autopointeurs ne sont pas limités aux tâches : on peut les utiliser pour lier n'importe quels types.

On peut réaliser ainsi des objets réellement actifs. Noter que l'on pourrait avoir plusieurs tâches associées à un même objet.

## *8.1 .4 Sémantique de valeur et sémantique de référence*

Il existe en fait deux sortes d'objets qu'il importe de bien différencier : ceux à sémantique de *valeur* et ceux à sémantique de *référence*[1]. Ils se distinguent par la signification de l'opération d'affectation.

Avec une sémantique de valeur, la signification de l'affectation :

```
X := Y;
```

est de transférer le *contenu* de X (sa *valeur*) dans Y. Une modification apportée par une partie du programme à X n'affecte pas les autres duplications (comme Y). La comparaison d'égalité n'implique que l'égalité des valeurs, que ces valeurs soient contenues dans un même objet ou non.

Nous parlerons de sémantique de référence chaque fois que la valeur abstraite est un identifiant : type accès, indice de tableau, ou toute autre forme de nommage dont les différentes duplications se réfèrent en fait au même *contenant*. La signification de l'affectation :

```
X := Y;
```

[1] Ce point a été identifié par M. Gauthier, dont on trouvera les travaux dans [Gau94].

devient : X et Y désignent le même objet. Toute modification apportée par un élément du programme au contenu de l'objet désigné par X sera donc visible (mais pas forcément immédiatement) à tous ceux qui utilisent des copies de la référence à l'objet (comme Y). La comparaison d'égalité entre deux variables aura la signification que les deux variables désignent le même objet (ce qui impliquera évidemment l'égalité des contenus). En revanche, deux objets peuvent être trouvés différents même si les contenus sont identiques.

Les types numériques offrent typiquement une sémantique de valeur, alors que les objets implémentés par un type accès offrent une sémantique de référence. Mais il ne faut pas restreindre cette différence à la simple alternative «pointeur ou pas pointeur», encore moins à la question de passage par adresse ou par valeur. Une chaîne de caractères représentant un nom de ficher a typiquement une sémantique de référence ! Il est également possible d'obtenir une sémantique de valeur sur un objet abstrait représenté par un type accès si l'on utilise un type limité muni d'une redéfinition de l'égalité et d'un sous-programme de recopie. Le choix entre type privé et type limité sera d'une grande importance pour renforcer le respect de la sémantique voulue.

La spécification de tout type de donnée abstrait doit explicitement mentionner s'il est à sémantique de valeur ou de référence. En aucun cas, un type ne doit fournir de sémantique mixte, comme dans l'exemple suivant :

```
type Info is ...
type Référence is access Info;

type Valeur is ... ;

type Sémantique_Mixte is    — A ne pas faire !!
   record
       Champ_1 : Référence;
       Champ_2 : Valeur;
   end record;
```

Un tel objet offrirait une sémantique de référence pour son `Champ_1` (c'est un pointeur) et une sémantique de valeur pour son `Champ_2`. Il n'est plus possible dans ce cas de décrire simplement la signification de l'affectation !

Même avec une sémantique de référence, on peut vouloir transférer le contenu d'un objet dans un autre. Un type de données à sémantique de référence doit alors disposer d'une fonction de recopie *différente de l'affectation*. Rappelons qu'Ada fournit désormais un puissant outil de contrôle du mécanisme d'affectation : un type dérivé du type prédéfini `Controlled` peut définir une opération, appelée `Adjust`, qui est appelée automatiquement après toute affectation à une variable de ce type. Il est ainsi possible de créer un type à sémantique de valeur, *même si l'implémentation utilise des pointeurs*, car l'opération `Adjust` peut forcer une duplication de la valeur de l'objet.

> Il existe également une opération `Initialize`, appelée lors de la création de l'objet, et une procédure `Finalize`, appelée lors de la destruction de l'objet ; cette dernière permet de récupérer l'espace mémoire alloué.

Remarquons que la distinction entre sémantique de valeur et sémantique de référence n'est importante que pour les types de données abstraits ; les machines abstraites étant par définition uniques, l'affectation n'est pas définie pour elles.

Bien que théoriquement rien ne l'impose, les gestionnaires de données doivent absolument être à *sémantique de référence.* Il serait évidemment désastreux du point de vue de l'efficacité, mais surtout au niveau du principe, d'avoir une sémantique de valeur (ce qui n'empêche pas, comme nous l'avons noté ci-dessus, de fournir une fonction de copie de la structure). En revanche, les *éléments gérés* par la structure de donnée pourront l'être soit sous forme de valeur (par exemple une liste de valeurs), soit sous forme de référence (par exemple une liste de pointeurs sur des objets). Cette distinction doit faire partie de la définition des propriétés de la structure de donnée.

A noter enfin que la *valeur* (interne) d'un type de donnée à sémantique de *référence* est un identifiant d'objet : par conséquent, une instanciation d'un gestionnaire de données gérant des valeurs avec un type à sémantique de référence sera en fait une structure de données manipulant des

références. L'instanciation d'une structure de données gérant des références avec un type à sémantique de référence sera une structure manipulant des «références doubles». Ceci peut arriver, mais doit être en général évité, car leur manipulation est délicate à maîtriser, et il est difficile d'en garantir une sémantique claire.

### *8.1 .5 Tout n'est pas objet* [1]

L'approche objet présente de nombreux avantages, mais cela ne signifie pas nécessairement qu'il faille *tout* modéliser en termes d'objets. Si l'entité du monde réel que l'on considère se résume à une suite d'actions, alors il est tout à fait souhaitable d'utiliser une approche structurée. Dans une conception orientée objet, ceci se produit dans deux cas particuliers importants : le premier et le dernier niveaux d'analyse.

Le premier niveau correspond à ce que l'on appelle généralement le programme principal. On aura défini les objets qui composent la représentation du domaine de problème, mais le programme principal est chargé d'animer, de faire vivre ces objets en appelant leurs fonctionnalités dans un certain ordre. Son analyse fait donc logiquement appel à la programmation structurée. De même, lorsqu'il s'agit de coder les corps des opérations des objets, il faudra bien utiliser une suite d'instructions du langage de programmation : là encore, une analyse en programmation structurée est appropriée.

On notera que pour un programme pas trop complexe, il n'existe pas de niveau intermédiaire entre le premier et le dernier niveaux : on se retrouve donc avec une analyse entièrement structurée. Autrement dit, l'approche objet ne renie pas la programmation structurée, mais la considère plutôt comme un cas limite «dégénéré» pour des programmes simples[2].

### *8.1 .6 Le choix du critère vertical*

Il existe encore un problème important que nous avons laissé en suspens : le choix du critère de décomposition verticale. La notion d'abstraction d'objets du monde réel nous fournit un critère de décomposition horizontale nous permettant de diviser le problème en un ensemble d'objets, représentés par des types de données abstraits (ou des machines abstraites). Cependant, dès que le projet atteint une taille importante, il devient nécessaire d'organiser les objets à plusieurs niveaux. Saisir à la fois tous les aspects d'un objet serait trop complexe ; il convient donc d'organiser la décomposition des objets.

Il existe plusieurs façons de faire qui, tout en partageant la même notion d'objet en tant que critère de décomposition horizontale, diffèrent par le critère de décomposition verticale choisi. Il n'y a donc pas une, mais *des* méthodes orientées objet. Ce point est rarement évoqué et conduit à d'innombrables discussions entre les partisans de telle ou telle méthode, persuadés de détenir seuls la «vraie» orientation objet.

Dans les chapitres suivants, nous présenterons plusieurs de ces méthodes. Les principales utilisent soit la *composition* comme critère vertical, soit la *classification*. D'autres organisations sont possibles, et rien ne dit que toutes les façons de faire aient déjà été publiées.

---

1 Titre obligeamment emprunté à un article de M. Gauthier [Gau92].

2 Ce qui explique qu'on ait pu utiliser si longtemps – et avec succès – la programmation structurée, jusqu'à l'explosion de complexité des logiciels que nous connaissons actuellement.

## 8.2 L'approche par composition

### *8.2 .1 Principes de la méthode*

#### a) Notion de niveaux d'abstraction

Il serait faux de croire qu'il existe un ensemble unique de valeurs et d'opérations permettant de caractériser entièrement une abstraction. Tout dépend du point de vue ou, si l'on veut, de l'utilisation que l'on veut faire de l'abstraction.

Une voiture comporte un moteur ; ce qui le caractérise (pour l'utilisateur moyen), c'est sa cylindrée, le carburant utilisé, sa puissance et sa vitesse maximale... plus bien entendu les différentes commandes permettant de le faire fonctionner, et le fait qu'il entraîne la voiture. Le constructeur aura une vue très différente du même moteur : il s'agira pour lui avant tout d'un ensemble de pièces détachées, qu'il faut assembler dans un ordre donné. L'ingénieur responsable de la conception du moteur le verra à son tour comme un ensemble de pièces métalliques de coefficients de dilatation différents, dont la taille change avec la température et qu'il s'agit d'assembler de façon que l'ensemble n'explose pas en service normal.

Laquelle de ces trois vues est la bonne ? Aucune, et toutes. Ce sont trois abstractions *différentes* du *même* objet. De plus, ces abstractions sont hiérarchisées : elles vont de la vue la plus générale vers la plus élémentaire, chaque niveau *ignorant les détails d'implémentation de la vue inférieure*.

Nous ne saurions trop insister sur le fait que cette hiérarchisation en niveaux d'abstraction non seulement est naturelle, mais que sans elle la vie de tous les jours serait impossible. Si seuls les gens capables de comprendre son fonctionnement interne pouvaient allumer un téléviseur, certaines émissions auraient moins d'audience...

Non seulement la hiérarchie des niveaux d'abstraction est partout, mais souvent un problème ne se résoudra que s'il est traité au niveau approprié. Par exemple, vous pouvez écrire :

```
I := 1;
```

et vous pensez : «La variable I prend la valeur 1.» Il s'agit bien entendu d'une abstraction : si vous ouvrez un ordinateur vous ne verrez rien qui ressemble à la notion de «variable I». En fait, le compilateur remplacera cette instruction par la suite d'instructions assembleur :

```
MOVA, #1
MOVI, A
```

Beaucoup de programmeurs pensent que c'est ce qui se passe «pour de vrai» dans l'ordinateur (et donc estiment que l'assembleur est le plus «vrai» des langages). Faux ! La notion d'instruction machine n'est à son tour qu'une abstraction : c'est un point d'entrée dans la mémoire de microprogrammes du processeur. Et même les micro-instructions ne sont que des suites de bits qui ouvrent et qui ferment des portes logiques... Mais bien entendu, la notion de porte logique n'est, elle encore, qu'une abstraction (commode) pour désigner un ensemble de transistors arrangés d'une certaine façon... On pourrait continuer ainsi jusqu'au niveau des atomes et des électrons individuels. Si maintenant le programme contient une erreur, par exemple si vous auriez dû écrire :

```
I := 0;
```

cela se traduira bien par le fait que le nombre d'électrons emprisonnés dans le puits quantique d'une jonction de transistor faisant partie d'une mémoire ne sera pas ce qu'il devrait être. Vous n'allez pas pour autant chasser les erreurs de programmation au microscope électronique ! Inversement, si un circuit d'un ordinateur est en panne, il sera très difficile de l'identifier au moyen d'un programme de haut niveau : seul l'analyseur logique de circuits permettra de le mettre en évidence.

Retenons que tous les problèmes de l'existence sont structurés en niveaux d'abstraction ; que pour chaque problème à résoudre, il existe un niveau d'abstraction approprié à sa solution, et que ni les

niveaux inférieurs, ni les niveaux supérieurs ne seront satisfaisants. Et que donc en ce qui concerne les problèmes purement informatiques, la bonne définition des niveaux d'abstraction est ce qu'il y a de plus fondamental dans l'architecture d'un projet.

### b) La COO par composition

Dans cette méthode, la décomposition verticale s'effectue par *niveaux d'abstraction* de plus en plus élémentaires ; les objets sont organisés en strates correspondant chacune à un niveau d'abstraction différent.

Un niveau d'abstraction (donc un module) définira *toutes* les propriétés abstraites d'un objet, pour une certaine *vue*. On passera au niveau suivant (définition de sous-modules) en effectuant un «zoom» sur l'objet pour en voir les détails : au lieu de considérer l'objet comme un tout, on s'intéressera alors à l'ensemble des objets de plus bas niveau qui le composent.

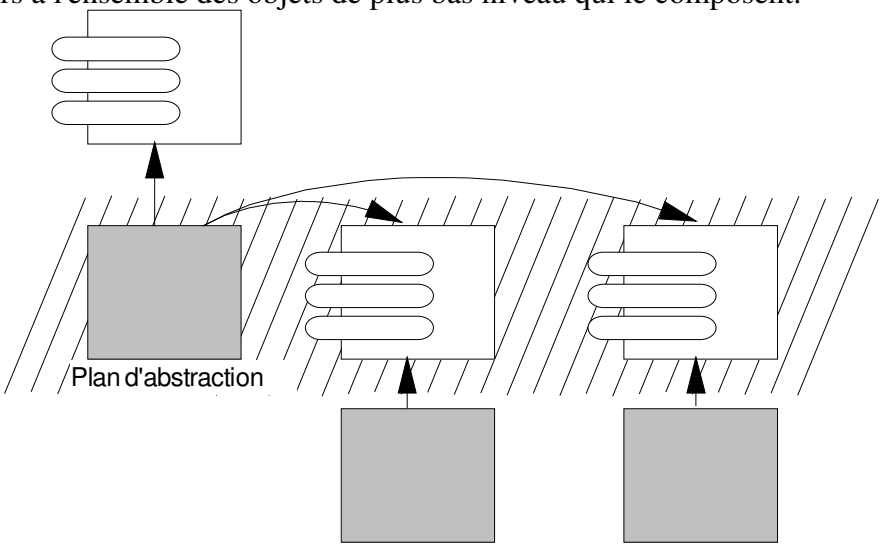


**Figure 15 :** Plans d'abstraction

Ce mécanisme autorise le développement d'un projet par *plans d'abstraction* successifs (Fig. 15). A un moment donné, on travaille sur une vue concrète (celle que l'on réalise) qui nécessitera la définition des vues abstraites (spécifications) du niveau inférieur. Un plan d'abstraction comprendra donc l'*implémentation* de l'abstraction considérée (les boîtes grisées de la figure 16), et les *spécifications* des niveaux immédiatement inférieurs. Les flèches verticales expriment la dépendance d'une implémentation à sa propre spécification, alors que les flèches horizontales expriment la dépendance d'une implémentation aux spécifications des objets qui la composent.

La notion de plan d'abstraction est renforcée par le mécanisme de compilation Ada : un plan sera constitué du corps d'une unité et des spécifications nommées par lui dans des clauses **`with`**. Il sera alors possible de faire faire certaines vérifications de cohérence par le compilateur. En effet, les objets informatiques étant proches des objets du monde réel, et le compilateur Ada d'une rigueur extrême sur tout ce qui concerne les cohérences de type, on constate expérimentalement qu'une incohérence dans la définition d'un objet entraîne souvent une incohérence de type détectée par le compilateur. On peut ainsi utiliser le compilateur comme moyen de vérification de la conception, moyen qui sera d'autant plus efficace que le programmeur aura été plus rigoureux dans la définition de ses types ; on obtient alors une «prime à la rigueur» qui n'est pas un des effets les moins intéressants du couplage de la méthode par composition au langage Ada.

Considérons, pour illustrer cette démarche, la notion de fichier indexé. Qu'est-ce qu'un fichier indexé ? Pour la vue «utilisateur», de plus haut niveau, c'est un ensemble de données dont les éléments sont accessibles au moyen d'une clé, le tout constituant un objet unique, ce que nous avons symbolisé (Fig. 16) par la présence d'une fonctionnalité `Read` (lecture) et d'une fonctionnalité `Seek` (recherche au moyen d'une clé). A un plus grand niveau de détail (premier niveau d'implémentation), on pourra voir que ce fichier (logique) est constitué, par exemple, d'un fichier de

données et d'un fichier d'index, constitués de fichiers directs au sens d'Ada : chaque élément est accessible par un `Read` aléatoire. Noter qu'il n'y a qu'une seule flèche entre l'implémentation de `Fichier_Indexé` et la spécification de `Fichier_Direct`, bien que l'on utilise deux fichiers directs : la signification de celle-ci est que la notion de fichier indexé utilise, pour son implémentation, la notion de fichier direct. Elle exprime une dépendance logique, non une relation d'inclusion. A un plus bas niveau d'abstraction, chacun de ces fichiers est implémenté au moyen de la notion de fichier du système d'exploitation, qui eux-mêmes se résument au niveau physique à un ensemble de secteurs sur un disque dur... Ces différentes *vues* du fichier sont indépendantes : une autre implémentation du fichier indexé pourrait utiliser d'autres mécanismes. Aucune des propriétés spécifiques des fichiers d'index ou de données n'est transmise au fichier indexé, mais le comportement global du fichier indexé résulte de l'assemblage de ses différents composants.

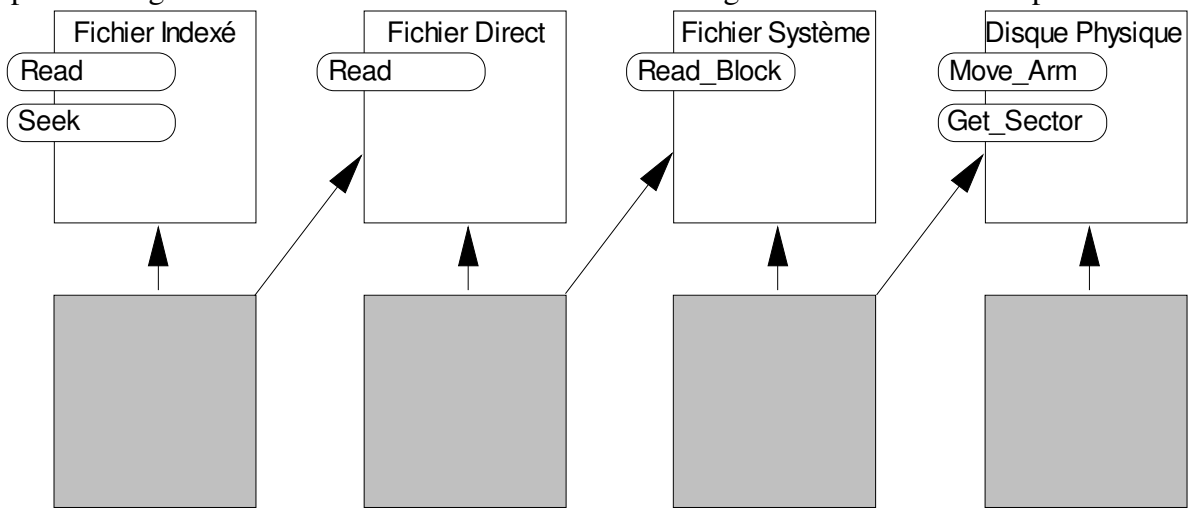


**Figure 16 :** Représentation d'un fichier indexé en composition

Comme dans tout développement de programme, cette méthode conduit à une décomposition en modules ; mais ici, un module contient *tous* les aspects d'*un* objet particulier. Ceci permet un regroupement des fonctionnalités utiles à haut niveau en occultant tous les détails d'implémentation : on réalise ainsi l'*indépendance* entre la vue abstraite d'un objet et son implémentation, ainsi qu'entre objets utilisant des sous-objets communs. On ne définit que des relations de «prestataire de service» à «utilisateur», ou des relations entre niveaux d'abstraction différents. La topologie des programmes construits selon cette méthode se présente comme des graphes généraux acycliques et *non transitifs* : toute modification d'un module n'affecte potentiellement que les niveaux directement inférieurs (et non pas *tous* les niveaux inférieurs comme en programmation structurée), et réciproquement tout niveau ne connaît que le niveau qui lui est immédiatement supérieur. La complexité des dépendances diminue considérablement, et les relations entre modules deviennent beaucoup plus faciles à gérer.

## *8.2 .2 Exemple en composition*

Nous voulons réaliser un «cahier de comptes» pour gérer une comptabilité personnelle. Nous allons nous appuyer sur ce que nous connaissons bien, le cahier que nous gérions auparavant à la main. Qu'*est-ce* donc qu'un cahier de comptes ? C'est un *ensemble ordonné* d'*écritures*. Nous voyons donc apparaître deux objets : un gestionnaire de données de type machine abstraite qui est le cahier lui-même, et un type de donnée abstrait qui est la notion d'écriture.

Le cahier doit offrir les fonctionnalités habituelles d'une liste séquentielle (insérer et supprimer des éléments, parcourir la liste, etc.). Il possède une propriété supplémentaire : les écritures doivent rester triées par rapport à un ordre courant. Nous imposons donc que la fonction `Insérer` ajoute toujours une écriture *à la bonne place*. Mais pour cela, il faut pouvoir comparer plusieurs écritures

en fonction d'un critère courant. Comment définir ce critère ? Le plus simple est de le définir par rapport à une fonction de comparaison. Nous devons également fournir un point d'entrée pour changer de critère de comparaison. Comme le cahier est évidemment un gestionnaire de données, nous pouvons considérer qu'il existe toujours une position courante, et que la suppression d'une écriture porte toujours sur l'écriture courante. Nous pouvons décrire cette vue du cahier comme :

```
with Les_Ecritures;
package Cahier_Compta is
   use Les_Ecritures;

   procedure Insérer (L_Ecriture : Ecriture);
   procedure Supprimer;
   function  Ecriture_Courante return Ecriture;

   type Déplacement is (Début, Fin, Précédent, Suivant);
   procedure Changer_Position (Vers : Déplacement);

   function Début_Cahier   return Boolean;
   function Fin_Cahier     return Boolean;

   Erreur_Déplacement : exception;

   type Fonction_Ordre is
     access function (X, Y : Ecriture) return Boolean;

   procedure Changer_Ordre (Comparaison: Fonction_Ordre);
end Cahier_Compta;
```

Une écriture comprend plusieurs éléments : une date d'opération, une date de valeur, un texte descriptif et un montant. En termes d'opérations, il faut pouvoir éditer une écriture, c'est-à-dire permettre à l'utilisateur de modifier son contenu. Enfin, nous devons fournir des fonctions de comparaison entre écritures. Nous pouvons décrire ainsi la vue abstraite des `Ecriture` :

```
with Ada.Calendar;
package Les_Ecritures is
   type Argent is delta 0.01 digits 10;

   use Ada.Calendar;
   type Ecriture is
     record
        Date_Opération : Time;
        Date_Valeur    : Time;
        Descriptif     : String(1..20);
        Montant        : Argent;
     end record;

   procedure Editer (L_Ecriture : Ecriture);

   function Date_Op_Inférieur (Gauche, Droite : Ecriture)
     return Boolean;
   function Date_Val_Inférieur (Gauche, Droite : Ecriture)
     return Boolean;
   function Montant_Inférieur (Gauche, Droite : Ecriture)
     return Boolean;
end Les_Ecritures;
```

Enfin, un programme de ce type n'a de sens que s'il est interactif. Il nous faut donc représenter l'interface utilisateur. Une interface digne des environnements modernes sortirait du cadre de cet exemple[1], mais nous pouvons toujours en faire une abstraction simplifiée : l'interface est ce qui permet au programme de recevoir des *ordres* de la part de l'utilisateur. Ces ordres sont des entités élémentaires. Il nous faudrait aussi différents moyen d'envoyer des données à l'utilisateur. A ce niveau, nous ne nous préoccupons pas directement des problèmes d'implémentation, aussi nous en tiendrons-nous au niveau *logique* : par exemple, nous pouvons prévoir d'envoyer des messages d'erreur à l'écran. Nous sommes trop haut dans les niveaux d'analyse pour spécifier des choses telles

[1] Dans le programme réel qui nous a inspiré cet exemple, environ 80% du code est lié à l'interface utilisateur !

que faire apparaître une fenêtre avec un beau point d'exclamation pour signaler un problème : nous prévoyons simplement d'envoyer des messages d'erreur, indépendamment de toute représentation. Nous pouvons exprimer ceci comme :

```
package Interface_Utilisateur is
   type Ordre is
     (Sortir,
      Avancer, Reculer,    Aller_Début,  Aller_Fin,
      Insérer, Modifier,   Supprimer,
      Trier_date_valeur,   Trier_date_opération,
      Trier_montant);
   function Ordre_Suivant return Ordre;

   procedure Signaler_Erreur (Texte : String);
end Interface_Utilisateur;
```

Ayant ainsi défini les briques de base, il ne nous reste plus qu'à les assembler pour vérifier qu'elles nous permettent de réaliser le comportement désiré :

```
with Interface_Utilisateur, Les_Ecritures, Cahier_Compta;
procedure Compta_Perso is
   use Cahier_Compta, Interface_Utilisateur, Les_Ecritures;

begin
   Cahier_Compta.Changer_Ordre (Date_Val_Inférieur'Access);
   loop
      case Ordre_Suivant is
      when Sortir =>
         exit;

      when Aller_Début =>
         Cahier_Compta.Changer_Position (Début);

      when Aller_Fin =>
         Cahier_Compta.Changer_Position (Fin);

      when Avancer =>
         if not Cahier_Compta.Fin_Cahier then
            Cahier_Compta.Changer_Position (Suivant);
         end if;

      when Reculer =>
         if not Cahier_Compta.Début_Cahier then
            Cahier_Compta.Changer_Position (Précédent);
         end if;

      when Supprimer =>
         Cahier_Compta.Supprimer;

      when Insérer =>
         declare
            Nouvelle_Ecriture : Ecriture;
         begin
            Editer (Nouvelle_Ecriture);
            Cahier_Compta.Insérer (Nouvelle_Ecriture);
         end;

      when Modifier =>
         declare
            Ancienne : constant Ecriture
                     := Cahier_Compta.Ecriture_Courante;
            Nouvelle : Ecriture := Ancienne;
         begin
            Editer (Nouvelle);
            if Nouvelle /= Ancienne then
               Cahier_Compta.Supprimer;
               Cahier_Compta.Insérer (Nouvelle);
            end if;
         end;
```

```
         when Trier_date_valeur =>
            Cahier_Compta.Changer_Ordre
               (Date_Val_Inférieur'Access);

         when Trier_date_opération =>
            Cahier_Compta.Changer_Ordre
               (Date_Op_Inférieur'Access);

         when Trier_montant =>
            Cahier_Compta.Changer_Ordre
               (Montant_Inférieur'Access);
         end case;
      end loop;
   end Compta_Perso;
```

Il manque bien sûr encore de nombreuses fonctionnalités (à commencer par la sauvegarde du cahier sur fichier...), mais l'important est que nous avons mis en place une structure où le rôle de chaque module est clairement identifié : *tout* ce qui concerne la définition et les propriétés des écritures prises individuellement se trouve dans le paquetage `Les_Ecritures` ; *tout* ce qui concerne le cahier, c'est-à-dire la gestion de l'ensemble d'écritures, est dans `Cahier_Compta` ; *tout* ce qui concerne les interfaces est dans `Interface_Utilisateur`. A partir de là, il est facile de rajouter des fonctionnalités en sachant toujours où intervenir.

L'autre point important est que cette structure nous permet de définir des comportements simples et des invariants : par exemple, `Insérer` ajoute toujours l'écriture au bon endroit compte tenu du critère de tri courant. Un phénomène très intéressant, qui s'est produit lorsque nous avons développé le projet réel d'où est tiré cet exemple, montre bien l'amélioration importante de sécurité apportée par cette indépendance. Nous avions une erreur dans l'algorithme de retri de `Changer_Ordre` qui conduisait à des écritures incorrectement ordonnées dans certains cas. Cependant, pour visualiser le cahier, le programme «sortait» une écriture du cahier pour une modification éventuelle, puis la «re-rentrait». A ce moment, le logiciel s'apercevait que l'écriture n'était plus à sa place, et la réinsérait au bon endroit. Autrement dit le logiciel se trouvait dans un état incorrect, mais le simple fait de parcourir le cahier faisait qu'il retournait spontanément à l'état correct. Le logiciel était donc en quelque sorte «autostable». Ce résultat fort rassurant a été obtenu grâce aux principes d'indépendance temporelle et de méfiance réciproque : chaque fonctionnalité ne sait rien ni de qui l'appelle, ni en quelles circonstances. Du coup, chacun assure sa propre sécurité... et corrige éventuellement les erreurs de son voisin. Un tel comportement, qui ne peut être obtenu que par la stricte encapsulation permise par la composition, renforce la confiance que l'on peut avoir quant au bon comportement du logiciel, même face à des événements inattendus.

## 8.3 L'approche par classification

A l'origine de cette approche se trouve l'émergence des langages orientés objet (LOO) [Mas89], dont la classification constitue le cadre méthodologique, en particulier pour maîtriser le mécanisme d'*héritage* qui les caractérise ; on a pris l'habitude d'appeler programmation orientée objet (POO) la programmation avec ces langages. De ce fait, la classification est souvent présentée dans la littérature comme la seule *méthode* orientée objet. Nous avons déjà vu que l'approche par composition permettait de développer des systèmes «orientés objet» sans nécessiter le recours au mécanisme d'héritage, et nous ne saurions trop insister sur le fait que le concept d'objet couvre un domaine beaucoup plus vaste que celui de la seule POO.

### 8.3 .1 Principes de la méthode

Dans cette méthode, le critère de décomposition verticale est la *classification* (au sens de la classification des espèces de Linné [Lin35]). Les objets sont regroupé en *classes*, elles-mêmes décomposées en *sous-classes* plus précises, etc[1]. Pour éviter toute confusion, on appelle *instance* un objet particulier appartenant à une classe.

Par exemple, on regroupera dans la classe `Insecte` les propriétés communes à tous les insectes. Par la suite, on pourra définir des sous-classes comme `Volants` ou `Rampants`, ne contenant respectivement que ce qui est spécifique aux insectes volants (ou rampants), mais commun à *tous* les insectes volants (ou rampants). Les `Volants` peuvent à leur tour se décomposer en `Mouches`, `Guêpes`, etc., suivant l'arborescence de la figure 17. La guêpe qui vient de vous piquer est une *instance* de la classe des `Guêpes`.

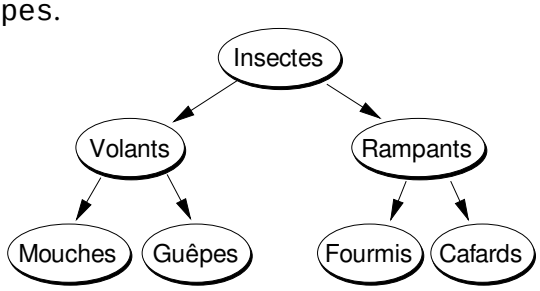


**Figure 17 :** Classification d'insectes

Bien entendu, une mouche *est* un insecte volant, qui *est* lui-même un insecte tout court. Une `Mouche` doit donc également posséder les propriétés des `Volants` et des `Insectes` : les propriétés des superclasses doivent être transmises à leurs descendants. Ce mécanisme est appelé *héritage*, puisqu'une classe enfant *hérite* des propriétés de ses ancêtres. On distingue classiquement l'héritage *simple* (toute classe n'est enfant que d'une seule superclasse) qui conduit à des programmes dont la topologie est un arbre, et l'héritage *multiple* (toute classe peut avoir plusieurs superclasses) qui conduit à des programmes dont la topologie est un graphe général acyclique. Nous reviendrons plus loin sur la question de l'héritage multiple.

Au fur et à mesure que l'on descend dans l'analyse, les objets possèdent de plus en plus de propriétés (celles dont ils ont hérité, plus celles qui sont particulières à la sous-classe), applicables à un nombre de plus en plus restreint d'instances. La descente dans l'analyse conduit donc simultanément à un *enrichissement* et à une *spécialisation* des classes.

Comme nous l'avons vu au paragraphe 8.1.1, une caractéristique de l'orientation objet en général est l'encapsulation, regroupement dans un même module de structures de données et de structures de programme. Dans le vocabulaire de la classification, on a pris l'habitude d'appeler *attributs* les structures de données liées à un objet, et *méthodes* les structures de programme (opérations) associées.

### 8.3 .2 Mécanismes

La mise en application de la classification nécessite des mécanismes spéciaux au niveau du langage de programmation. On qualifie de «langage orienté objet» un langage offrant ces mécanismes. Nous allons présenter ceux offerts par Ada ; d'autres langages orientés objet offrent des fonctionnalités similaires, mais non strictement équivalentes. Nous les comparerons à la fin de ce chapitre.

[1] Tous les objets appartenant à une classe possèdent les mêmes propriétés, ils forment donc une classe d'équivalence au sens mathématique - d'où le nom.

### a) Types étiquetés

En Ada, les mécanismes que nous allons présenter ne sont disponibles qu'avec les types *étiquetés* (*tagged*), c'est-à-dire des types article (**`record`**) déclarés avec le mot réservé **`tagged`**. Ceci n'est pas une restriction, mais une protection supplémentaire. Nous verrons en effet que l'héritage apporte une plus grande souplesse d'évolution, au prix d'un certain affaiblissement du contrôle (statique) des types. Les utilisateurs qui ne souhaitent *pas* utiliser ces mécanismes (comme par exemple les utilisateurs temps réel) ont la garantie, si le mot **`tagged`** n'apparaît pas dans leurs programmes, de bénéficier de la même qualité de vérification statique qu'en Ada 83.

Notons que, comme pour un type non étiqueté, la définition d'un type étiqueté n'inclut pas d'opérations ; une classe complète (c'est-à-dire un type de donnée abstrait muni d'opérations) se réalise en Ada sous la forme d'un paquetage comportant la définition d'un type étiqueté et des sous-programmes portant sur ce type. Les attributs de la classe seront alors les champs du type étiqueté, et les méthodes les sous-programmes associés.

Nous allons présenter progressivement les concepts de la classification en utilisant un exemple très classique, celui d'objets graphiques, figures géométriques destinées à être représentées sur un écran. Tous les objets graphiques possèdent un attribut en commun, leur position sur l'écran, et des méthodes pour les dessiner, les effacer et les déplacer. On peut définir la classe la plus générale comme :

```
package Classe_Objet_Graphique is

   type Coordonnée is range 1..1024;
   type Distance   is range 0..1024;

   type Objet_Graphique is tagged
      record
         X, Y : Coordonnée;
      end record;

   procedure Dessiner (Objet : Objet_Graphique);
   procedure Effacer(Objet : Objet_Graphique);
   procedure Déplacer (Objet : in out Objet_Graphique;
                       En_X   : in    Coordonnée;
                       En_Y   : in    Coordonnée);
end Classe_Objet_Graphique;

package body Classe_Objet_Graphique is
   procedure Dessiner (Objet : Objet_Graphique) is
   begin
      null;
   end Dessiner;

   procedure Effacer(Objet : Objet_Graphique) is
   begin
      null;
   end Effacer;

   procedure Déplacer (Objet : in out Objet_Graphique;
                       En_X   : in    Coordonnée;
                       En_Y   : in    Coordonnée) is
   begin
      Objet := (En_X, En_Y);
   end;
end Classe_Objet_Graphique;
```

> Noter que nous faisons la différence au niveau du typage entre une `Coordonnée` qui représente une position absolue à l'écran, et une `Distance` qui est homogène à la différence de deux coordonnées. Nous aurons besoin de la `Distance` par la suite.

Ceci définit une classe `Objet_Graphique`, munie des attributs `X` et `Y` et des méthodes `Dessiner`, `Effacer` et `Déplacer`. Avec cette définition de la classe, les coordonnées `X` et `Y` sont accessibles de l'extérieur (la définition figure dans la partie visible du paquetage). Nous fournissons

cependant la méthode `Déplacer` pour modifier ces valeurs[1], car il s'agit d'une opération qui conceptuellement pourrait faire plus que simplement modifier les coordonnées. D'autre part, notre classe est trop générale à ce stade pour nous permettre de définir quoi que ce soit de précis pour les méthodes `Dessiner` et `Effacer` ; ces dernières ne font donc rien (nous verrons plus loin une solution plus satisfaisante à ce problème).

### b) Extension de type et héritage ; redéfinition de méthodes

Nous voulons maintenant définir un objet `Cercle`. Un cercle est une sorte d'objet graphique, mais il est plus spécialisé : tous les objets graphiques ne sont pas des cercles. Un cercle possède, en plus d'une position, une autre grandeur caractéristique : son rayon. Nous pouvons définir un type `Cercle` comme une *extension* (un enrichissement) du type `Objet_Graphique` :

```
type Cercle is new Objet_Graphique with
   record
       Rayon : Distance;
   end record;
```

Le type `Cercle` est dérivé du type `Objet_Graphique` ; tous les attributs définis pour le type `Objet_Graphique` sont disponibles. Il disposera également (aura hérité) de méthodes `Dessiner`, `Effacer` et `Déplacer` dont l'implémentation, fournie par le compilateur, consistera à appeler les méthodes correspondantes définies pour `Objet_Graphique` en convertissant le paramètre de `Cercle` en `Objet_Graphique.`

> Ce mécanisme n'est pas nouveau ; il existait déjà en Ada 83 pour les types dérivés non étiquetés, mais seuls les types étiquetés permettent d'enrichir le type en rajoutant des composants (clause **`with record ... end record`**).

On voit immédiatement l'intérêt de cette approche : en cas de redéfinition des caractéristiques de l'`Objet_Graphique`, tous les types dérivés sont remis à jour sans se préoccuper de rien. Si nous souhaitions par exemple ajouter *par la suite* un nouvel attribut à `Objet_Graphique` (une couleur par exemple), aucun changement ne serait nécessaire dans `Cercle` ni dans aucun objet qui serait dérivé directement ou indirectement de `Objet_Graphique`.

Les méthodes définies dans `Objet_Graphique` ne sont cependant pas appropriées pour un `Cercle`. Il faut effectivement faire le dessin, mais `Déplacer` ne fait que changer les attributs sans mettre à jour l'écran. Il arrivera ainsi fréquemment que les méthodes dont hérite un type ne fournissent pas la bonne fonctionnalité : le type dérivé doit définir sa propre façon de faire, sa *méthode* justement, pour réaliser la fonctionnalité abstraite. C'est pourquoi on peut *redéfinir* des sous-programmes dont un type a hérité. Par exemple :

```
type Cercle is new Objet_Graphique with
   record
       Rayon : Distance;
   end record;

procedure Dessiner (Objet : Cercle) is
begin
   Instructions pour dessiner le cercle
end Dessiner;

procedure Effacer(Objet : Cercle) is
begin
   Instructions pour effacer le cercle
end Effacer;
```

[1] Nous reviendrons plus loin sur le problème du déplacement de l'objet.

```
procedure Déplacer (Objet : in out Cercle
                    En_X  : in     Coordonnée;
                    En_Y  : in     Coordonnée) is
begin
   Effacer(Objet);
   Objet := (En_X, En_Y, Objet.Rayon);
   Dessiner(Objet);
end Déplacer;
```

Les règles normales de la surcharge permettent de déterminer la méthode appelée : un appel à `Déplacer` avec un paramètre de type `Objet_Graphique` appellera la méthode d'origine, alors que si le paramètre est de type `Cercle`, c'est la procédure redéfinie qui sera appelée.

### c) Classes, polymorphisme et liaison dynamique

Imaginons maintenant que nous ayons besoin d'un service assez général ; par exemple, une procédure qui dessinerait deux fois un objet, le second légèrement décalé par rapport au premier. Il n'y a pas de raison de rattacher ce service à une classe particulière : c'est un besoin utilisateur, non une propriété fondamentale de l'abstraction des objets graphiques dont hériteraient tous les descendants. Cependant, il faudrait pouvoir l'appliquer à *tous* les objets graphiques ; autrement dit, nous voulons une opération qui porte sur la *classe* des objets graphiques.

C'est possible grâce à l'attribut `'Class` qui s'applique à un type pour désigner la *classe* associée au type, c'est-à-dire le type lui-même ainsi que l'ensemble de tous les types dérivés (directement ou indirectement) du type concerné. Noter la distinction faite entre, par exemple, le type `Objet_Graphique` (qui ne contient que les objets déclarés avec ce type) et la classe `Objet_Graphique'Class` qui inclut tous les objets du type `Objet_Graphique`, ou de types dérivés d'`Objet_Graphique`, tels que `Cercle`. Pour bien marquer cette différence, on utilisera l'appellation *type spécifique* pour parler d'un type particulier à l'intérieur d'une classe, et *type à l'échelle de classe*[1] pour désigner les types «classe» couvrant plusieurs types spécifiques différents. Nous pouvons maintenant définir la procédure ainsi :

```
procedure Dédoubler(L_Objet : in Objet_Graphique'Classe) is
   Copie : Objet_Graphique'Class := L_Objet;
begin
   Dessiner (Copie);
   Déplacer (Copie, En_X => L_Objet.X +10,
                    En_Y => L_Objet.Y +10);
   Dessiner (Copie);
end Dédoubler;
```

Le paramètre n'est plus défini comme appartenant à un type spécifique, mais à une *classe* ; cette procédure peut donc être appelée en fournissant un paramètre réel appartenant à n'importe quel type de la classe, c'est-à-dire aussi bien au type `Objet_Graphique` qu'à `Cercle` ou à n'importe quel autre type dérivé, directement ou indirectement, de `Objet_Graphique`. On exprime ainsi que l'algorithme est applicable à tous les objets qui sont des `Objet_Graphique`[2].

Dans la procédure `Dédoubler`, nous avons un paramètre formel dont le type spécifique est *a priori* inconnu : il dépend de l'objet qui sera passé au sous-programme au moment de l'appel. De même, nous avons une variable locale[3] (`Copie`) déclarée avec un type à l'échelle de classe ; une telle variable doit être initialisée, et c'est le type spécifique de la valeur initiale qui détermine le type de la variable. On parle alors de *polymorphisme*, puisque ces variables peuvent prendre plusieurs types, plusieurs formes, à l'exécution. Ada autorise également des pointeurs sur classe :

```
type Ptr_Objet is access Objet_Graphique'Class;
```

---

1 Traduction lourde, mais fidèle, de *class wide type*.

2 Remarquer l'*affaiblissement du typage* lié à cette façon de faire : une même procédure est appelable avec des paramètres de types différents.

3 En travaillant sur une copie du paramètre formel, nous pouvons garantir que cette procédure ne modifie pas la position de l'objet passé en argument.

Un tel pointeur peut désigner n'importe quel objet de la *classe*, c'est-à-dire de type `Objet_Graphique`, `Cercle`, `Rectangle`, ou de n'importe quel autre type dérivé de `Objet_Graphique`. Nous verrons plus loin un exemple d'utilisation des pointeurs sur classe pour réaliser des *listes hétérogènes*.

La procédure `Dédoubler` pose cependant une question importante. Puisqu'elle va pouvoir s'appliquer à *tout* objet graphique, comment savoir au moment de la compilation s'il faut appeler la méthode `Dessiner` d'origine ou celle redéfinie par `Cercle` ? On ne le peut pas. Ce n'est qu'au moment de l'appel de `Dessiner` par `Dédoubler` que l'on saura s'il faut appeler la méthode `Dessiner` définie sur `Cercle` ou celle définie sur `Rectangle`. On parlera alors de *liaison dynamique* (*dynamic binding*) puisque le travail de mise en relation des sous-programmes à appeler ne peut être effectué dès la compilation, mais seulement à l'exécution, pour chaque appel. On peut comprendre ce mécanisme comme une résolution de surcharge qui aurait lieu à l'exécution ; c'est le type de l'*instance* (le paramètre réel fourni au sous-programme `Dédoubler`) qui détermine le sous-programme effectivement appelé. Ceci implique de garder *à l'exécution* la trace de son type avec chaque valeur. Cette «marque d'origine» est l'étiquette, le fameux *tag*, ce qui explique que cette façon de programmer ne soit possible qu'avec des types étiquetés.

La liaison dynamique va nous permettre de résoudre élégamment le problème de la procédure `Déplacer`. Jusqu'à présent, nous devions la redéfinir pour chacun des types dérivés, car celle dont nous héritions ne faisait que changer la position. On pourrait penser définir le `Déplacer` d'`Objet_Graphique` comme :

```
procedure Déplacer (Objet : in out Objet_Graphique;
                    En_X  : in     Coordonnée;
                    En_Y  : in     Coordonnée) is
begin
   Effacer(Objet);
   Objet := (En_X, En_Y);
   Dessiner (Objet);
end Déplacer
```

mais ce ne serait pas suffisant, car le `Déplacer` dont hériteraient les classes dérivées continuerait à appeler les méthodes `Dessiner` et `Effacer` définies pour des `Objet_Graphique`. Or il est évident que pour déplacer un cercle, il faut redessiner un cercle, et que pour déplacer un rectangle, il faut redessiner un rectangle ! Autrement dit, il faudrait que dans `Déplacer`, les appels à `Dessiner` et `Effacer` s'effectuent de façon dynamique, en fonction du type effectif du paramètre réel. C'est possible en faisant «oublier» au compilateur le type effectif du paramètre, c'est-à-dire en le convertissant vers la classe :

```
procedure Déplacer (Objet : in out Objet_Graphique;
                    En_X  : in     Coordonnée;
                    En_Y  : in     Coordonnée) is
begin
   Effacer(Objet_Graphique'Class (Objet));
   Objet := (En_X, En_Y);
   Dessiner (Objet_Graphique'Class (Objet));
end Déplacer
```

Dans les appels ci-dessus, le type effectif du paramètre de l'appel à `Effacer` et `Dessiner` n'est plus connu ; le compilateur ira donc consulter l'étiquette associée au paramètre formel `Objet` pour décider quelle méthode appeler. Nous obtiendrons bien la liaison dynamique avec la méthode définie pour le paramètre réel (celui fourni par l'appel).

En résumé, il suffit de se rappeler que le type *spécifique* de l'expression figurant dans un appel de sous-programme détermine le sous-programme appelé. Si ce type n'est pas connu à la compilation (c'est-à-dire si l'expression est d'un type à l'échelle de classe), alors il y a liaison dynamique et le sous-programme appelé est déterminé par l'étiquette du paramètre.

### d) Liaison avec le parent

Lorsque l'on redéfinit des méthodes héritées, il est fréquent que l'on souhaite simplement ajouter certains traitements au traitement «de base» fourni par le type parent. Il faudra donc appeler la méthode définie par le parent depuis le corps d'une méthode d'un enfant. Ceci se fait en Ada par le même mécanisme que celui vu précédemment : il suffit d'appeler le sous-programme voulu en effectuant une conversion vers le type qui gouverne la méthode que l'on souhaite appeler, puisque seule la cohérence des types détermine le sous-programme appelé. Par exemple, nous pouvons avoir un objet `Cercle_Pointé`, analogue à `Cercle`, mais où l'on marque le centre du cercle par un point. Nous pouvons le définir ainsi :

```
type Cercle_Pointé is new Cercle with null record;

procedure Dessiner (Objet : Cercle_Pointé) is
begin
   Dessiner (Cercle (Objet));
   -- Dessiner le centre
end Dessiner;

procedure Effacer  (Objet : Cercle_Pointé) is
begin
   -- Effacer le centre
   Effacer (Cercle (Objet));
end Dessiner;
```

Nous créons un nouveau type `Cercle_Pointé`, mais comme il ne comporte pas de nouveau champ, nous devons le signaler par la clause **`with null record`**. Il n'est pas nécessaire de récrire la procédure de dessin du cercle : l'avantage de la POO est qu'on ne recode que la différence entre le nouveau comportement et celui que l'on possédait déjà. Par conséquent, le `Dessiner` d'un `Cercle_Pointé` appelle le `Dessiner` de `Cercle` grâce à une conversion de type ; de même pour `Effacer`. On peut comprendre cet appel comme signifiant : «Dessiner ce `Cercle_Pointé` en le considérant comme un `Cercle`.» Noter au passage que le type figurant dans l'appel est connu statiquement, il n'y a donc pas de liaison dynamique. Remarquer également que dans le cas d'une hiérarchie complexe, n'importe quel enfant peut choisir d'appeler une des méthodes de n'importe lequel de ses ancêtres, simplement en convertissant le paramètre dans le type approprié.

### e) Types abstraits

Il est évident que tout objet graphique doit pouvoir être dessiné à l'écran ; c'est pourquoi nous avons défini les méthodes `Dessiner` et `Effacer` dans la classe `Objet_Graphique`. Nous avons dû leur donner des implémentations vides, car la classe `Objet_Graphique` est trop générale pour pouvoir décrire quoi faire. En fait, cela n'aurait aucun sens de définir un `Objet_Graphique` qui ne serait pas également quelque chose de plus précis : la classe de départ n'a d'intérêt que pour définir des sous-classes, non des instances. Nous pouvons exprimer cette contrainte en déclarant le type `Objet_Graphique` de la façon suivante :

```
type Objet_Graphique is abstract tagged
    record
        X, Y : Coordonnée;
    end record;
```

ce qui nous permet maintenant de définir les méthodes `Dessiner` et `Effacer` comme des procédures abstraites :

```
procedure Dessiner(Objet : Objet_Graphique) is abstract;
procedure Effacer (Objet : Objet_Graphique) is abstract;
```

De tels sous-programmes ne sont pas définis effectivement (on ne fournit pas d'implémentation), ils ne servent qu'à marquer que les types dérivés d'`Objet_Graphique` devront obligatoirement redéfinir ces procédures. Comme il est nécessaire que les utilisateurs soient informés de cette propriété, on ne peut définir de sous-programmes abstraits d'un type étiqueté que si le type a

lui-même été déclaré **abstract**. Le type Objet_Graphique, ainsi que les types dérivés qui n'auraient pas redéfini les sous-programmes abstraits, sont appelés *types abstraits*[1] : il est interdit de déclarer des objets ou des valeurs de ces types puisque toutes leurs propriétés ne seraient pas définies ; ils ne peuvent servir qu'à dériver d'autres types.

### f) Dissimulation d'information

Les attributs du type Objet_Graphique tel que défini jusqu'à présent sont totalement visibles. En général, on préférera contrôler l'accès aux attributs. Il faut pour cela en faire un type privé, mais bien entendu il est nécessaire de spécifier que même privé, le type est étiqueté, afin de pouvoir l'utiliser en tant que tel dans des dérivations ultérieures. Notre spécification de paquetage va donc devenir :

```
package Classe_Objet_Graphique is

   type Coordonnée is range 1..1024;
   type Distance   is range 0..1024;

   type Objet_Graphique is abstract tagged private;

   procedure Positionner (Objet : in out Objet_Graphique;
                          En_X  : in     Coordonnée;
                          En_Y  : in     Coordonnée);

   function X_courant (Objet : Objet_Graphique)
     return Coordonnée;
   function Y_courant (Objet : Objet_Graphique)
     return Coordonnée;

   procedure Déplacer (Objet : in out Objet_Graphique;
                       En_X  : in     Coordonnée;
                       En_Y  : in     Coordonnée);

   procedure Dessiner(Objet : Objet_Graphique) is abstract;
   procedure Effacer (Objet : Objet_Graphique) is abstract;

private
   type Objet_Graphique is abstract tagged
     record
       X, Y : Coordonnée;
     end record;
end Classe_Objet_Graphique;
```

La structure étant devenue cachée, nous avons fourni un constructeur (Positionner) et deux sélecteurs (X_Courant et Y_Courant) afin de permettre respectivement de donner une valeur aux coordonnées et de retrouver leurs valeurs courantes. Réciproquement, il est possible d'étendre un type de façon privée, c'est-à-dire en cachant à l'utilisateur le contenu de l'extension. Par exemple, Cercle pourrait être défini comme suit :

```
with Classe_Objet_Graphique; use Classe_Objet_Graphique;
package Classe_Cercle is
   type Cercle is new Objet_Graphique with private;

   procedure Dessiner (Objet : Cercle);
   procedure Effacer  (Objet : Cercle);

private
   type Cercle is new Objet_Graphique with
     record
       Rayon : Distance;
     end record;
end Classe_Cercle;
```

[1] Attention : le terme «type abstrait» est défini par le langage et désigne un type muni de la clause **is abstract**, à ne pas confondre avec la notion générale de type représentant une abstraction d'une entité du monde réel, pour laquelle nous utilisons le terme «type de donnée abstrait».

Remarquer la spécification explicite des procédures `Dessiner` et `Effacer` : le compilateur sait ainsi que le paquetage va fournir ces sous-programmes, et que le type peut ne pas être abstrait.

### g) Facettes multiples

Dans l'exemple que nous avons vu, la classe `Cercle` héritait des propriétés d'une seule classe, `Objet_Graphique`. Or il arrive parfois que l'on veuille considérer des objets complexes selon plusieurs vues, plusieurs facettes. Par exemple, on peut définir une classe `Figure_Géométrique` permettant d'obtenir diverses caractéristiques (périmètre, surface...) des objets géométriques.

Il est bien certain qu'un cercle est un objet graphique, mais c'est également un objet géométrique. Il n'est ni nécessaire, ni souhaitable de rassembler ces deux notions dans une même classe, car elles sont orthogonales : les segments sont des objets graphiques tout à fait respectables, mais cela n'aurait pas de sens de parler de leur périmètre ni de leur surface. Il faudrait donc ajouter à `Cercle` les propriétés liées à la notion de `Figure_Géométrique` en plus de celles héritées de la classe `Objet_Graphique` ; nous dirons que nous lui rajoutons une *facette* de figure géométrique. Ceci se fait facilement en Ada au moyen de la technique d'*enrichissement de classe* par des génériques. Nous pouvons écrire un générique destiné à fournir des propriétés géométriques à toute autre classe :

```
with Grandeurs; -- Définit Longueur, Surface etc.
generic
   type Objet is abstract tagged limited private;
package Géométrique is
   type Objet_Géométrique is
      abstract new Objet with private;

   use Grandeurs;
   function Aire (L_objet : Objet_Géométrique)
      return Surface;
   function Périmètre (L_objet : Objet_Géométrique)
      return Longueur;
   -- etc...

private
   type Objet_Géométrique is new Objet with null record;
end Géométrique;
```

A partir de là, nous pouvons utiliser ce générique pour créer un nouveau type *enrichi* de propriétés géométriques :

```
with Géométrique;
with Classe_Objet_Graphique;
package Classe_Graphique_Géométrique is
   new Géométrique(Classe_Objet_Graphique.Objet_Graphique);
```

Nous pouvons dériver de nouveaux objets graphiques indifféremment de `Objet_Graphique_Géométrique` si nous voulons à la fois les deux facettes, ou seulement de `Objet_Graphique` pour ceux (les segments par exemple) dont les propriétés géométriques n'ont aucun sens. Si nous considérons maintenant un cercle comme un objet géométrique aussi bien que graphique, il faut définir les méthodes pour le dessiner, l'effacer, ainsi que pour en calculer l'aire et le périmètre ; nous devons donc (re)définir les procédures correspondantes :

```
with Grandeurs;
with Classe_Graphique_Géométrique;
package Classe_Cercle is
   type Cercle is new Objet_Graphique with private;

   procedure Dessiner (Objet : Cercle);
   procedure Effacer  (Objet : Cercle);

private
   type Cercle is
      new Classe_Graphique_Géométrique.Objet_Géométrique
      with record
           Rayon : Distance;
        end record;

   use Grandeurs;
   function Aire      (L_objet : Cercle) return Surface;
   function Périmètre (L_objet : Cercle) return Longueur;
end Classe_Cercle;
```

Dans cet exemple, nous annonçons, en partie visible, uniquement que le `Cercle` est un `Objet_Graphique` et nous n'exposons donc que la redéfinition des méthodes correspondantes. Dans la partie privée, nous le dérivons *en fait* de `Classe_Graphique_Géométrique`, et nous redéfinissons en partie privée les méthodes des objets géométriques. C'est autorisé, car `Classe_Graphique_Géométrique` est lui même dérivé de `Objet_Graphique`. Ainsi, nous n'exportons aux utilisateurs extérieurs que le `Cercle` en tant qu'`Objet_Graphique`, mais l'implémentation bénéficie en plus de la facette d'`Objet_Géométrique`. Bien entendu, on aurait pu laisser cette facette visible, mais cet exemple montre l'extrême précision que l'on peut obtenir dans le contrôle des propriétés exportées et de celles conservées privées.

## *8.3 .3 Utilisation de l'héritage*

L'héritage, comme nous venons de le voir, s'appuie sur des mécanismes particuliers au niveau du langage. Nous allons maintenant présenter quelques utilisations typiques de ces mécanismes. Nous verrons ensuite un exemple d'utilisation plus «méthodologique» au service de la classification.

### a) Gestionnaires de données hétérogènes

Il est parfois nécessaire de définir des listes de données (ou tout autre style de gestionnaire de données) qui ne sont pas toutes du même type. Typiquement, dans une interface graphique, on maintiendra une liste de tous les *widgets*[1] actifs à l'écran à un moment donné. Bien sûr, ceci n'a de sens que si tous les éléments hétérogènes ont certaines propriétés communes, sinon on ne pourrait rien en faire ; il est donc logique de considérer qu'ils doivent tous appartenir à une même classe. La construction de telles structures est facile en Ada grâce aux pointeurs sur classe. Supposons par exemple que nous ayons défini ainsi la classe des widgets :

```
package Classe_Widget is
   type Coordonnée is range 1..1024;

   type Widget is abstract tagged record
      Xmin, Xmax: Coordonnée;
      Ymin, Ymax: Coordonnée;
   end record;

   procedure Evénement_Souris (Dans       : Widget;
                               En_X, En_Y : Coordonnée)
      is abstract;
```

[1] Abréviation homologuée pour *window gadget*, toute «bricole» apparaissant à l'écran, telle que fenêtre, bouton, menu, etc.

```
   procedure Afficher (Le_Widget : Widget);
end Classe_Widget;
```

Le widget occupe à l'écran un espace déterminé par ses coordonnées. La procédure `Afficher` demande au widget de se redessiner. En cas de «clic» de la souris sur un widget actif, on appelle la procédure `Evénement_Souris` en donnant les coordonnées *relatives* (c'est-à-dire par rapport à `Xmin, Ymin`). A partir de là, le widget peut déterminer ce qu'il doit faire de façon indépendante de sa position absolue à l'écran, la liaison dynamique permettant à chaque widget d'avoir *sa* méthode pour répondre à un clic. Cette classe abstraite sera étendue pour obtenir des widgets concrets, comme par exemple :

```
with Classe_Widget; use Classe_Widget;
package Classe_Menu is
   type Menu is new Widget with private;

   procedure Evénement_Souris (Dans       : Menu;
                               En_X, En_Y : Coordonnée);

   type Numéro_Choix is range 1..25;

   procedure Ajouter_Choix (A : Menu; Choix : String);
   function  Elément_Choisi (Dans : Menu)
     return Numéro_Choix;
private
   ...
end Classe_Menu;
```

Bien sûr, il faut avoir la liste de tous les widgets à l'écran, pour déterminer où le clic a eu lieu et quel est le widget concerné. Une telle gestion de liste peut s'exprimer comme :

```
with Classe_Widget; use Classe_Widget;
package Liste_Widgets is
   type Poignée_Widget is access all Widget'Class;

   procedure Enregistrer    (Le_Widget : Widget'Class);
   procedure Désenregistrer (Poignée   : Poignée_Widget);

   -- Fonction d'itérateur
   function Premier_Widget return Poignée_Widget;
   function Widget_Suivant return Poignée_Widget;
   function Fin_De_Liste   return Boolean;
   Plus_De_Widget : exception;
end Liste_Widgets;
```

Cette liste nous permet d'enregistrer tous les objets dérivés de `Widget`, comme des menus, boutons, etc. Noter que nous n'avons pas besoin de savoir le type précis (spécifique) du widget pour accéder aux éléments `Xmin`, `Xmax`, `Ymin`, `Ymax` ni pour appeler la procédure `Evénement_Souris`.

Supposons maintenant que nous voulions effectuer une opération spéciale uniquement sur les widgets qui sont des menus, comme de rajouter un élément à tous les menus affichés. Ce problème est difficile, car tout ce que l'on sait *a priori* des éléments de la liste est qu'ils appartiennent à un type dérivé plus ou moins directement de `Widget`. Il faut donc «remonter» le typage de «un widget» vers la notion plus précise de «un menu». Nous pouvons tester si une valeur appartient à un type par l'opérateur **`in`** et convertir un type à l'échelle de classe vers n'importe lequel de ses descendants. Par exemple :

```
declare
   Poignée : Poignée_Widget := Premier_Widget;
begin
   while not Fin_De_Liste loop
      if Poignée.all in Menu then
         Ajouter_Choix(A     => Menu (Poignée.all),
                       Choix => "Coucou");
      end if;
   end loop;
end;
```

Si nous n'avions pas pris la précaution de le tester avant, il se pourrait que nous tentions de convertir en `Menu` quelque chose qui serait un autre dérivé de `Widget` – un bouton par exemple ; dans ce cas, il y aurait automatiquement levée de l'exception `Constraint_Error`. La cohérence des types reste donc garantie, même en présence de structures hétérogènes. Cependant, cette cohérence ne peut être assurée qu'à l'exécution, alors que pour les types non étiquetés elle est toujours vérifiée à la compilation : on voit ici encore qu'une plus grande dynamicité s'obtient au prix d'un affaiblissement du typage.

### b) Implémentations multiples

La classification et le polymorphisme permettent d'utiliser des objets dont il existe plusieurs implémentations, de façon transparente. Imaginons par exemple un logiciel qui dispose d'une fonction d'impression. Dans des environnements tels que Windows, l'utilisateur peut choisir à tout moment de changer de type d'imprimante. Comment le logiciel va-t-il faire pour savoir quels sont les bons codes à envoyer ? Il suffit de dire qu'il existe une classe des objets «imprimante», munie d'un certain nombre d'opérations : écrire une ligne, changer de jeu de caractères, etc. Au départ, on définit ceci comme :

```
package Classe_Imprimante is
   type Imprimante is abstract tagged null record;

   type Enrichissement is (Normal, Gras, Souligné);
   procedure Sélectionner (Effet: Enrichissement);

   type Alignement is (Gauche, Droit, Justifié, Centré);
   procedure Sélectionner (Justification : Alignement);
   -- etc...

   procedure Imprimer (Texte : String);
end Classe_Imprimante;
```

Ensuite, on pourra dériver des classes, selon les différents modèles d'imprimantes. Une procédure d'impression peut alors ignorer totalement sur quel périphérique physique elle s'effectue :

```
procedure Imprimer_Fichier (Nom : String;
                            Sur : Imprimante'Class) is
begin
   ...
   Sélectionner (Effet => Gras);
   Sélectionner (Justification => Justifié);
   Imprimer (Ligne);
   ...
end Imprimer_Fichier;
```

Du point de vue abstrait, on passe à la procédure un objet «abstraction d'imprimante», et la procédure lui demande d'effectuer les différentes opérations. Du point de vue du langage, la liaison dynamique assure la ventilation à l'exécution de la demande vers le sous-programme associé, qui n'a même pas besoin d'être connu au moment de la compilation de la procédure utilisatrice.

### c) Objets répartis

La conjonction du modèle d'exécution répartie (cf. paragraphe 3.7) et de la liaison dynamique permet de créer des *objets répartis*, c'est-à-dire situés sur des nœuds d'un réseau et appelables depuis d'autres nœuds. La liaison dynamique peut s'effectuer de façon transparente à travers le réseau sans que l'utilisateur ait à s'en préoccuper. Imaginons par exemple des serveurs de bandes magnétiques. Différents nœuds disposent de lecteurs, éventuellement de modèles différents. Nous représentons ainsi la classe des bandes magnétiques :

```
package Bandes is
   pragma Pure(Bandes);

   type Donnees is ...;

   type Instance is abstract tagged limited private;
   procedure Rembobiner (T : access Instance) is abstract;
   procedure Lire       (T      : access Instance;
                         Valeur : out    Donnees) is abstract;
   procedure Ecrire     (T      : access Instance;
                         Valeur : in     Donnees) is abstract;

private
   type Instance is abstract tagged limited null record;
end Bandes;
```

Nous définissons ensuite un «serveur de nom», c'est-à-dire un service permettant d'enregistrer sous un nom logique des pointeurs sur des objets `Tape`. Le paquetage ayant le pragma `Remote_Call_Interface`, le type pointeur qui y est défini est un type «étendu», c'est-à-dire qu'il contient l'information nécessaire pour retrouver l'objet sur le réseau :

```
with Bandes;
package Serveur_De_Nom is
  pragma Remote_Call_Interface;


  type Ptr_Bande is access all Bandes.Instance'Class;
  function  Trouver     (Nom  : String) return Ptr_Bande;
  procedure Enregistrer (Nom  : String; Bande : Ptr_Bande);
  procedure Retirer     (Bande: Ptr_Bande);
end Serveur_De_Nom;
```

Nous définissons ensuite les caractéristiques d'un certain modèle de dérouleur. Ce paquetage ne se trouvera physiquement que sur le seul nœud possédant le dérouleur considéré :

```
package Gestionnaire_Bande is
   pragma Elaborate_Body (Gestionnaire_Bande);
end Gestionnaire_Bande;

with Bandes, Serveur_De_Nom;
package body Gestionnaire_Bande is
  type Autre_Bande is new Bandes.Instance with ...
  procedure Rembobiner (T : access Autre_Bande) is ...
      -- Redéfinition de Rembobiner
  procedure Lire (T : access Autre_Bande; Valeur : out Donnees) is ...
      -- Redéfinition de Lire
  procedure Ecrire (T : access Autre_Bande; Valeur : in Donnees) is ...
      -- Redéfinition de Ecrire


  Bande1, Bande2 : aliased Autre_Bande;
begin
  Serveur_De_Nom.Enregistrer ("Bande 1", Bande1'Access);
  Serveur_De_Nom.Enregistrer ("Bande 2", Bande2'Access);

end Gestionnaire_Bande;
```

Remarquer que le corps du paquetage enregistre les objets `Tape1` et `Tape2` auprès du serveur de nom. Le client (qui n'a aucune raison de se trouver sur le même nœud que le dérouleur) récupérera *via* le serveur de nom un pointeur (distant) sur l'objet désiré. Tous les appels aux méthodes de cet objet seront donc routés à travers le réseau vers l'objet considéré :

```
with Bandes, Serveur_De_Nom;
procedure Client is
  B1, B2 : Serveur_De_Nom.Bande_Ptr;
  Tampon : Donnees;
  use Bandes;
begin
  B1 := Serveur_De_Nom.Trouver ("Bande 1");
  B2 := Serveur_De_Nom.Trouver ("Bande 2");
  Rembobiner (B1);
  Rembobiner (B2);
  Lire (B1, Tampon);
  Ecrire (B2, Tampon);
end Client;
```

Ce mécanisme est très efficace, car seule l'obtention d'un pointeur sur l'objet requiert un appel au serveur de nom ; l'appel de ses méthodes s'effectue directement à travers le réseau. Ce modèle est compatible avec le modèle CORBA[1] ; le mécanisme de communication entre les partitions peut utiliser le format IIOP, norme d'interopérabilité de CORBA. D'autre part, les schémas d'implémentation de CORBA avec Ada 95 ont été standardisés, et plusieurs fournisseurs offrent des outils de traduction d'IDL[2] vers Ada, aussi bien en «libre» qu'en propriétaire traditionnel.

### *8.3 .4 Exemple en classification*

Après avoir vu les mécanismes du langage, nous allons voir comment les utiliser au service de la méthode par classification. Supposons que nous voulions développer le système de paye d'une entreprise. Il existe de nombreuses sortes d'employés : certains disposent d'un salaire fixe, d'autres sont payés à la commission ; les charges sociales ne se calculent pas de la même façon pour un apprenti et un employé normal... Nous allons donc identifier ce qui est commun à tous les employés, puis modéliser chacune des modalités de paye dans des classes représentant directement chacun des types d'employés.

Première question : qu'*est-ce* qu'un employé, en général, indépendamment de toute spécialisation ultérieure ? C'est d'abord la représentation d'une personne, caractérisée par un nom, un numéro d'identification, un âge... En fait ce genre de caractéristiques n'est pas fondamental pour le problème courant ; nous pouvons nous contenter de repérer chaque employé par un numéro. Nous pourrons rajouter par la suite les autres caractéristiques, les spécialisations que nous aurons faites de l'employé en bénéficieront automatiquement grâce au mécanisme d'héritage.

Les opérations que doit fournir la classe des employés, pour notre vue, sont le calcul du salaire et celui des charges sociales. Ceux-ci sont toujours calculés par rapport à une certaine «base» : salaire de référence, catégorie, nombre de points... Ceci dépend de l'entreprise. Pour notre exemple, nous considérerons que cette base est donnée sous forme d'un montant en argent. Ensuite, chaque forme de salarié peut nécessiter l'adjonction d'une mention spéciale sur le bulletin de paye. Nous prévoyons donc une fonctionnalité à cet effet. Enfin, il n'est pas possible d'avoir un employé qui n'appartienne pas à quelque sous-catégorie plus raffinée ; l'employé doit donc être un type abstrait. Nous allons exprimer cette analyse dans le langage ; comme nous avons choisi de travailler ici en classification, nous adopterons la convention de nommer le paquetage avec le nom de l'entité, et d'appeler systématiquement le type associé `Instance`. Ainsi, un nom de la forme

[1] *Common Object Request Broker Architecture* : modèle de définition de serveur d'objets distribués, en cours de standardisation.

[2] *Interface Definition Language* : langage de spécification d'objets distribués, indépendant des langages de programmation, également en cours de standardisation.

`Employé.Instance` désignera clairement une instance d'employé. Par analogie, nous définirons systématiquement un nom pour le type à l'échelle de classe associé, que nous appellerons `Classe`. On trouvera la justification complète de cette convention dans [Ros95]. Ceci nous donne (nous supposons que le type `Argent` est défini dans le paquetage `Données_Globales`) :

```
with Données_Globales; use Données_Globales;
package Employé is
   type Numéro_Employé is range 1..10_000;
   type Instance is abstract tagged
     record
        Numéro    : Numéro_Employé;
        Référence : Argent;
     end record;
   subtype Classe is Instance'Class;

   function Salaire (De : Employé.Instance) return Argent
     is abstract;
   function Charges (De : Employé.Instance) return Argent
     is abstract;
   procedure Mentions_Spéciales (De : Employé.Instance);
end Employé;
```

Remarquer que la procédure `Mentions_Spéciales` n'est *pas* abstraite : nous fournirons simplement une implémentation qui ne fait rien. Ainsi, tous les types dérivés ultérieurs qui n'ont pas à imprimer de mentions spéciales pourront conserver la procédure par défaut dont ils auront hérité.

A partir de là, il existe deux sortes d'employés : les salariés (dont le salaire est fixe) et les commissionnés, dont le salaire dépend d'une commission, ou d'une autre forme de rétribution liée à leurs performances. Les salariés n'ont besoin *a priori* d'aucune fonctionnalité supplémentaire : nous allons les dériver directement des employés :

```
with Données_Globales; use Données_Globales;
with Employé;
package Salarié is
   type Instance is new Employé.Instance with null record;
   subtype Classe is Instance'Class;

   function Salaire (De : Salarié.Instance) return Argent;
   function Charges (De : Salarié.Instance) return Argent;
   procedure Mentions_Spéciales (De : Salarié.Instance);
end Salarié;
```

Nous obtenons une classe non abstraite en la dérivant de la classe abstraite et en fournissant la définition des sous-programmes hérités qui étaient déclarés abstraits. Avec notre convention, les utilisateurs pourront déclarer des objets ou des procédures comme :

```
J_P_Rosen : Salarié.Instance;
procedure Emettre_Bulletin_Paye (Pour : Employé.Classe);
```

Remarquer que nous utilisons toujours les types sous la forme de noms complets, comme `Salarié.Instance` ou `Employé.Classe`, exprimant clairement la différence entre les entités portant sur une instance particulière et celles applicables à toute une classe. Ceci correspond encore à une sorte de documentation, vérifiée par le langage, de ce que nous voulons exprimer.

Puisque nous ne rajoutons rien pour faire les `Salarié`, ne pourrait-on supprimer `Employé` et mettre `Salarié` comme racine de l'arbre de dérivation ? Du point de vue du langage, c'est tout à fait faisable. Mais d'une part, notre analyse a identifié que les salariés et les commissionnés étaient deux formes d'employés : il faut traduire cette structure dans le langage. Ensuite, rien ne dit que, par la suite, nous ne serons pas amenés à rajouter aux salariés des éléments qui n'ont aucun sens pour les commissionnés ; de même peut-être voudrons-nous écrire des procédures applicables à tous les salariés, mais non aux commissionnés (en termes de langage, une procédure avec un paramètre de type `Salarié.Classe`). Si nous faisions de `Salarié` la classe racine, ces éléments seraient également transmis aux `Commissionné`.

Les commissionnés sont payés en fonction d'un (petit) fixe, correspondant au salaire de référence, et de leur chiffre d'affaire. En plus des paramètres communs à tout employé, un commissionné doit posséder un chiffre d'affaire courant. Ce qui nous donne :

```
with Données_Globales; use Données_Globales;
with Employé;
package Commissionné is
   type Instance is new Employé.Instance with
     record
       Chiffre_d_Affaire : Argent := 0.0;
     end record;
   subtype Classe is Instance'Class;

   function Salaire (De : Commissionné.Instance)
     return Argent;
   function Charges (De : Commissionné.Instance)
     return Argent;
   procedure Mentions_Spéciales(De: Commissionné.Instance);
end Commissionné;
```

Tout employé peut cotiser volontairement à une caisse de retraite ; cette cotisation est prise sur son salaire, mais est déductible des charges sociales[1]. Il accumule des cotisations, mais peut choisir à tout moment le montant à cotiser et interroger l'état de son compte. Il faut donc pouvoir rajouter une facette «Cotisant» à tout employé (mais pas à un type qui n'appartiendrait pas à la classe des employés). Nous traduirons ceci comme :

```
with Données_Globales; use Données_Globales;
with Employé;
generic
   type Origine is new Employé.Instance with private;
package Cotisant is
   type    Instance is new Origine with private;
   subtype Classe   is Instance'Class;

   function Salaire (De : Cotisant.Instance) return Argent;
   function Charges (De : Cotisant.Instance) return Argent;
   procedure Mentions_Spéciales (De : Cotisant.Instance);
   procedure Changer_Cotisation
     (De         : in out Cotisant.Instance;
      Cotisation : in     Argent);
   function Montant_Cotisé (Par : Cotisant.Instance)
     return Argent;

private
   type Instance is new Origine with
     record
       Cotisation_mensuelle : Argent := 0.0;
       Cotisation_Accumulée : Argent := 0.0;
     end record;
end Cotisant;
```

A partir de là, il est facile de définir la classe des commissionnés cotisants :

```
with Commissionné, Cotisant;
package Commissionné_Cotisant is
   new Cotisant (Commissionné.Instance);
```

Grâce à notre convention de nommage, nous pouvons parler de `Commissionné_Cotisant.Instance` ou de `Commissionné_Cotisant.Classe` de la même façon que pour les types qui n'ont pas été obtenus par des génériques. Bien entendu, il faut déduire le montant de la cotisation du montant du salaire ; ceci se fait facilement en redéfinissant le calcul du salaire d'un cotisant comme ceci :

```
function Salaire (De: Cotisant.Instance) return Argent is
begin
   return Salaire (Origine (De)) - Montant_Cotisé (De);
end Salaire;
```

[1] Si un législateur social lit ces lignes, qu'il veuille bien pardonner les simplifications et inexactitudes de ce modèle....

Ceci exprime bien que le salaire d'un `Cotisant` est son salaire d'origine diminué du montant cotisé. De même, si notre `Cotisant` veut rajouter une mention spéciale, celle-ci ne doit pas remplacer, mais s'ajouter aux mentions spéciales que le type d'origine pourrait avoir. Il faut donc appeler la procédure d'origine avant la nôtre propre :

```
procedure Mentions_Spéciales (De : Cotisant.Instance) is
begin
   Mentions_Spéciales( Origine (De));
   Put ("Cotisation : ");
   Put (Montant_Cotisé (Cotisant.Classe (De)));
end Mentions_Spéciales;
```

Remarquer que nous appelons le montant cotisé en convertissant le paramètre vers la *classe* pour forcer une liaison dynamique : si un type dérivé redéfinit les modalités de calcul des cotisations, on appellera cette fonction, et non celle définie pour `Cotisant.Instance`. Il ne sera pas nécessaire de redéfinir la procédure `Mentions_Spéciales` : la procédure ci-dessus a déjà été prévue pour fonctionner avec les types qui en hériteront plus tard.

Nous pourrions continuer longtemps ainsi ; chaque fois qu'une nouvelle forme d'employé apparaît, il s'agit généralement (légalement) d'une subdivision d'une classification d'employé existante : il nous suffit de refléter dans notre programme la hiérarchie exacte définie par les textes légaux, conventions collectives, etc. Lorsqu'une variation s'applique de façon orthogonale à la hiérarchie, nous la rajoutons sous la forme d'une facette, comme pour le cas de `Cotisant` ci-dessus.

Une fois définie notre hiérarchie de types, il nous reste à l'utiliser. Ceci se fait en général dans un contexte plus procédural. Si l'on veut éditer un bulletin de paye, il serait ridicule de définir une classe «bulletin»... qui n'aurait aucune sous-classe[1]. Une telle procédure pourrait avoir le schéma général suivant :

```
procedure Editer_Bulletin (De: Employé.Classe) is
begin
   ...
   Put ("Salaire : "); Put (Salaire (De)); New_Line;
   Put ("Charges : "); Put (Charges (De)); New_Line;
   Mentions_Spéciales (De);
   ...
end Editer_Bulletin;
```

Cette procédure peut traiter n'importe quel employé, et le mécanisme de la liaison dynamique garantit que le calcul du salaire, des charges, ainsi que les mentions spéciales imprimées seront bien ceux correspondant à l'instance utilisée pour l'appel.

## *8.3 .5 La classification en Ada et dans d'autres langages*

L'héritage et les autres services nécessaires à la classification n'étant apparus en Ada qu'en 1995, celui-ci a naturellement tiré les leçons de ses prédécesseurs et adopté des mécanismes cohérents avec le reste de sa philosophie, mais qui diffèrent parfois de ceux d'autres langages orientés objet. Nous allons maintenant comparer la façon de réaliser les concepts de la classification en Ada avec celle d'autres langages, et montrer les raisons des solutions adoptées. Nous utiliserons pour cette comparaison principalement Java et C++, parce que ce sont les langages orientés objet les plus populaires actuellement. Nous ne parlerons pas de SmallTalk, car son absence totale de typage (qui est un atout pour le maquettage) l'éloigne des domaines où l'on utilise Ada. Quant à C#, il n'est pas suffisamment différent de Java sur le plan du principe pour qu'il soit utile d'en parler spécifiquement.

[1] Ceci ne signifie pas que tous les bulletins sont semblables, mais que les différences proviennent des employés, non des bulletins eux-mêmes.

### a) Classes et modularisation

La plupart des LOO disposent d'une construction syntaxique spéciale pour la notion de *classe*, qui regroupe dans une même déclaration les attributs et les méthodes de la classe. Ada ne possède pas de telle construction : une classe est réalisée par un paquetage contenant la déclaration d'un type étiqueté et des opérations associées. Une classe est donc une sorte de patron de conception (design pattern), ce qui la rend moins visible en tant que telles que dans les autres LOO. Ceci provient du choix d'Ada de fournir des «blocs de base» (*building blocks*) permettant de réaliser les structures nécessaires à *toutes* les méthodologies.

Inversement, en Java, la classe est la seule unité de structuration, ce qui oblige à *tout* définir sous forme de classes. Chaque classe appartient logiquement à un paquetage qui en limite la visibilité: ainsi, seules les classes dites «publiques» sont visibles d'un paquetage à un autre. Toutefois, cette notion de paquetage ne correspond à aucune structuration physique en unités de compilation; il n'est pas possible, par exemple, de déterminer simplement l'ensemble des classes appartenant à un paquetage. Ce n'est donc pas une aide à la modularisation.

C++ occupe une position intermédiaire; le fichier source a un effet de visiblité locale, et il est possible de déclarer dans un même fichier plusieurs classes, et même des fonctions non nécessairement reliées à une classe. ceci permet un certain regroupement des classes logiquement reliées, mais sans faire apparaître clairement la notion de module au niveau du langage. La classe reste le principal moyen de définir des types évolués; on voit alors apparaître des «fausses classes» qui ne servent qu'à encapsuler des éléments, sans aucune relation avec la classification en tant que méthode[1]. Quant aux namespace, ils permettent de contrôler la visibilité des identificateurs, mais n'offrent pas plus de structuration logique que les paquetages Java.

La solution des blocs de base adoptée par Ada présente un certain nombre d'avantages, au moins dans le cadre d'un langage généraliste :

> Les notions d'encapsulation et de typage sont orthogonales. De plus, le mécanisme utilisé pour le support de l'héritage (les types étiquetés) est indépendant du mécanisme utilisé pour la modularisation (les paquetages).
>
> Certains utilisateurs peuvent ne vouloir que certaines fonctionnalités ; par exemple, on peut utiliser l'extension de types sans pour autant avoir besoin de la liaison dynamique. L'utilisateur ne paie alors le prix que des fonctionnalités qu'il utilise.
>
> Il n'existe aucun problème pour imbriquer les déclarations. Un paquetage définissant une classe peut définir un autre paquetage de classe imbriqué ; une procédure (qu'elle soit ou non une «méthode» d'un type étiqueté) peut contenir ses propres classes locales, etc. Sans entrer dans les détails techniques, noter qu'*aucun* autre langage orienté objet n'offre cette possibilité, car elle complique notablement l'écriture du compilateur.

### b) Instances et sémantique de référence

Certains langages orientés objet, dans un but d'unification des concepts, ne font pas de différence entre classes et instances ; une classe est alors une instance d'une *métaclasse*, ou classe de classes. Cette approche est mathématiquement et intellectuellement très pure, mais ne conduit pas nécessairement à une meilleure lisibilité ni à une plus grande maintenabilité des programmes...

Dans la plupart des langages à objets, une variable déclarée avec un type peut contenir non seulement des valeurs de son type, mais également tout objet correspondant à sa *classe*. Autrement dit, on ne fait pas la différence entre ce que l'on appelle en Ada un type spécifique et un type à l'échelle de classe. En Java par exemple, une variable déclarée comme `Objet_Graphique` pourrait contenir n'importe quel objet graphique, aussi bien un `Cercle` qu'un `Rectangle`. Ce n'est possible que parce qu'au niveau de l'implémentation, les variables ne sont que des pointeurs vers les instances. Un objet ne peut exister simplement parce qu'il est déclaré : il faut une opération de

[1] On entend ainsi parfois parler de «classes fonctionnelles» : le nom même apparaît comme un désaveu de la méthode.

création explicite. Malheureusement, ceci implique que *toutes* les classes sont à sémantique de référence : il n'est pas possible de créer de types abstraits à sémantique de valeur. Une telle solution, si elle peut être commode pour des conceptions fondées uniquement sur le mécanisme d'héritage et le polymorphisme, aurait été inacceptable en Ada, qui doit pouvoir soutenir toutes les méthodologies.

Il est bien entendu possible d'obtenir en Ada le même comportement qu'avec les autres langages orientés objet, en travaillant uniquement avec des pointeurs sur classe. Mais cela n'est pas nécessaire pour bénéficier de la liaison dynamique ni des autres mécanismes de la POO. Il n'y a donc pas de pointeurs cachés qui interdiraient quasiment l'utilisation de ces mécanismes dans des contextes «temps réel».

### c) Variables de classe

Certains LOO autorisent la définition de *variables de classe*, par opposition aux attributs normaux appelés *variables d'instance*. Une variable de classe est, comme son nom l'indique, liée à une classe, et donc partagée par toutes les instances appartenant à la classe. Ceci correspond à la notion de *membre statique* en Java et en C++. Imaginons par exemple que nous voulions assigner un numéro différent à chaque nouvel objet créé : il faut bien avoir une variable globale servant de compteur d'objet, qui doit être accessible depuis toutes les instances. Et comme nous ne souhaitons pas autoriser un accès indiscipliné à cette variable à l'extérieur de la classe, il faut une variable privée, non pas à chaque instance, mais à la classe. Ainsi formulée, cette notion paraît quelque peu bizarre. Sa réalisation en Ada est beaucoup plus naturelle : il s'agit simplement d'une variable globale du corps de paquetage :

```
package Classe_Objets is
   type Objet is tagged
       record
           ...
       end record; — Opérations sur Objet
end Classe_objet;

package body Classe_Objets is
   type T_Compteur is range 0..1000;
   Compteur : T_Compteur := 0;

   — Implémentation des opérations
end Classe_Objets;
```

### d) Association des méthodes aux objets

Dans la plupart des langages orientés objet, une méthode est «propriété» d'un objet, et l'appel d'une méthode utilise une syntaxe particulière. Si un objet `O` appartient à la classe `C` et qu'on veut appeler sa méthode `M`, on écrira :

```
O.M;
```

En Ada, une méthode est simplement une procédure qui possède un paramètre du type `C`, et l'appel s'écrit :

```
M (O);
```

Apparemment, la solution des LOO correspond mieux à la notion de «demander à l'objet O d'exécuter sa méthode M». Cependant, le problème se complique si la méthode doit porter sur plusieurs paramètres de la même classe ; l'écriture «LOO» rompt la symétrie :

```
O1.M(O2);
```

alors que celle-ci est conservée en Ada :

```
M(O1, O2);
```

Si nous définissons une sous-classe `S` de `C` (en termes Ada, nous dérivons le type `S` du type étiqueté `C`), il n'y a avec les LOO classiques qu'un seul terme principal qui détermine la méthode appelée. Par conséquent, la méthode dont héritera un objet de classe dérivée `D` sera une méthode dont le paramètre sera de classe `C` ; en Ada, *tous* les paramètres sont dérivés, ce qui signifie que la procédure `M` dérivée portera bien sur deux paramètres de type `D`. Supposons par exemple que nous redéfinissions la comparaison d'égalité :

```
type C is tagged record ...;
function "=" (O1, O2 : C) return Boolean;

type D is new C with ...;
-- on hérite de :
-- function "=" (O1, O2 : D) return Boolean;
```

Dans le cas des autres LOO, on hériterait d'une fonction de comparaison d'un `C` avec un `D`, c'est-à-dire de quelque chose qui n'aurait aucun sens ! La solution Ada est donc moins «pure» du point de vue de la théorie des LOO, mais elle est plus sûre et fournit un comportement plus conforme aux attentes de l'utilisateur dès qu'une méthode doit porter sur plus d'un seul objet.

Notons que la syntaxe Ada permet de disposer automatiquement d'un nom (celui du paramètre) pour désigner l'objet sur lequel porte la fonction, éliminant le besoin de constructions spéciales (**`current`** en Eiffel ou **`this`** en Java et C++).

> Ce point syntaxique pourrait paraître mineur, mais les utilisateurs d'autres langages sont tellement habitués à la notation préfixée que la notation Ada représente un obstacle sérieux à la compréhension du modèle Ada de programmation orientée objet. Et dans les cas simples, la notation préfixée est effectivement plus naturelle. Pour ces raisons, Ada 0Y incluera une notation alternative, où O.M(P) sera équivalent à M(O,P). Il s'agit d'une simple équivalence syntaxique sans changement sur le principe.

Enfin, la notation Ada autorise la liaison dynamique sur le type retourné par une fonction. Etant données les définitions suivantes:

```
type T is tagged record ...;
function F return T;

type D is new T with ...;
function F return D;
```

il est possible de faire un appel dynamique à la fonction F, où le contexte d'appel déterminera, possiblement dynamiquement, quelle fonction est appelée. Cette possibilité n'existe ni en Eiffel, ni en C++, ni en Java....

### e) Liaison dynamique

La liaison dynamique est une propriété fondamentale des langages orientés objet. En Java, elle est systématique : c'est toujours l'instance qui détermine la méthode effectivement appelée. Il est possible cependant de caster un objet dans un type ancêtre (parent direct ou type ancêtre plus éloigné) pour forcer l'appel d'une méthode ancêtre redéfinie.

En C++, les méthodes «normales» n'effectuent jamais de liaison dynamique ; seules celles définies avec le mot clé **`virtual`** peuvent le faire. Il est possible de forcer un appel statique vers un ancêtre, mais en l'absence d'une telle spécification explicite on ne peut pas savoir, sans regarder la définition de la méthode, si l'appel est statique ou dynamique.

Le mécanisme adopté par Ada a l'avantage de toujours laisser le choix *à l'appelant* d'effectuer un appel spécifique ou dynamique. Ceci oblige le programmeur à réfléchir précisément aux propriétés souhaitées, mais permet de connaître précisément, à la lecture du programme, les conditions d'appel.

### f) Types abstraits

Cette notion est présente de façon extrêmement voisine en Java. Comme en Ada, seules les classes abstraites peuvent posséder des méthodes abstraites.

En C++, il est possible de définir des méthodes abstraites (appellées vrituelles pures) en affectant la valeur 0 à la fonction (!!). La classe correspondante devient abstraite sans que cela apparaisse syntaxiquement dans la définition de la classe. En particulier, il n'est pas possible de définir une classe abstraite ne possédant aucune méthode abstraite.

### g) Approches de l'héritage multiple

Nous avons vu qu'en Ada, une approche par composition de générique permet de voir un même objet selon plusieurs facettes différentes. D'autres langages utilisent pour cela la notion d'héritage multiple, qui correspondrait en Ada au fait de pouvoir dériver de plusieurs types à la fois.

Si l'héritage multiple est séduisant en théorie, il apporte avec lui un problème immédiat : lorsque l'on cherche où se trouve une méthode appliquée à une instance, il faut remonter plusieurs chemins (puisque plusieurs classes ancêtres sont autorisées). Que faire si plusieurs méthodes portant le même nom sont accessibles par des chemins différents ?

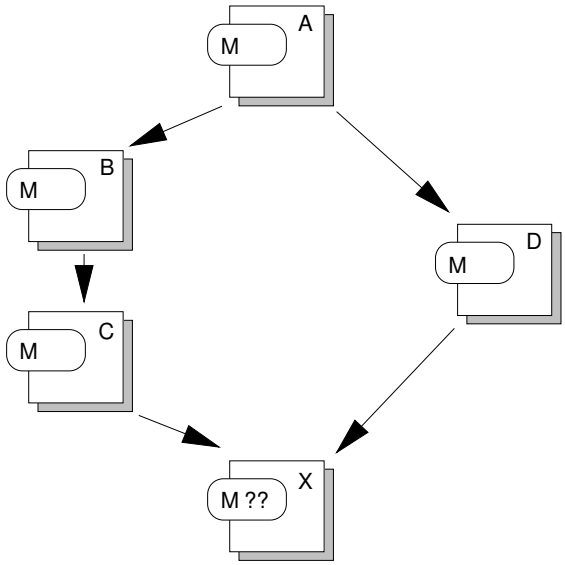


**Figure 18 :** Exemple de graphe d'héritage multiple

Considérons le graphe de la figure 18. L'objet X hérite d'une méthode M définie dans C aussi bien que dans D. Mais celle de C est héritée de B, où il s'agit d'une redéfinition de celle de A, alors que celle de D est une redéfinition directe de celle de A. Laquelle prendre ? Celle de A qui est commune à tout le monde ? Celle de D qui est plus directe ? Celle de C (donc de B) parce que c'est la plus à gauche[1] ?

Ce problème est connu dans la littérature sous le nom d'héritage *répété*. Il est très important dans le domaine de l'intelligence artificielle lorsqu'il s'agit de modéliser des connaissances, car il faut alors trouver les relations conceptuelles exactes qui lient les différents objets. Dans le domaine qui nous intéresse, celui du génie logiciel, il s'agira de coïncidences de noms généralement fortuites, et ce problème sera résolu par des méthodes autoritaires. En Eiffel par exemple, l'utilisateur doit manuellement lever toute ambiguïté (en fournissant d'autres noms, non ambigus, aux méthodes héritées homonymes).

La solution choisie par Ada au problème des vues multiples permet de s'affranchir de ce problème : l'enrichissement des classes se faisant par ajouts successifs, les nouvelles méthodes remplacent naturellement les anciennes. C'est l'ordre d'instanciation (choisi par l'utilisateur) qui détermine la méthode effectivement utilisée. De plus, cette solution procure une grande finesse dans la définition des relations entre objets. Si un générique de facette annonce :

```
generic
   type Origine is abstract tagged limited private;
package Facette is ...
```

[1] Ce critère de choix ne paraîtra stupide qu'à ceux qui n'ont jamais eu à écrire un compilateur...

alors il pourra être utilisé avec n'importe quel type étiqueté. Mais s'il annonce :

```
generic
   type Origine is new Objet_Géométrique with private;
package Facette is ...
```

alors, il ne pourra être instancié que sur un descendant (direct ou indirect) du type `Objet_Géométrique`. Autrement dit, on peut créer ainsi des facettes qui ne peuvent servir qu'à enrichir certaines classes bien précises (mais qui du coup peuvent tirer parti de la connaissance supplémentaire des propriétés du type ancêtre). En héritage multiple classique, cela reviendrait à créer une classe dont on ne pourrait hériter que si l'on héritait simultanément d'une autre classe ; un tel type de contrôle serait très difficile à définir.

### h) L'héritage d'interfaces

La notion d'interface est un concept introduit par Java (et repris par C#), principalement dans le but de fournir les services de l'héritage multiple tout en paliant ses inconvénients. Afin de bien comprendre de quoi il s'agit, nous allons revenir quelque peu sur le mécanisme d'héritage.

Dans la présentation que nous en avons faite ci-dessus, nous avons classiquement expliqué qu'une classe parent fournit des attributs et des méthodes, et que l'on peut en dériver une classe enfant, possédant les mêmes propriétés et méthodes (au moins), puisqu'elle en hérite. De plus, si le comportement de certaines méthodes héritées ne conviennent pas à la classe enfant,, celle-ci peut les redéfinir, c'est-à-dire fournir un comportement différent pour la même méthode. Au passage, on remarque que toute méthode est soit héritée, soit redéfinie, mais ne peut disparaître. Par conséquent, tous les descendants d'une classe ont (au moins) les mêmes méthodes que leurs ancêtres. Cette façon de présenter l'héritage correspond à l'approche dite programmation delta: on voit une classe enfant comme une légère modification du comportement d'une classe existante.

On peut cependant présenter l'héritage d'une manière légèrement différente. Une classe parente définit un ensemble de propriétés partagées par tous ses descendants. Par exemple, tous les animaux mangent; par conséquent, la classe de tous les animaux fournit la méthode «Manger». Ensuite, chaque forme particulière d'animal définit comment elle mange en fournissant sa propre définition de «Manger». Il peut arriver cependant que toutes les sous-classes d'une classe donnée partagent une même façon de faire; par exemple, toutes les variété de chiens mangent de la même façon. Il est alors commode que la classe «Chien» fournisse une implémentation par défaut de «Manger», qui sera partagée par toutes ses sous-classes, à moins d'être explicitement redéfinie.

Dans cette seconde présentation, nous mettons l'accent sur la présence d'une interface commune, dont la présence est garantie chez tous les descendants d'une classe; l'héritage des méthodes n'a qu'un rôle secondaire, une sorte de valeur par défaut qui peut, mais pas nécessairement, être fournie par un ancètre. D'ailleurs, la notion de méthode abstraite vue précédemment correspond exactement à celà: une méthode pour laquelle l'ancètre ne fournit pas de valeur par défaut, et qui doit donc être redéfinie par les descendants. De ces deux aspects de l'héritage, il apparaît clairement que le plus important est la notion d'interface garantie; c'est ce qui permet le polymorphisme, la possibilité qu'une variable contienne des objets dont le type n'est pas connu statiquement, tout en autorisant de lui appliquer les méthodes appropriées.

Une interface en Java est simplement une forme d'héritage réduit à son premier aspect : la définition d'un ensemble de méthodes fournies, sans valeurs par défaut[1]. Une classe qui implémente l'interface promet de fournir les méthodes définies par l'interface. Un exemple simple d'interface est donné dans l'exemple suivant :

```
public interface Enumeration { // Dans le paquetage java.util
   public boolean hasMoreElements ();
   public Object nextElement () throws NoSuchElementException;
}
```

[1] Une interface ne peut pas non plus définir d'attributs.

Il est alors possible de définir une méthode qui s'applique à toute classe qui implémente l'interface Enumeration :

```
//Cette méthode imprime tous les éléments d'une structure de données :
void listAll (Enumeration e) {
   while e.hasMoreElements ()
       System.out.println (e.nextElement());
}
```

Si une classe Java n'a le droit d'hériter que d'une seule autre classe, elle peut déclarer implémenter autant d'interfaces qu'elle le souhaite; ceci répond à la plupart des besoins pour lesquels l'héritage multiple peut être utile, mais sans les problèmes dus à l'héritage répété, puisque la méthode doit être explicitement redéfinie.

D'une certaine façon, on peut dire qu'une interface n'est rien d'autre qu'une classe abstraite qui ne définit aucun attribut, et dont toutes les méthodes sont abstraites. Mais on peut même considérer que la notion d'interface n'a rien à voir avec l'héritage, et constitue même en un sens le contraire de l'héritage : ce n'est qu'un contrat qui peut être respecté par différentes classes, sans qu'il soit nécessaire d'avoir aucune relation conceptuelle entre les différentes classes qui implémentent l'interface. En bref, l'héritage correspond à une relation «est un», alors que les interfaces correspondent à une relation «fournit».

En plus de cette relation de base, une interface peut en étendre une autre; ceci introduit alors une vraie relation d'héritage entre les interfaces; mais il n'en reste pas moins que la notion d'implémentation d'interface n'est pas en elle-même une relation d'héritage.

Il est possible d'obtenir en Ada l'équivalent du mécanisme d'héritage d'interface au moyen d'un patron de conception particulier [Ros02]. La méthode utilisée est cependant un peu complexe à mettre en oeuvre, compte tenu de la popularité croîssante de la notion d'interface. Il est donc prévu qu'Ada 2005 fournisse un mécanisme d'héritage d'interface, très proche dans son principe de ce qui est fourni par Java.

### *8.3 .6 Conclusion*

Les types étiquetés d'Ada fournissent l'outil de base pour les méthodes par classification. Les objets à facettes multiples sont réalisés par un moyen original, l'utilisation de la généricité, plutôt que par un mécanisme spécial du langage comme l'héritage multiple. Cette différence d'approche par rapport aux autres langages permet une meilleure sécurité et un contrôle plus précis du typage et des relations entre classes, au prix d'une plus grande lourdeur d'écriture. Ceci correspond à la philosophie du langage, qui privilégie la facilité de maintenance et la fiabilité par rapport à la facilité de conception initiale. Nous nous permettrons d'insister sur ce point, car la question de l'héritage multiple fait l'objet d'un vif débat jusque dans la communauté POO traditionnelle, certains y voyant la panacée, d'autres un danger permanent. Il est certain que si l'on demande s'il est possible, en Ada 95, de dériver un type directement de plusieurs autres à la fois, la réponse est «non». Mais en fait, l'héritage multiple, comme tout autre outil fourni par un langage d'ailleurs, n'est qu'un *moyen* au service de *besoins*. Si l'on demande si Ada 95 répond aux besoins que satisfait l'héritage multiple dans d'autres langages, la réponse est «oui», par ses moyens propres, qui conservent au langage ses points forts que sont la sécurité, la portabilité, l'efficacité et l'indépendance vis-à-vis des représentations machine.

Une autre originalité d'Ada pour la classification est que le mécanisme du langage ne repose absolument pas sur l'utilisation de pointeurs cachés ou non. L'utilisateur reste libre de définir des types à sémantique de valeur aussi bien qu'à sémantique de référence. De plus, de nouveaux outils tels que les autopointeurs et les pointeurs généralisés permettent de définir des relations subtiles entre classes. Il est possible de définir des objets actifs, et même des objets distribués. On peut donc dire non seulement qu'Ada supporte entièrement les méthodes par classification, mais aussi que les

nouveaux outils qu'il apporte sont de nature à faire progresser les méthodes par classification elles-mêmes.

## 8.4 Classification ou composition ?

L'approche objet conduit à une réorganisation complète de la structure des programmes ; elle aura donc une profonde influence sur toutes les étapes du cycle de vie du logiciel. Un certain nombre de ses avantages proviennent de l'approche objet elle-même, que celle-ci soit ensuite organisée par composition ou par classification[1]. Nous allons maintenant essayer de dégager les points communs aux deux approches, et aussi ce qui les différencie.

### *8.4 .1 Avantages communs aux deux approches*

Les méthodes orientées objet conduisent à des programmes plus lisibles et plus compréhensibles : les objets concrets constituent finalement ce que le programmeur connaît le mieux. Une approche «objet» sera plus proche du domaine du monde réel, donc plus facile à appréhender, que les approches fonctionnelles. Les partisans de la classification comme ceux de la composition comparent d'ailleurs toujours leurs méthodes à l'approche fonctionnelle, car c'est par rapport à elle que le bénéfice est le plus grand.

L'objet inclut toutes les valeurs et opérations nécessaires et suffisantes pour le caractériser ; il constitue donc une unité à forte cohésion et faible couplage. Pour déterminer les caractéristiques d'un objet, le programmeur est guidé par les caractéristiques «absolues» de l'objet réel que modélise l'objet informatique, et non par les besoins d'une application particulière ; ces caractéristiques sont donc indépendantes de l'utilisation qui en est faite dans le cadre d'un projet informatique particulier. On obtient ainsi des unités facilement *réutilisables*. Tout nouveau programme partageant avec un programme précédent certaines notions du monde réel pourra réutiliser les modules déjà développés. Nous discuterons cependant de ce point de façon plus approfondie par la suite, car les deux approches ne favorisent pas le même genre de réutilisabilité.

L'objet regroupe les différents aspects d'une abstraction ; cette *encapsulation* est caractéristique de toutes les démarches «objet». On obtient ainsi une meilleure *localisation*, c'est-à-dire que les endroits où il est nécessaire d'intervenir en cas d'évolution du logiciel ou de maintenance sont beaucoup plus faciles à identifier.

Enfin, il n'y a aucune difficulté à utiliser la méthode pour définir des systèmes parallèles : c'est le comportement naturel de la plupart des objets du monde réel ; lorsque vous mettez le gaz sous votre autocuiseur, vous allez faire autre chose et ne vous en occupez plus jusqu'au moment où il siffle pour indiquer qu'il a atteint la température requise. Le parallélisme est une conséquence naturelle de cette approche[2], ce qui rend les méthodes orientées objet particulièrement appropriées à la conception de systèmes parallèles. On dispose ainsi d'une méthode unique pour les systèmes séquentiels et parallèles.

### *8.4 .2 Réutilisabilité*

La réutilisabilité est un peu le monstre du Loch Ness du génie logiciel : tout le monde en parle, mais bien peu la mettent en pratique... Encore faut-il savoir ce que l'on entend par réutilisabilité. On peut dire que l'approche objet, de manière générale, promeut la réutilisabilité, et c'est un argument

[1] Ce qui n'empêche pas les partisans de chacune des méthodes de s'en attribuer le mérite exclusif.

[2] Nous pensons qu'un vrai langage orienté objet se *doit* de permettre le parallélisme. Ce critère n'est pas retenu par la plupart des apôtres de la programmation orientée objet, car alors ni C++, ni Eiffel, ni la plupart des autres LOO ne pourraient obtenir le label magique... Cette idée fait cependant progressivement son chemin, puisque Java et C# offrent des possibilités de programmation concurrente.

utilisé par les partisans de la composition comme de la classification. La raison en est simple : les modules correspondant de façon bijective aux objets du monde réel, le fait qu'un nouveau logiciel manie les mêmes objets réels que ceux d'un logiciel existant est une condition *suffisante* pour permettre la réutilisation des modules correspondants.

Mais du fait de leurs différences, les deux approches conduisent à des vues sensiblement différentes de la réutilisabilité. L'approche par classification favorise le développement incrémental. Il est facile, à partir d'une conception initiale, de dériver de nouvelles versions comportant des propriétés additionnelles sans pour autant affecter les utilisateurs de l'ancienne version : il suffit de faire une nouvelle classe dérivée de l'ancienne, à laquelle on ajoute les nouvelles fonctionnalités. Si certains aspects de l'ancienne version ne conviennent plus, il suffira de les redéfinir, et seulement eux. On peut donc aisément construire un nouvel objet à partir du code développé pour un objet existant. Il n'est pas nécessaire de concevoir spécifiquement dans un but de réutilisabilité : il s'agit d'une réutilisabilité *a posteriori*, qui permet de développer des objets «sur mesure» sans avoir à reconcevoir leurs aspects standard ou déjà développés. Le prix à payer pour cette facilité est qu'elle risque de conduire à une prolifération de versions ou de variations à partir d'une base donnée, rendant la maintenance difficile, si ce n'est parfois périlleuse, car une modification dans un objet de base sera propagée à un nombre inconnu d'objets dérivés. Si l'on a développé par exemple un objet `Menu`, il est possible que chacun des *N* programmeurs d'une équipe décide d'ajuster le `Menu` à ses désirs propres, conduisant à la nécessité d'en maintenir *N*+1 versions !

En approche par composition, on construit des composants logiciels, par analogie avec les composants matériels, dont le but est d'être utilisés comme briques de base dans le développement de logiciels complets. Ceci implique les mêmes contraintes que pour les composants matériels : leur conception doit être soignée, car une modification de leur spécification peut être coûteuse. Ce sera au programmeur de s'adapter aux composants disponibles, au lieu de les adapter à ses besoins propres. La qualité du projet sera fortement conditionnée par la qualité des composants, et ceux-ci devront être conçus dès le début dans un but de réutilisabilité : c'est une vue *a priori* de la réutilisabilité. En revanche, une fois l'interface bien établie, la maintenance peut corriger, modifier ou adapter l'implémentation en ayant la garantie que cela ne peut avoir aucune conséquence néfaste sur les utilisateurs du composant. L'étanchéité entre spécification et implémentation est totale.

Il pourrait apparaître que cette approche nuit à l'évolutivité des programmes ; il n'en est rien, et le mieux pour s'en convaincre est de comparer avec les composants matériels. Depuis sa création, la spécification (le brochage) du 7400[1] n'a pas bougé ! En revanche, son implémentation (sur silicium) a pu changer, et varie même d'un fabricant à l'autre sans que l'utilisateur s'en préoccupe. Des produits extrêmement différents ont pu être ainsi créés en réutilisant les mêmes composants de base.

Notons pour clore cette discussion sur la réutilisabilité que les deux approches ont un comportement dynamique opposé. L'héritage tend à produire des composants de plus en plus spécialisés (donc de moins en moins réutilisables) au fil du temps et des dérivations. En revanche, la démarche adoptée en composition va consister à construire des composants spécifiques, puis à les généraliser, par exemple en les rendant génériques. Au fil du temps, les modules tendent donc à devenir de plus en plus généraux et réutilisables. On peut dire que la démarche par classification va du général au particulier, alors que la démarche par composition va du particulier vers le général.

## *8.4 .3 Structuration*

L'approche objet met en relief l'importance de la structuration globale du projet, que nous avons appelée la «topologie de programme». Mais cet aspect est lié à la méthode choisie. On ne sera donc pas étonné de trouver des structures totalement opposées lorsque l'on change de méthode. Comme les différents critères de décomposition sont ce qui différencie les méthodes, des éléments de «haut

[1] Note à l'intention des non-électroniciens : il s'agit d'un circuit intégré comportant quatre portes NAND dans un boîtier – vraisemblablement le circuit le plus utilisé dans les montages logiques.

niveau» pour une méthode seront considérés comme de «bas niveau» pour une autre, et inversement. Un changement de méthode aboutit à un retournement de la structure même du projet, comme une chaussette que l'on retourne. *Il n'empêche que les instructions de base se retrouvent quasiment identiques quelle que soit la méthode employée.* Ce qui change, c'est la façon dont elles sont organisées, regroupées entre elles.

Nous discutions ainsi avec des ingénieurs chargés d'un logiciel de contrôle de lancement de fusée. Quoi de plus séquentiel et impératif qu'un compte à rebours ? Ils avaient donc fait une analyse structurée, où le compte à rebours était l'élément de premier niveau de leur analyse. Dans une approche objet, on commencerait par identifier les éléments principaux intervenant dans le problème : le pas de tir, le centre de contrôle, etc. On finirait par retrouver le compte à rebours, mais seulement comme un détail d'implémentation du système de lancement, et non plus comme le point de départ de la conception.

De même, les encapsulations qui sont caractéristiques des approches objet en général ne regrouperont pas les mêmes éléments selon que l'on travaille en composition ou en classification. Supposons par exemple que nous voulions représenter un carré avec des propriétés graphiques et mathématiques. En composition, nous en ferions un objet autonome, qui regrouperait des objets de plus bas niveau, l'un décrivant les propriétés graphiques, l'autre les propriétés mathématiques ; d'autres objets similaires réutiliseraient ces mêmes sous-objets, mais toutes les propriétés de l'objet «carré» seraient regroupées dans un seul module. Inversement, en classification, le carré hériterait certaines propriétés de la classe des objets graphiques, et d'autres de la classe (ou de la facette) des objets mathématiques : ce seraient alors les propriétés communes à plusieurs objets similaires qui seraient regroupées. Ainsi, les propriétés d'une vue d'un objet sont regroupées en composition, alors qu'elles sont réparties sur une ou plusieurs branches du graphe en classification ; inversement, les propriétés communes à plusieurs objets sont dupliquées en composition, alors qu'elles sont factorisées en classification (Fig. 19).

Ceci montre bien qu'il n'existe pas *une* méthode d'organisation absolue, mais *des* méthodes orthogonales... et pas seulement en informatique. Si par exemple je désire trier des documentations publicitaires, je peux les regrouper par marques ou par types de produits. Chaque fabricant fournissant plusieurs produits, l'une ou l'autre des classifications peut se révéler plus favorable. Si je cherche souvent le catalogue d'un fournisseur, il vaudra mieux trier par marques, alors que si je recherche tous ceux qui peuvent fournir des éditeurs syntaxiques, il vaut mieux trier par type de produit. C'est en fonction de l'utilisation souhaitée et des diverses contraintes du projet que l'on choisira l'une ou l'autre des méthodes.

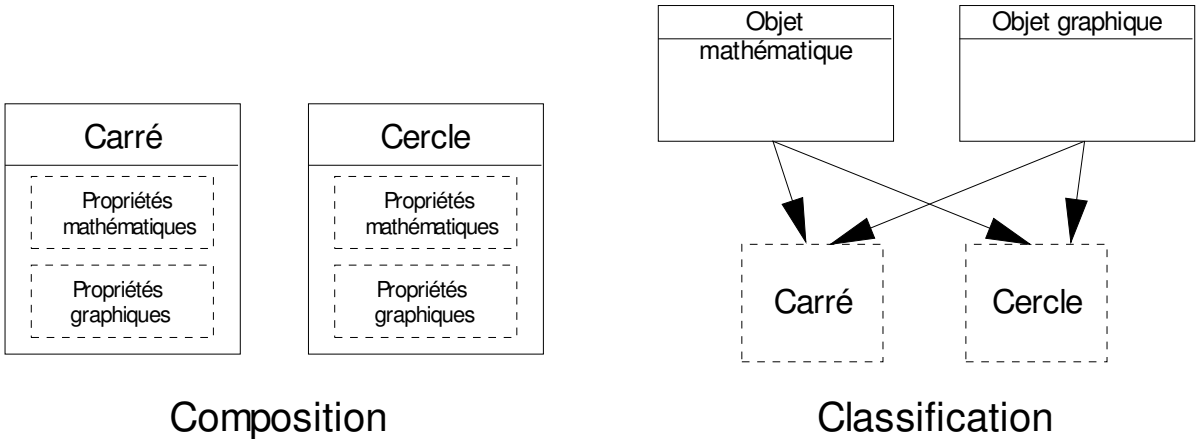


**Figure 19 :** Regroupement des propriétés en composition et classification

### 8.4 .4 Complexité

Nous avons vu (paragraphe 8.2.1.b) que le graphe d'une conception par composition était non transitif. Si nous ajoutons un objet à la conception, il dépendra conceptuellement d'un certain nombre de voisins *immédiats* ; si en moyenne nous avons $K$ dépendances, ajouter un élément dans un graphe à $N$ éléments augmentera la complexité de dépendance de $K$ ($K$ sera typiquement compris entre 2 et 10 ; au-delà, il faut se poser des questions sur la conception). La complexité de dépendance totale du graphe croîtra donc comme $K \times N$.

Dans un graphe de classification, un objet dépend transitivement de tous ses ancêtres. Le nombre de dépendances d'un objet est donc proportionnel au nombre total $N$ d'objets dans le graphe. Bien sûr, aucun objet ne dépend de *tout* le graphe. Soit $k$ la proportion du graphe dont dépend en moyenne l'objet ($k$ est toujours bien inférieur à 1). Ajouter un objet ajoute $k \times N$ dépendances, donc la complexité totale du graphe croîtra comme $k \times N^2$.

Il est certain que pour $N$ pas trop grand, $k \times N^2$ sera inférieur à $K \times N$. Dans ce cas, la complexité de dépendances engendrée par la classification sera inférieure à celle de la composition. Mais lorsque $N$ croît, et ceci *quelles que soient les valeurs de k et K*, tôt ou tard $k \times N^2$ dépassera (et même de beaucoup) $K \times N$.

Autrement dit, pour un petit projet, une structure en classification peut se révéler avantageuse sur le plan de la complexité ; mais la taille des projets réels tend à croître rapidement, et fatalement une structure par composition finira pas se révéler préférable. On mesure ici l'effet pervers qui peut se produire lorsqu'une entreprise effectue un projet pilote pour mesurer l'impact possible d'une nouvelle technologie. Un tel projet est par nécessité de taille réduite, et l'on évalue généralement l'effet sur des gros projets par extrapolation *linéaire*. Or rien ne dit que la croissance de la complexité est linéaire, et nous avons vu que dans le cas de la classification elle ne l'est pas. Les conclusions tirées du projet pilote risquent donc de n'être absolument pas applicables à un projet de taille réelle.

### 8.4 .5 Avantages et inconvénients de la composition

La composition conduit à une hiérarchie en niveaux d'abstractions qui se prêtent bien à une approche descendante selon des couches logicielles étanches. Elle promeut le développement de composants logiciels standard réutilisables. En revanche, des modifications des spécifications des composants de base peuvent avoir des répercussions importantes en termes de recompilation sur l'ensemble des projets. Il importe donc que ces composants soient *stabilisés* avant d'être utilisés à grande échelle. En pratique, il apparaît qu'après une période d'évaluation, les spécifications des composants tendent à se figer. Une fois cet état atteint, de nombreuses applications peuvent les utiliser. Si un besoin de modification se fait sentir, il faut apprécier s'il est préférable de modifier un composant existant ou d'en développer un nouveau sur des bases légèrement différentes. Enfin il peut être plus intéressant de modifier la conception d'un programme pour l'ajuster aux composants existants plutôt que de chercher à créer de toutes pièces un composant «parfait». Cet état d'esprit est malheureusement étranger à la plupart des développements logiciels, alors qu'il est universellement accepté dans le monde des composants matériels.

Rigoureuse, abstraite, la composition exige plus d'effort de réflexion de la part des développeurs dans les phases initiales de conception ; mais n'était-ce pas là l'un de ses buts ? Les bénéfices s'en font sentir après. Réflexion d'un utilisateur (de la société Stratégies, développeur du logiciel CADWIN) : «Nous avons eu du mal en phase de spécification, mais dès qu'on arrive à la maintenance, on se régale.»

Les objets sont faciles à maintenir : un objet rassemblant en un seul module toutes les opérations logiquement reliées, l'endroit d'une modification ou d'une réparation est entièrement défini dès lors que l'objet en cause est identifié ; de plus, l'étanchéité des implémentations vis-à-vis des

spécifications fait que l'on peut garantir qu'une intervention dans une implémentation n'aura pas d'effet fâcheux sur le reste du programme. Une demande de modification des spécifications au cours du développement d'un projet n'aura de conséquences que sur le module correspondant à l'objet du monde réel sujet à modification.

Il est difficile d'évaluer l'impact économique d'une méthode, car de nombreux autres facteurs peuvent intervenir. Par contre, des études ont été conduites pour mesurer la rentabilité effective de l'utilisation d'Ada (83) ; on citera en particulier l'étude de Reifer [Rei87, Rei89], portant sur 41 projets terminés, totalisant 15 millions de lignes de code ; 30 d'entre eux utilisaient la conception orientée objet par composition comme méthode d'analyse détaillée. Parmi les résultats intéressants de cette étude, on notera que la répartition conception/développement/mise au point, traditionnellement 40/20/40, était passée à 50/15/35, avec une diminution moyenne de 25% du taux d'erreurs dans les programmes. L'étude ne cherchait pas à différencier ce qui venait seulement de la méthode ou seulement du langage, car il est certain que les bénéfices ne peuvent être obtenus que par la conjonction de la méthode et d'un langage approprié.

En résumé, la composition nécessite un effort de réflexion et d'analyse intense ; elle se prête au développement de composants logiciels standard, immuables dans leur spécification, mais dont l'implémentation peut être remise en cause à tout moment sans perturber les utilisateurs, suivant le modèle des composants électroniques. Moins évolutive que d'autres méthodes en phase de conception, elle permet de mieux maîtriser la phase de maintenance : une modification d'une *implémentation* n'a aucune répercution sur les modules utilisateurs ; une modification d'une *spécification* a des conséquences limitées, et dont on peut déterminer l'étendue *a priori*.

## *8.4 .6 Avantages et inconvénients de la classification*

L'approche par classification conduit à un regroupement dans un même module de tous les aspects communs à différents objets. Le principal intérêt de cette méthode réside donc dans la *factorisation* des propriétés communes. En revanche, le graphe de dépendance est *transitif*, c'est-à-dire que les différentes propriétés d'une instance d'objet se trouvent réparties sur toute une branche du graphe.

Cette méthode favorisera la programmation par *adaptation* de logiciels, c'est-à-dire qu'il sera facile, à partir d'un objet qui correspond *presque* aux besoins du programmeur, de dériver un nouvel objet en ne récrivant que le différentiel, c'est-à-dire la différence entre l'objet souhaité et l'objet existant. En revanche, une modification d'un objet ancêtre modifiera le contexte et la sémantique de tous ses descendants. La classification permet donc de minimiser l'effort de récriture en phase de développement, lorsque la définition des objets évolue rapidement ; en revanche, les conséquences d'une modification en phase de maintenance peuvent s'étendre à une partie importante, et non connue *a priori*, du projet.

De façon générale, les méthodes par classification apportent une grande souplesse lorsque le domaine est mal connu, les concepts mouvants ou en cours de raffinement. Elles permettent bien de modéliser des hiérarchies de connaissance, et souvent de manipuler commodément les entités des applications graphiques. La liaison dynamique permet de pallier l'absence de parallélisme de beaucoup de langages (c'est pourquoi elle est utilisée dans les systèmes de fenêtrage comme X Window ou MacApp). Ces méthodes se prêtent en revanche assez mal à une approche descendante de la conception, et d'ailleurs beaucoup de partisans de la classification prônent le développement ascendant, en partant des bibliothèques de classes existantes. Une telle façon de faire, si elle permet de développer rapidement des prototypes lorsque l'on dispose d'une vaste bibliothèque de composants (cas des environnements graphiques), devient malheureusement vite impraticable lorsque la taille des logiciels atteint l'échelle industrielle.

On peut enfin se poser des questions sur la notion même de classification comme moyen de conception. Si toutes les sciences utilisent la classification, c'est pour répertorier des éléments

préexistants. La problématique est toute différente lorsqu'il s'agit de concevoir un système nouveau. On cherchera en vain dans l'ingénierie électronique, qui est pourtant la branche qui se rapproche le plus de l'informatique, un mécanisme pouvant se comparer à l'héritage pour la *conception* de systèmes.

## *8.4 .7 Composition et classification : ne peut-on les fusionner ?*

Parvenu à ce point, il est logique de se demander s'il ne serait pas possible de définir une méthode incorporant à la fois les concepts de composition et de classification pour jouir des avantages propres à chaque approche.

Nous ne saurions trop insister sur le fait que ces deux notions sont absolument orthogonales. [Mas89] utilisent d'ailleurs un exemple de composition pour montrer *le* cas (selon eux) où il ne faut pas utiliser l'héritage. Par exemple, un ordinateur est composé de transistors ; il n'hérite pas pour autant des propriétés des transistors ; que signifierait «polariser un ordinateur» ? Cette orthogonalité se traduit par le fait que les graphes de dépendance correspondant à chacune des structures possèdent des propriétés radicalement différentes : transitifs avec une complexité quadratique pour la classification, non transitifs avec une complexité linéaire pour la composition.

Les modes de raisonnement sont totalement différents[1], et même dans la vie courante des personnes différentes tendront à adopter naturellement l'une ou l'autre forme de description. Par exemple, nous participions un jour à une réunion de travail où nous disposions sur la table de jus de fruit concentré, que l'on pouvait diluer soit avec de l'eau plate, soit avec de l'eau gazeuse, permettant à chacun de préparer la boisson de son choix. Cette description est en fait une modélisation par composition. En classification, nous aurions dit que nous disposions de boisson à l'orange, subdivisée en orange piquante et orange plate...

La dichotomie entre les deux approches n'est cependant pas totale. Les langages orientés objet offrent nécessairement des possibilités de composition, et des langages purement «compositionnels», tels que Ada 83, offraient déjà, par les types dérivés, un mécanisme qui avait toutes les propriétés d'un héritage (statique), et même une forme de polymorphisme par les types à discriminants. Ces avancées vers l'autre domaine permettent à ces langages de récupérer quelques avantages spécifiques de «l'autre monde» sans sacrifier leur philosophie propre. Néanmoins, chacune des approches accordera la place dominante à la composition ou à la classification.

Nous verrons que de nombreuses méthodologies cherchent à exprimer *simultanément* des dépendances de type «composition» (agrégations) et de type «classification» (héritage). Ceci est dû à l'importance, à notre avis exagérée, accordée par de nombreux partisans de l'orienté objet à l'héritage, et comporte un risque inhérent sur le plan de la complexité : en superposant un graphe de composition et un graphe de classification, on risque d'obtenir une complexité globale qui serait la somme des deux complexités, si ce n'est pire.

Plutôt que de vouloir gérer simultanément les deux approches, il peut être plus raisonnable de les faire cohabiter indépendamment, c'est-à-dire de développer certaines parties d'un projet en composition, et d'autres en classification. Selon le profil de chaque partie, on choisirait la méthode la plus appropriée. Dans ce cas, il est tout de même nécessaire de définir un principe d'organisation global pour tout le projet, et il sera toujours plus facile de choisir la composition d'abord. En effet, il est possible d'incorporer dans un graphe de composition des branches qui, *localement*, sont des graphes d'héritage. Cela vient du fait que par définition, on ignore en composition comment sont implémentés les objets : si ceux-ci utilisent l'héritage, cela demeurera invisible aux niveaux supérieurs. Mais la démarche inverse est beaucoup plus difficile, puisque l'héritage suppose la transparence du graphe. Pour prendre une image, si l'on met une boîte transparente dans une boîte opaque, celle-ci reste opaque ; alors que si l'on met une boîte opaque dans une boîte transparente, celle-ci n'est plus transparente.

[1] Ce qui fait souvent ressembler les discussions entre partisans de chacune des méthodes à des dialogues de sourds.

Ceci ne signifie pas qu'il soit impossible d'utiliser de la composition dans une conception organisée par classification (nous avons déjà dit que c'était nécessaire), mais que l'utilisation de la composition rompt le mécanisme général de la classification, alors qu'une utilisation locale de l'héritage dans une structure par composition n'a pas d'impact sur la philosophie générale de la conception.

Nous pensons donc que même si l'on veut faire coopérer les deux approches, une méthode se doit de choisir une direction principale qui subordonne l'autre ; les parties développées avec l'«autre» philosophie doivent rester localisées. Et dans ce cas, l'approche par *composition d'abord* bénéficie d'un avantage, puisqu'elle est moins perturbée par des classifications locales que dans la démarche inverse. Dans la cinquième partie de cet ouvrage, nous proposerons une telle méthode donnant la priorité à la composition ; la classification n'y sera utilisée que là où elle apporte un gain notable, et toujours dans les couches basses de la conception, afin de limiter la complexité.

Bien entendu, il ne faudrait pas conclure de cette discussion que la classification doive être rejetée absolument : elle est certainement très performante dans le cadre de développements rapides, de complexité moyenne et/ou à faible durée de vie. Chaque projet possède ses contraintes propres, et le point important est de trouver, pour chaque type de développement, la méthode la plus adaptée ; la quatrième partie de cet ouvrage présentera quelques critères de choix permettant de déterminer la méthode adaptée à un projet en fonction de ses caractéristiques.

## 8.5 Exercices

1. Analyser en composition l'exemple de la paye qui a servi à illustrer la classification. Comparer l'organisation des solutions.
2. Analyser en classification l'exemple du cahier de comptes qui a servi à illustrer la composition. Comparer l'organisation des solutions.
3. Faire un tableau de l'évolution de la complexité d'un programme en fonction du nombre $N$ de modules (cf. paragraphe 8.4.4) en classification et en composition. On prendra $k = 0{,}1$ et $K = 5$ et $N = 10$, 20, 50, 100 et 200. Essayer ensuite avec $k = 0{,}2$ et $K = 4$ ou 6. Conclusion ? Si l'on estime qu'un module fait en moyenne 200 lignes de programme, à partir de quelle taille vaut-il mieux utiliser la composition ?

# 9

# Les méthodes entités-relations

## 9.1 Principes des méthodes entités-relations

Les principes des méthodes entités-relations ont été établis par Chen [Chen76], à une époque où l'on ne parlait pas encore d'objets. En fait, la notion d'«entité» est très proche de la notion d'objet, et les publications plus récentes se référant à ces méthodes se considèrent comme «orientées objet». Vu l'importance de ces méthodes, nous avons préféré leur consacrer un chapitre séparé, mais compte tenu de nos définitions précédentes, on peut aussi les considérer comme une forme particulière de méthode objet, où la dimension «verticale» est souvent absente. Tous les critères des autres méthodes (composition, classification) peuvent en effet se mettre sous forme de «relations» particulières, ce qui aboutit à un «aplatissement» total de la conception.

Un schéma d'entités-relations représente une description (statique) d'un système réel pour lequel on souhaite développer une application informatique. Il s'agit donc plutôt d'une méthode d'analyse que d'une méthode de conception comme celles que nous avons vues précédemment ; autrement dit, elle décrit plus le but à atteindre que le moyen d'y parvenir. Il faudra par la suite développer une conception, et donc transformer le modèle en un autre. Ce problème a donné lieu à de nombreuses publications ; on consultera par exemple avec intérêt [Bar92] qui propose une méthode pour passer d'un modèle entités-relations (REMORA) à un modèle objet compositionnel en Ada.

Les *entités* représentent des classes d'objets, munis d'attributs. Le modèle sous-jacent de la plupart des méthodologies associées est celui des bases de données relationnelles ; une entité correspond dans ce cas à une table dont chaque ligne correspond à une instance de l'entité, et chaque colonne à un attribut. Mais on peut également voir une entité comme un type article dont les différents composants correspondent aux attributs.

Ces entités (représentées par des rectangles) sont reliées par des *associations* (représentées par des lignes entre les rectangles) qui expriment les relations logique entre les entités; les relations sont marquées par un nom ou un verbe (dans un losange – ou un ovale selon la méthode) qui exprime la nature de la dépendance (Fig. 20)[1]. Les relations sont conceptuellement réciproques ; selon les conventions, on indique ou non la relation dans les deux sens (la réciproque de «passe» dans l'exemple de la figure 20 serait «est passée par»).

[1] Ces notations sont celles initialement définies par ces méthodes. Elles sont antérieures à UML, et d'ailleurs à l'origine de certaines de ses représentations.

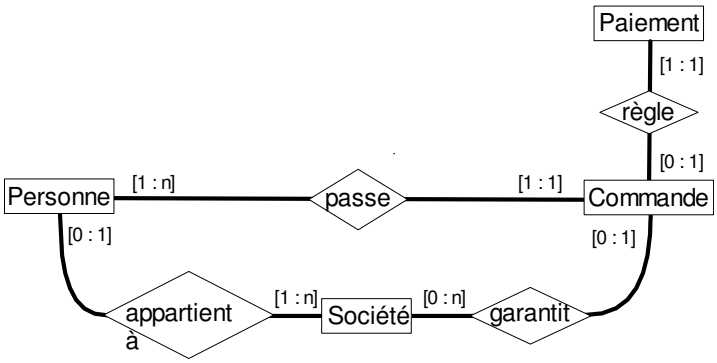


**Figure 20 :** Schéma d'entités-relations

On indique la *cardinalité* de l'association, c'est-à-dire le nombre d'instances d'objets situés d'un côté de la relation susceptibles de dépendre d'une instance d'objet situé de l'autre côté, sous la forme d'une paire de nombres [minimum : maximum]. Il existe de nombreuses variantes de ces diagrammes.

Le schéma de la figure 20 représente une partie des éléments entrant dans un système de gestion des commandes. Les entités sont *Personne*, *Commande*, *Paiement* et *Société*. Une personne peut passer plusieurs commandes, mais n'est entrée dans le système que si elle en a passé au moins une : c'est ce qu'indique la cardinalité [1 : n] au départ de la relation *passe* ; inversement, une commande ne peut être passée que par une seule personne, d'où la cardinalité [1 : 1] en sens inverse. Une personne appartient (mais pas forcément) à une société (cardinalité [0 : 1]), mais une société comporte plusieurs personnes (cardinalité [1 : n]). Un paiement règle une seule commande (cardinalité [1 : 1]), et une commande n'est associée à un règlement (et un seul) que lorsque le règlement a été effectué (cardinalité [0 : 1]). Enfin, une société peut garantir 0, 1 ou plusieurs commandes (cardinalité [0 : n]), mais une commande ne peut être garantie que par une société au plus (cardinalité [0 : 1]).

## 9.2 Ada et les méthodes entités-relations

Au niveau du langage, il serait bon de pouvoir représenter aussi directement que possible les éléments qui ont été définis par la conception. La représentation des entités ne pose pas vraiment de problème : ce sont des données classiques. En revanche, la notion de relation, qui était très simple au départ, s'est progressivement enrichie, au fur et à mesure du développement des méthodes. On a ajouté progressivement des attributs de relation, des conditions, etc. Si l'on peut facilement considérer que les entités sont assimilables à des objets (au sens des méthodes orientées objet), doit-on considérer les associations comme des objets ?

La logique voudrait évidemment que l'on réponde non, car une relation n'est pas de même *nature* qu'un objet. Cependant, il serait logique d'avoir des types de données pour représenter les associations. Dans un langage purement orienté objet, où la classe est le seul moyen de représenter des données, on se trouve ici face à une contradiction. En Ada en revanche, on peut très bien utiliser des types de données pour représenter des associations sans pour cela devoir les assimiler à des objets.

[Jea93] a présenté différentes façons de modéliser les associations en Ada 83. Nous les présentons brièvement ci-dessous, en y ajoutant les nouvelles possibilités d'Ada 95. Il convient d'abord de noter que la notion d'«association» est extrêmement vague et peut servir à exprimer, dans un cadre unique, toutes sortes de dépendances entre les entités. Ainsi, on peut avoir des relations «est composé de» (agrégation) exprimant qu'une entité est composée d'autres entités, et des relations «est un» (héritage) exprimant qu'une entité est une spécialisation d'une autre. Dans ces cas, l'implémentation est triviale : un article composé des différents éléments pour l'agrégation, un type (étiqueté) dérivé pour l'héritage.

L'agrégation peut poser un problème ; dans le modèle entités-relations, les relations sont censées être réciproques. Par conséquent, si «A contient B», on doit pouvoir dire «B est contenu dans A». L'utilisation d'un simple type article ne permet pas de traduire cette relation inverse (d'utilisation rare, reconnaissons-le). Si elle est nécessaire, on peut l'obtenir au moyen d'un autopointeur :

```
type A;

type B (Englobant : access A) is
   record
       ...
   end record;

type A is limited
   record
       Composant_B : B (A'Access);
       ... autres composants
   end record;
```

La relation «A contient B» correspond à l'imbrication physique des champs de l'article, et la relation «B est contenu dans A» est obtenue par le pointeur `Englobant`. Au risque de lasser le lecteur, force nous est de constater qu'Ada est, à notre connaissance, le seul langage permettant ainsi de gérer *automatiquement* ce type de relation inverse. Pour les autres types de relation, il convient d'analyser soigneusement leur type pour déterminer l'implémentation appropriée ; [Jea93] présente d'ailleurs une taxonomie des relations.

Si la relation est unidirectionnelle, on pourra utiliser une référence, sous forme d'un composant pointeur si l'entité associée est susceptible de changer dans le temps, ou d'un discriminant pointeur si l'entité associée est unique et non facultative (traduisant ainsi un couplage beaucoup plus fort entre les entités). Dans le cas des relations où la symétrie est importante, on pourra utiliser des pointeurs doubles. Si la relation est multiple (cardinalité supérieure à 1), on pourra définir une table (ou une liste chaînée) représentant les éléments participant à l'association. Il est facile de faire un générique fournissant la gestion de ces tables d'associations.

Noter que certaines méthodes ajoutent des attributs aux relations ; on utilisera dans ce cas le même principe, mais l'association sera représentée par un article contenant les attributs, ainsi éventuellement que la table d'associations.

On voit à travers ces quelques exemples qu'il n'y a pas de *moyen unique* pour représenter toutes les formes d'associations, mais que c'est la *variété* des structures de données et autres outils fournis par Ada qui permet de représenter toutes les formes d'associations.

## 9.3 Exercices

1. Représenter le schéma d'entités-relations de l'exemple de paye du paragraphe 8.3.4
2. Les schémas d'entités-relations sont-ils appropriés aux problèmes de gestion ? Et aux problèmes de temps réel ? Défendez votre réponse en fonction des contraintes de chacun de ces domaines.

# 10
# Méthodologies

Nous avons présenté les différentes *méthodes*, c'est-à-dire les grands principes permettant d'organiser les développements de logiciels. Mais des principes ne sont pas suffisants en pratique : il faut également des modalités d'application, des directives organisées qui expliquent concrètement *comment* conduire un développement logiciel. C'est le but des méthodologies.

Certaines méthodologies étaient, au moins au départ, fondées sur une seule méthode. Mais souvent, la méthode unique n'est pas suffisante pour couvrir les contraintes souvent contradictoires d'un développement. Aussi de nombreuses méthodologies empruntent-elles à plusieurs méthodes, mais privilégient généralement l'une d'entre elles : c'est la *direction principale* de la méthodologie. Après les principes généraux, nous présenterons quelques-unes des méthodologies les plus courantes, selon leur direction principale.

. La mode du tout orienté-objet et d'UML tend à créer une sorte de pensée unique. Et pourtant, l'informatique couvre aujourd'hui des domaines tellement variés qu'il semble difficile de croire qu'un seul processus puisse répondre à tous les besoins. C'est pourquoi nous présentons ci-dessous certaines méthodologies anciennes, qui ne sont parfois plus guère utilisées. Nous pensons qu'il est utile que le lecteur soit convaincu qu'il existe plusieurs façons de faire, et que le rôle de l'ingénieur est avant tout de déterminer quelle technologie est la plus adaptée à son besoin, plutôt que d'adopter aveuglément la seule démarche dont on parle dans les journeaux.

## 10.1 Démarche et notation

La plupart des méthodologies comportent deux éléments principaux : une *démarche* et une *notation* [Rum94]. La démarche définit le processus de conception ; elle décrit les différentes étapes permettant de passer de l'énoncé d'un problème à sa solution. La notation décrit un ensemble de symboles permettant d'exprimer la conception. Son but est de guider la démarche et de produire un document de référence pour la maintenance. Enfin, la plupart des méthodologies sont dotées d'outils supports facilitant l'application de la démarche et automatisant l'expression de la notation, notamment lorsque elle est essentiellement graphique.

Le point le plus fondamental d'une méthodologie est la démarche. Le point le plus visible est la notation. Il importe donc de toujours se rappeler que la notation doit être subordonnée à la démarche : trop souvent, nous avons pu constater que l'application d'une méthodologie se bornait à utiliser la notation, ce qui peut conduire à des échecs, car les outils supports ne permettent alors pas d'exprimer l'intention réelle des concepteurs.

*Une méthodologie, ce n'est pas seulement tracer des flèches entre des boîtes !*

## 10.2 UML

Pendant longtemps, chaque méthodologie avait ses notations propres. Certaines d'entre-elles ont parfois été adoptées par d'autres méthodologies que celle d'origine, mais généralement avec des «adaptations». Ceci a conduit à des difficultés d'interprétation des symboles courants (formes des boîtes et des flêches).

UML (Unified Modeling Language, langage de modélisation unifié) est né du besoin d'unifier les différentes notations. Il est apparu en 1994, quand J. Rumbaugh rejoignit Rational, où G. Booch était responsable des méthodologies. En 1995, I. Jacobson rejoignit l'équipe, et après plusieurs itérations, UML 1.0 fut publié en Janvier 1997. Plusieurs fabricants firent alors équipe avec Rational, ainsi que l'Object Management Group. En Septembre 1997, les nouveaux partenaires produisirent UML 1.1, première version largement diffusée de la notation.

La méthode continua d'évoluer jusqu'à la version 1.4. Courant 2001, les membres de l'OMG décidèrent que le temps d'une mise à jour majeure était venu, ce qui produisit UML 2.0. A l'heure où nous écrivons ces lignes (2004), UML 2.0 est stabilisé et la plupart des documents sont disponibles, mais il reste encore certains éléments en cours de parution.

Il importe de comprendre qu'UML n'est qu'une notation, pas une méthodologie. Hélas, bien des gens ignorent le sage précepte qui conclue le paragraphe précédent, et «font de l'UML» comme méthodologie. Ceci ne veut strictement rien dire; on peut utiliser une méthodologie qui utilise UML comme représentation, mais l'important est la méthodologie, pas la représentation.

UML se voulait indépendant des langages de programmation; en pratique, surtout avant la version 2, il était directement calqué sur C++. On y retrouve des notions comme la distinction entre membres publics, protégés, et privés, qui n'existent que dans ce langage. En revanche, la notion de module élémentaire, comme les paquetages Ada, est difficile à représenter (les paquetages UML correspondent plutôt aux namespace de C++). UML 2 a introduit la notion de décomposition hiérarchique (qui est à la base de la méthode HOOD), permettant enfin une décomposition descendante ; on peut s'étonner qu'il ait fallu tant de temps pour introduire une notion aussi fondamentale.

UML définit 13 formes de diagrammes différents, qu'il serait trop long de détailler ici ; de nombreux ouvrages sont à la disposition du lecteur intéressé. Mais il est important de comprendre qu'aucune méthodologie ne saurait utiliser tous les diagrammes, pas plus qu'aucun programme n'utilise jamais toutes les possibilités de son langage de programmation. Le but d'UML est de fournir une infrastructure de description, afin que chaque méthodologie y puise les représentations qui lui sont nécessaires.

## 10.3 Méthodologies en programmation structurée

La principale méthodologie formalisée utilisée de façon industrielle est SA/SD (*Structured Analysis/Structured Design*), définie initialement par Yourdon et Constantine [You79] et enrichie par DeMarco, Page-Jones... Elle a introduit une notation, largement reprise par la suite, pour exprimer les *flots de données* et leurs transformations : un cercle représente un traitement, une flèche un flot de données qui se propage de traitement en traitement. Les éléments à l'origine des données (*sources*) ou destinataires finals de celles-ci (*puits*) sont représentés par des rectangles. De plus, les données peuvent être stockées dans des *réservoirs de données* représentés par deux traits parallèles. Les diagrammes résultant sont appelés «diagrammes en flots de données», ou DFD (*Data Flow Diagram*). La figure 21 en donne un exemple.

Sur ce diagramme, nous voyons que le *client* passe une *commande*. La saisie de la commande donne naissance à un *ordre de fabrication* en cuisine et fournit le *prix* à payer. Le client effectue un paiement, qui va dans la *caisse* (un réservoir de données). Cela produit un *calcul de monnaie*, qui fournit au client la monnaie, en provenance de la caisse. De l'autre côté, l'ordre de fabrication a déclenché la fabrication d'un *hamburger* à partir de *pain* et de *viande*, tous deux provenant du *réfrigérateur*, autre réservoir de données.

Ce type de diagramme permet d'exprimer les traitements et les flots de données ; il n'exprime pas en revanche les synchronisations. Par exemple, on ne voit pas que le client ne peut fournir le paiement qu'une fois le prix connu. Le diagramme des traitements est complété par un dictionnaire de données qui décrit, de façon textuelle, les caractéristiques des données échangées. Bien entendu, ce sont les traitements qui constituent l'unité principale de structuration, et le découpage en couches s'obtient en subdivisant les traitements généraux en traitements plus spécifiques.

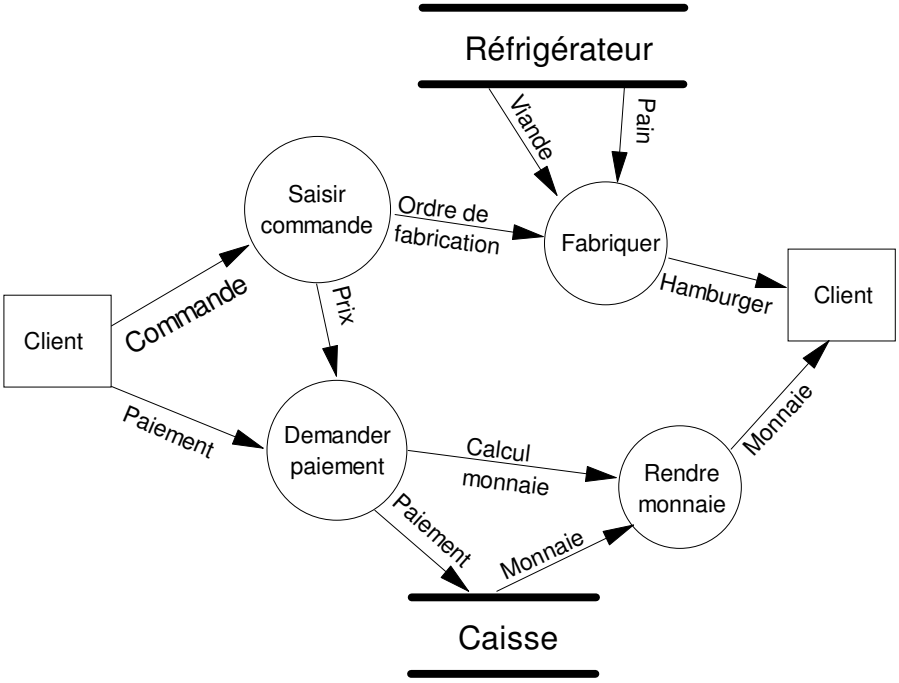


**Figure 21 :** Diagramme de flot de données SA

La méthode a par la suite été enrichie par Ward et Mellor en fonction des contraintes particulières de la programmation en temps réel pour donner la méthode SART (*SA Real-Time*) [War85]. En particulier, on considère qu'un système est caractérisé par un *état* courant, et qu'il subit des *transitions* qui le font passer d'un état à l'autre. Ceci est décrit par des diagrammes états-transitions, qui ont été repris (avec d'innombrables variantes) dans la plupart des méthodologies.

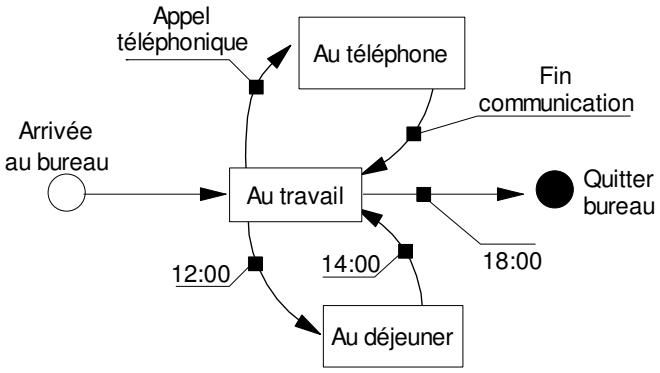


**Figure 22 :** Diagramme d'états-transitions

Sur le diagramme de la figure 22, on part de l'état initial «Arrivée au bureau» pour arriver à l'état «Au travail». Sur l'événement «Appel téléphonique», on passe à l'état «Au téléphone» dont on revient sur l'événement «Fin communication». De même, l'événement «12 heures» fait passer dans l'état «Au déjeuner» dont on sort par l'événement «14 heures». Enfin l'événement «18 heures» fait passer à l'état final, «Quitter bureau».

## 10.4 Méthodologies en composition

### *10.4 .1 La méthode de Booch*

Cette méthode, qui est à la base des autres méthodes orientées objet par composition, a été définie par G. Booch, dans son livre *Ingénierie du logiciel avec Ada* [Boo88]. Son but était de fournir une méthode adaptée à Ada, mais ceci ne l'empêche pas d'être utilisable avec d'autres langages. Depuis, Booch a défini une autre méthode (Rose, [Boo91]) qui prend en compte les aspects de classification ; le terme «méthode de Booch» tend donc à devenir ambigu.

Pour mettre en œuvre sa méthode, Booch a défini un certain nombre d'étapes permettant de guider la démarche pour obtenir de «bons» objets. Ces étapes ont été reprises avec plus ou moins de modifications ou de précisions dans la plupart des méthodologies par composition.

*Identifier les objets*. Cette première étape vise à identifier les objets *du monde réel* que l'on voudra réaliser. Pour cela, on doit identifier les propriétés caractéristiques de l'objet.

Cette étape est bien entendu celle qui demande le plus de talent et d'expérience personnelle au programmeur ; un moyen relativement informel pour identifier les objets consiste à faire une description informelle (en français) du problème. On pourra déduire les bons candidats objets des noms utilisés dans cette description, et leurs propriétés des adjectifs et autres qualifiants.

*Identifier les opérations*. On cherchera ensuite à identifier les actions que l'objet subit et provoque. Les verbes utilisés dans la description informelle précédente fournissent de bons indices pour l'identification des opérations. C'est également à cette étape que l'on pourra définir les conditions d'ordonnancement temporel des opérations, si nécessaire.

*Etablir la visibilité*. L'objet étant identifié par ses caractéristiques et ses opérations, on définira ses relations avec les autres objets ; autrement dit, on établira quels objets le «voient» et quels objets «sont vus» par lui. Autrement dit, on insérera alors l'objet dans la topologie du projet.

*Etablir l'interface*. A partir de là, on définit l'interface précise de l'objet avec le monde extérieur. Cette interface définit exactement quelles fonctionnalités seront accessibles, et sous quelle forme. Cette étape doit s'écrire de préférence à l'aide d'une notation formelle ; une spécification de paquetage Ada constitue une notation tout à fait acceptable.

- *Implémenter les objets*. La dernière étape consiste bien entendu à implémenter les objets en écrivant le code correspondant aux spécifications dans un langage de programmation. Cette étape peut nécessiter la définition de nouveaux objets de plus bas niveau d'abstraction, ce qui provoque l'itération de la méthode.

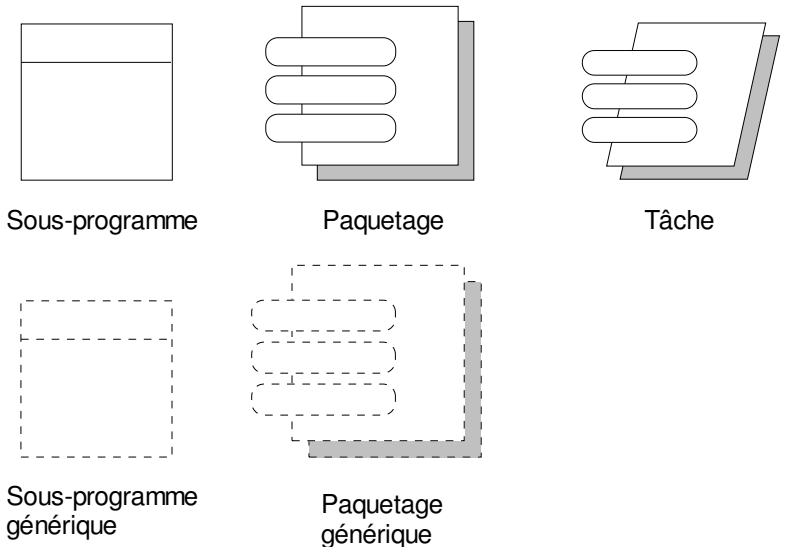


**Figure 23 :** Diagrammes de Booch

Booch a introduit des diagrammes, représentés en figure 23, généralement utilisés pour représenter les entités Ada. Ils servent à construire des graphes qui, comme nous l'avons vu dans le chapitre sur la composition, sont essentiellement structurels et dont les flèches expriment les liens de dépendance entre entités. Elles correspondent donc aux clauses **`with`** du programme Ada.

### *10.4 .2 La méthode HOOD*

La méthode HOOD (*Hierarchical Object Oriented Design*) [Ros97] a été développée par CISI Ingénierie, Matra et la société danoise CRI pour le compte de l'Agence spatiale européenne. Elle a été choisie parmi quinze méthodes concurrentes pour le développement des logiciels de la station spatiale Columbus et de l'avion spatial Hermès[1] et est largement utilisée aussi bien dans le domaine spatial que dans l'industrie en général. Elle intègre non seulement une méthode de conception, mais également la définition de toute la documentation associée, et permet de faire la liaison avec des méthodes formelles. Elle a été conçue pour être utilisée avec des outils supports qui sont aujourd'hui présents sur le marché ; la situation est même concurrentielle. Plusieurs versions de la méthode sont apparues, suivant l'expérience acquise et les recommandations des utilisateurs (le HUG, *HOOD User Group*). La première version stable et largement utilisée fut la 3.1; elle servit de base à une extension, appelée HRT-HOOD (Hard Real-Time[2] HOOD) pour les logiciels demandant un contrôle précis et prouvable de l'ordonnancement. La version actuelle est la version 4, qui permet notamment d'introduire une dimension de classification dans la conception, mais de façon toutefois très contrôlée

La méthode elle-même résulte de la fusion des notions de machines abstraites et d'objets. Un projet est donc décomposé en objets qui sont, en principe (les évolutions récentes de la méthode on assoupli ce point), uniquement des machines abstraites. On distingue les objets passifs, s'exécutant séquentiellement et les objets actifs pouvant s'exécuter en parallèle.

Un objet peut être terminal ou non terminal. Dans ce dernier cas, toutes les opérations fournies sont réalisées par «sous-traitance» à des objets «enfants», conduisant à la structure hiérarchique qui a donné son nom à la méthode. Nous pouvons illustrer cette notion de sous-traitance de la façon suivante : imaginons un objet «téléviseur». Extérieurement, il est muni d'opérations (les boutons) telles que «Marche/arrêt», «Réglage volume» ou «Choisir chaîne». Effectivement, l'utilisateur considère qu'il met en marche «la télévision» ou qu'il règle «la télévision». *En fait*, le bouton d'arrêt fait partie *au niveau de l'implémentation* (quand on ouvre la boîte noire) de l'alimentation secteur, alors que le bouton de volume appartient à l'ampli son et le sélecteur au tuner. En termes HOOD, on dirait que la télévision *sous-traite* ces opérations aux objets enfants correspondants. Ce principe est illustré par la figure 24.

[1] Les vicissitudes qu'on subies ces projets sont sans rapport avec l'utilisation de la méthode HOOD...

[2] Temps réel dur

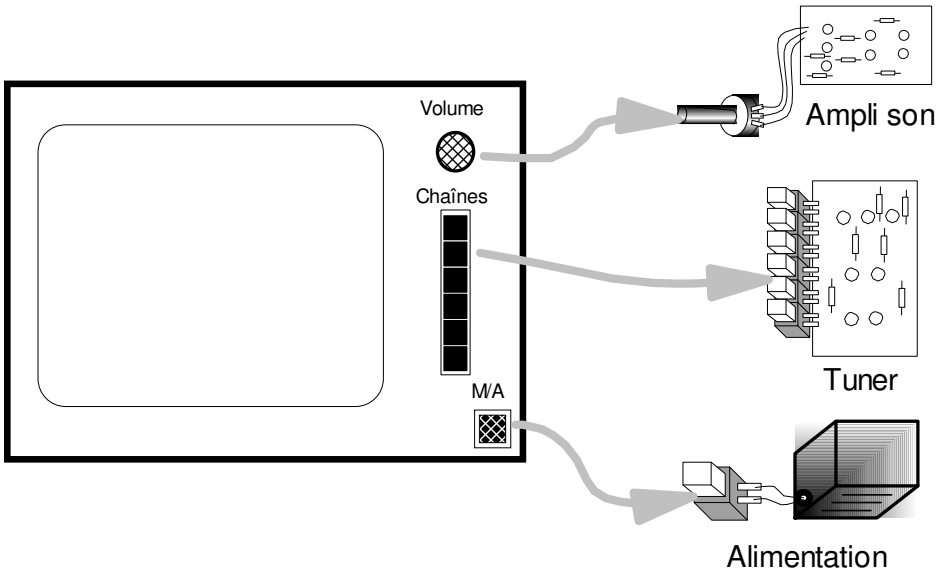


**Figure 24 :** HOOD et la sous-traitance

La méthode sépare clairement les différents aspects de l'analyse. En plus de la relation hiérarchique qui décrit la structure du projet, on analyse et on décrit de façon indépendante les aspects impératifs (correspondant à l'exécution séquentielle des opérations, relevant donc d'une analyse en programmation structurée et décrite par du code Ada) et les aspects réactifs (description des interactions entre objets actifs, susceptibles de modélisation par des méthodes formelles telles que les réseaux de Petri, ou utilisation standardisée de formes Ada). Cependant, chacune de ces descriptions est fournie localement, pour chaque objet ; on ne trace jamais de flot de données global. Ceci permet de conserver une stricte hiérarchie et l'étanchéité des niveaux d'abstractions.

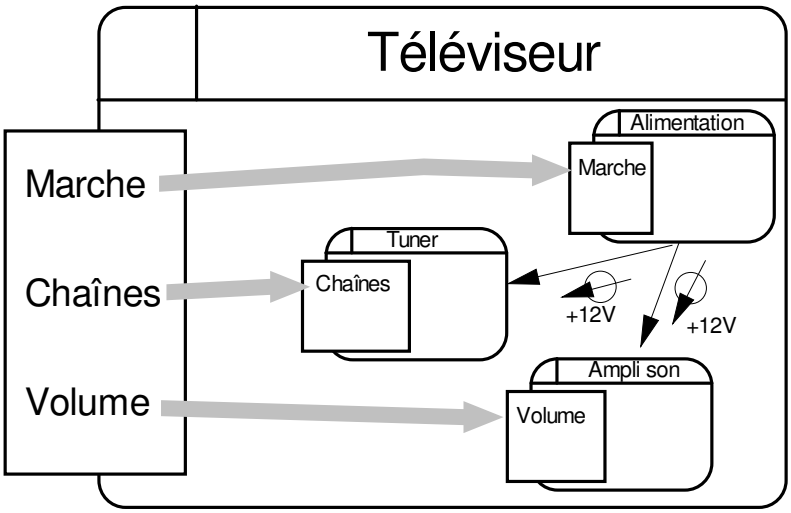


**Figure 25 :** Représentation de la télévision par des objets HOOD

La méthode définit un formalisme graphique et un formalisme textuel, décrivant les flux de contrôle, de données et d'exceptions aussi bien que la structure des dépendances logiques. Notre téléviseur peut ainsi être représenté sous forme d'objets HOOD comme sur la figure 25. Le diagramme montre également des relations entre les objets enfants (ici, l'alimentation 12V fournie par l'alimentation aux autres modules), non visibles de l'extérieur. Les flèches d'utilisation peuvent porter des informations de flux de données (les flèches dans un rond) et de flux d'exceptions.

La méthode est également adaptée à la définition de systèmes distribués, grâce à la notion de *nœud virtuel*. Un nœud virtuel représente une unité possible de distribution. Une étape de la méthode consiste à attribuer un nœud virtuel à chaque objet de la conception. Les nœuds virtuels sont ensuite eux-mêmes alloués à des nœuds physiques (comme des ordinateurs sur un réseau). On sépare donc la répartition du système entre une répartition logique et une répartition physique. Ceci

permet notamment d'allouer plusieurs nœuds virtuels à une même machine : il est ainsi possible d'avancer la conception et même la validation du logiciel indépendamment de sa répartition physique, et même avant d'avoir fait les choix définitifs de matériel.

On trouvera une description complète de la méthode HOOD dans [Ros97].

### *10.4 .3 La méthode Mac_Adam*

Mac_Adam [Rig87,Rig89] est une chaîne complète de développement de logiciels, comportant donc plusieurs facettes. La partie méthodologique est directement inspirée de la COO de Booch, avec une orientation plus spécifique du temps réel. Une première étape permet de définir le contexte du problème et les interfaces externes, conduisant à une modélisation de l'espace de problème. Du rapprochement entre ce modèle abstrait et les contraintes spécifiques du temps réel, on dérive un regroupement en tâches, la définition des protections et la structuration en sous-systèmes.

En plus de cet aspect purement «méthodologie de conception», la méthode gère le suivi des unités de test et d'intégration, et la documentation. Celle-ci est conforme à la directive 2167 du DoD [DoD87]. Un outil associé permet de suivre les différentes étapes de la méthode jusqu'à la génération de squelettes de programmes.

### *10.4 .4 La méthode de Buhr*

Composition et classification ne sont pas les seules façons d'organiser des objets ; la méthode de Buhr [Buh84,Buh89] est une autre méthode de conception «orientée objet», visant principalement la conception de systèmes temps réel. Cette méthode utilise le comportement (*behaviour*) comme critère de décomposition verticale, selon une technique inspirée de celles utilisées dans le monde de l'automatique. C'est donc une méthode orthogonale à la fois à la composition et à la classification[1].

Le module de base dans la méthode de Buhr est soit la boîte (*box*), qui est une unité passive, soit le *robot* qui est une unité active. Ces deux unités sont appelées globalement des *machines*. Une machine est une unité de comportement, possédant des *ports* qui lui permettent de communiquer avec d'autres robots par l'intermédiaire de *canaux.* Des *événements* circulent dans ces canaux. Ces notions sont par la suite raffinées sous forme de *boutons* qui doivent être *poussés* par un *doigt* pour déclencher une action. Certains boutons ne peuvent s'activer que sous l'action combinée d'un doigt qui pousse et d'un doigt qui *tire* (*push-pull button*).

Les grandes étapes de l'analyse d'un niveau d'abstraction selon la méthode de Buhr sont :

*Séparation des comportements*. Il s'agit de définir ici les unités de comportement qui seront représentées par des machines individuelles.
*Définition des protocoles abstraits.* Il faut identifier les protocoles selon lesquels les machines vont communiquer entre elles.
*Définition des intermédiaires*. Les canaux peuvent eux-mêmes être des machines relativement compliquées. On va définir ici les machines intermédiaires permettant d'assurer les communications.
*Définition des protocoles concrets*. Il faut définir pratiquement les détails des protocoles, en tenant compte des particularités structurelles des ports correspondants.
*Définition des agendas.* Il faut maintenant spécifier l'ordonnancement temporel des protocoles.

- *Valider le comportement*. Reparcourir la définition globale élaborée jusque-là et envisager la réponse du système à différents événements.

[1] Cette notion de méthodes orthogonales est explicitement utilisée par Buhr dans [Buh89].

Comme on le voit, le grand intérêt de la méthode de Buhr est de permettre la prise en compte de l'ordonnancement temporel dès les premières étapes de la conception, ce qui en fait un système très efficace pour le développement de logiciels temps réel. La notion de machine intègre bien également les composants matériels, ce qui permet de définir un système (matériel et logiciel) comme un tout, en remettant à plus tard le stade où l'on doit décider de ce qui doit être implémenté par logiciel ou par matériel. En revanche, elle se prête moins bien à une approche descendante par niveaux successifs, ce qui limite sa portée aux systèmes de taille relativement modeste.

La méthode est supportée par un formalisme graphique, et il existe des outils d'environnement permettant de le mettre en œuvre, aussi bien pour la conception elle-même que pour la vérification automatisée des diagrammes, la simulation temporelle et la génération de squelettes de code.

### *10.4 .5 La méthode Rose*

Remplacer par RUP?

La méthode Rose [Boo91], la «nouvelle» méthode de Booch, reprend les principes de la méthode initiale, mais y incorpore une dimension de classification. Les étapes sont restées à peu près les mêmes que dans la méthode d'origine, mais les diagrammes se sont enrichis pour décrire différentes formes de dépendance entre objets : utilisation, instanciation, héritage, métaclasses... Ces formes sont cependant limitatives (au contraire des méthodes entités-relations). La méthode étant moins orientée Ada que précédemment, on distingue les diagrammes de classes des diagrammes de modules.

La méthode travaille par niveaux d'abstraction dans la direction verticale ; en particulier, on cherche à définir des sous-systèmes étanches.

## 10.5 Méthodologies en classification

On pourrait s'attendre à ce que la vogue de la classification ait fait naître de nombreuses méthodes de conception, ou au moins qu'une méthode s'impose largement. Or il n'en est rien. Le monde de la classification «fait de l'UML», mais UML est une représentation, pas une méthode. On cherchera en vain une démarche méthodologique par classification formalisant le processus permettant de passer de l'énoncé d'un problème à sa réalisation informatique, comme cela existe pour HOOD, Buhr ou SART par exemple.

Ceci peut s'expliquer par l'histoire : le monde de la classification est parti d'une approche langage, et le souci a plus été de représenter les arborescences de classes (qui deviennent rapidement très compliquées) que de guider le processus de conception lui-même. Enfin, beaucoup de méthodes (elles bien établies) intègrent une dimension de classification, sans que celle-ci constitue la direction principale. C'est en particulier généralement le cas des méthodologies «entités-relations» qui sont présentées au paragraphe suivant.

## 10.6 Méthodologies par entités-relations

### *10.6 .1 Shlaer et Mellor*

La méthode de Shlaer et Mellor [Shl88] est caractéristique des méthodologies entités-relations modernes, qui se considèrent «orientées objet» dans la mesure où les «entités» manipulées sont des abstractions d'objets du monde réel. Elle vise essentiellement les applications centrées sur les bases

de données, car les entités sont considérées comme des «tables» et les relations comme des «jointures» (au sens de SQL). Certaines relations plus compliquées [m : n] peuvent elles-mêmes être représentées sous forme de tables. L'héritage est introduit par la relation «est-un». Ce modèle structurel est complété par des diagrammes en flots de données exprimant les traitements.

C'est essentiellement une méthode d'analyse, dans la mesure où l'ouvrage de référence se concentre essentiellement sur la façon d'obtenir les renseignements du client afin d'obtenir une «bonne» modélisation du système à concevoir. En revanche, on ne parle que très peu de la dimension verticale : comment subdiviser un système complexe en sous-systèmes plus simples.

### *10.6 .2 OMT / Rumbaugh*

La méthode OMT [Rum94] ressemble beaucoup par certains points à la méthode de Shlaer et Mellor, mais l'aspect «bases de données» n'y est pas fondamental : celles-ci ne sont considérées que comme une façon parmi d'autres d'implémenter les notions d'entités et d'attributs. L'idée de base est que tout système doit être décrit de trois façons orthogonales :

L'aspect structurel, décrit par un schéma d'entités-relations. Il comporte des notations spéciales pour les relations d'agrégation et d'héritage.
L'aspect événementiel, décrit par un schéma états-transitions. Il décrit les événements susceptibles de provoquer des traitements, et les différents états du système.

- L'aspect fonctionnel, décrit par un diagramme de flots de données. Ce schéma décrit les traitements effectués sur les données.

La méthode accorde la place principale à l'aspect structurel, les autres aspects n'intervenant que plus tard dans la conception. Comme avec Shlaer et Mellor, la description se veut complète, c'est-à-dire qu'elle veut exprimer *tous* les aspects de *tous* les composants du système. En conséquence, la décomposition *verticale* n'est que peu présente : le schéma d'entités-relations est découpé en «feuillets», mais qui constituent plus une facilité pour éviter de manipuler de trop grands diagrammes qu'une découpe hiérarchisée. L'analyse fonctionnelle demande d'identifier des sous-systèmes, mais ne dit rien des critères à utiliser pour ce faire. Il s'agit donc d'une méthode très complète au niveau spécification, mais on peut lui reprocher un manque de «profondeur» si on veut la poursuivre vers les étapes ultérieures du développement.

## 10.7 Méthodologies : oui, mais...

Il n'est pas question de remettre en cause l'importance des méthodologies, mais ceci ne dispense pas de certaines précautions. Aucune méthodologie, pas plus qu'aucun langage[1] ne peut garantir le succès d'un développement ; toute méthodologie a ses forces et ses faiblesses, et une méthodologie mal comprise ou mal utilisée peut même *compliquer* le processus de conception. C'est pourquoi nous pensons qu'il est utile de terminer cette partie par quelques mises en garde.

Tout d'abord, être «orienté objet» est devenu une nécessité marketing à défaut d'être une nécessité technique ; toutes les méthodes utilisent maintenant, de près ou de loin, ce terme. Il importe de bien voir ce que cela recouvre. En particulier, l'importance exagérée accordée souvent à l'héritage a conduit à rajouter une dimension d'héritage à beaucoup de méthodes où cela n'était pas indispensable.

Ensuite, la méthodologie doit apporter une *simplification* par rapport au langage ; il ne faut jamais oublier que le but premier est de lutter contre la complexité. Une méthode mal utilisée ou inadaptée au projet peut conduire à l'effet inverse. Il nous est ainsi arrivé de conduire des audits sur des projets où, après avoir peiné sur des pages et des pages de boîtes et de flèches, nous avons fini par demander à voir les spécifications Ada... qui exprimaient finalement beaucoup mieux la

[1] Fût-ce Ada (mais si, mais si...).

structure du projet que les diagrammes. En particulier, il est *inutile* de poursuivre l'application de la méthode au-delà du niveau sémantique du langage : à quoi bon tracer des arborescences de parcours et écrire du pseudo-code s'ils ne fournissent pas une vue plus synthétique que le programme Ada[1] lui-même ?

Les méthodologies ont toujours été soumises à deux contraintes contradictoires : être limitatives pour fournir un cadre rigide, donc directif, aux développeurs, et être puissantes, donc permettre d'exprimer un maximum de choses. Le côté directif de la méthode est souvent mal perçu, et le programmeur est souvent tenté d'abandonner la méthode plutôt que de s'y plier. On peut résumer (un peu caricaturalement) ceci de la façon suivante:

La méthode: il est interdit de faire X.
Le programmeur: mais je veux faire X!
La méthode: désolé, mais c'est interdit
- Le programmeur: dans ce cas, j'abandonne la méthode.

L'histoire de la méthode HOOD par exemple est caractéristique : d'une approche strictement hiérarchisée en machines abstraites, on est passé progressivement à des «environnements» échappant à la hiérarchie, puis à des «types de données abstraits» qui échappent aux exigences des machines abstraites; et finalement, HOOD 4 a introduit une dimension de classification.

Cette évolution élargit le champ d'application des méthodologies, en diminuant les contrôles qu'elles apportent. Si l'on décide d'adopter une méthodologie «puissante», donc vraisemblablement peu contraignante, il peut être souhaitable d'en limiter les possibilités pour améliorer les contrôles : on définira alors un «cadre d'utilisation» de la méthodologie, exprimant ces restrictions. Une liste soigneusement définie d'exceptions aux règles (et contrôlée par le responsable qualité) autorisera des violations dans certains cas au programmeur, et permettra d'éviter le syndrome ci-dessus.

[1] Bien sûr, avec des langages de plus bas niveau qu'Ada, cette étape supplémentaire peut être nécessaire.

# 11

# Maquettage et développement progressif

Le développement progressif par maquettage ne constitue pas à proprement parler une *méthode*, tout au moins pas au sens des méthodes que nous avons vues dans les chapitres précédents. Il s'agit plutôt d'un contexte méthodologique compatible avec la plupart des méthodes, bien que plus particulièrement facile à mettre en œuvre avec les méthodes orientées objet.

## 11.1 Maquettage et prototypage

Maquettage, prototypage, ces mots sont utilisés fréquemment en informatique, parfois de façon interchangeable, et généralement avec des significations mal définies. Rappelons donc leur définition dans le domaine général de l'industrie et voyons comment ils peuvent s'appliquer à l'informatique.

Un *prototype* est le premier exemplaire d'un produit destiné à être produit par la suite en grande série. Il est parfaitement fonctionnel et permet d'évaluer complètement ses possibilités ; en revanche, sa technique de fabrication est particulière, relevant souvent de méthodes artisanales. Son but est de tester les réactions des consommateurs avant de mettre en place la production en série, stade où il ne sera plus possible de revenir en arrière pour corriger d'éventuels petits défauts. La notion qui s'en rapproche le plus dans le domaine de l'informatique est ce que l'on appelle une *version bêta* : fourniture du logiciel à un petit nombre d'utilisateurs sélectionnés afin de tester leurs réactions, diagnostiquer les erreurs résiduelles par une utilisation en vraie grandeur, et corriger des imperfections notamment en matière d'ergonomie. Comme pour un prototype industriel, il n'est plus question à ce niveau de remettre en cause la structure fondamentale du logiciel.

Toute autre est une *maquette* : il s'agit d'une version à petite échelle du produit, destinée à montrer l'apparence future de celui-ci et à vérifier auprès du client qu'il correspondra bien à sa demande. Son but est donc de vérifier la bonne orientation du développement, avant de poursuivre plus avant. La maquette est donc par nature incomplète, et ne vérifie pas une part importante du cahier des charges. Par exemple, un appareil électronique destiné à être embarqué à bord d'une fusée doit supporter des contraintes mécaniques et de vibrations draconiennes ; cela n'empêche pas sa maquette de comporter des fils volants prêts à se détacher au moindre souffle ! Dans le domaine du logiciel, une maquette permettra de vérifier l'interface utilisateur et les fonctionnalités fournies, mais pourra ne pas vérifier des contraintes du cahier des charges telles que le temps de réponse ou le volume de données manipulées.

Ce qui différencie la maquette du prototype est donc que la maquette est par nature incomplète, et sert à valider une approche *avant* de poursuivre le développement. Au contraire, le prototype permet de vérifier qu'une première version complète du logiciel est satisfaisante, *après* développement, et permet seulement d'ajuster des défauts résiduels.

## 11.2 Maquettage rapide

Il est bien connu que c'est lorsqu'un projet est terminé qu'il faudrait le commencer. Si l'on avait pu connaître dès le début toutes les difficultés qui ne sont apparues que plus tard, on aurait pu adopter une structure plus efficace. C'est pourquoi il est parfois intéressant de développer des maquettes préliminaires destinées à explorer le domaine de problème.

La réalisation de ces maquettes est en général gouvernée par des contraintes opposées à celles du produit final : le temps de développement doit être très faible (pour ne pas trop grever le coût du développement final), il n'y aura aucune phase de maintenance, aucune contrainte de documentation ni de fiabilité ni de performance... On utilisera donc souvent des langages interprétés comme SmallTalk ou même APL.

Bien entendu, il faut absolument *jeter la maquette* dès qu'elle a permis de faire avancer suffisamment la connaissance du domaine de problème. En effet, cette forme de développement présente un risque : les développeurs peuvent avoir le sentiment d'écrire deux fois le logiciel, et réagir en tentant de «faire grossir» l'implémentation maquette au lieu de le reconcevoir entièrement. Comme les impératifs de développement de la maquette sont différents de ceux du produit définitif (les compromis de départ correspondent à des contraintes radicalement différentes, parce qu'on ne se soucie pas de maintenance sur une maquette), la structure adoptée risque d'être inapropriée.

Le programmeur hésite souvent à effacer de son disque dur ce qui représente de nombreuses heures de travail. Il faut le convaincre qu'il conservera le résultat le plus important : le *savoir-faire acquis* ; l'effort de datctylographie supplémentaire n'est pas du travail perdu, car il a servi à augmenter sa connaissance du domaine de problème. D'ailleurs, bon nombre des meilleurs logiciels du marché ont été écrits entièrement deux fois (parfois plus).

## 11.3 Maquettage progressif

Nous avons vu (paragraphe 4.3) que les grandes étapes du développement, quelle que soit la méthode utilisée, étaient constituées de niveaux horizontaux successifs. Par définition, un tel niveau est *trop complexe* pour pouvoir être réalisé directement (ou alors, c'est que l'on est arrivé à la dernière étape de raffinement). En revanche, il est possible de réaliser une version *simplifiée* de chaque module : une telle *maquette* devra se présenter vis-à-vis de l'extérieur comme le module définitif (les spécifications sont établies), mais son implémentation peut être extrêmement sommaire. Par exemple, un module destiné à gérer des millions d'enregistrements dans une base de données répartie sur un réseau peut être maquetté par un module ne sachant gérer que dix enregistrements dans un tableau en mémoire... Il n'empêche que les fonctionnalités sont présentes et permettent de vérifier la complétude des spécifications.

Nous parlerons de développement par maquettage progressif (à ne pas confondre avec le maquettage rapide) lorsque cette pratique est systématisée : à chaque étape du développement, on spécifie le module à concevoir, puis on en réalise une maquette *exécutable* : on dispose donc toujours, et dès les premières étapes, d'un programme *complet*, mais *grossier*. Au fur et à mesure du développement, les couches hautes sont remplacées par leur version définitive, mais implémentées au moyen de couches inférieures qui se trouvent elles-mêmes à l'état de maquettes, et ainsi de suite jusqu'à ce que l'ensemble du projet ait atteint l'état définitif. On pourrait donc également qualifier ces méthodes de «démaquettage progressif» : on évolue progressivement d'une version totalement «maquette» vers la version définitive. Ceci permet un passage en douceur de la spécification de haut niveau à l'implémentation, suivant l'idée de spécification progressive que l'on pratique avec SETL [Ros85].

## 11.4 Un outil indispensable : le paquetage ADPT

Pour que l'approche par maquettage progressif soit intéressante, encore faut-il que le développement de la maquette n'entraîne qu'un effort négligeable par rapport au développement du module définitif. Le paquetage ADPT est l'outil indispensable pour spécifier des conceptions incomplètes, tout en conservant la possibilité de compiler en permanence l'état de la conception. Nous utiliserons ce paquetage dans nos exemples ultérieurs ; cette section n'est qu'une présentation générale de son fonctionnement.

L'idée première de ce paquetage provient du composant du domaine public TBD_PACKAGE (TBD = *To Be Defined,* A définir), écrit par Bryce Bardin, et que l'on peut se procurer sur les serveurs du domaine public (Public Ada Library aux Etats-Unis, serveur du CNAM en France). ADPT signifie «A Définir Plus Tard». L'idée de base est de fournir un moyen de spécifier... que certaines choses ne sont pas encore spécifiées. Les perfectionnements que nous avons apportés à ce paquetage par rapport à la version originelle concernent les possibilités de trace et de spécification de durée d'exécution, qui en font un véritable outil de maquettage, y compris en ce qui concerne les propriétés temporelles. Voici la structure générale de ce paquetage :

```
package ADPT is
-- Type complètement indéfini
   type ADPT_Type is private;
   function ADPT_Fonction (Info  : String   := "";
                           Durée : Duration := 0.0)
       return ADPT_Type;
   ADPT_Valeur : constant ADPT_Type;

-- Type Enumératif
   type ADPT_Type_Enuméré is (ADPT_Valeur_Enumérée);
   function ADPT_Fonction (Info  : String   := "";
                           Durée : Duration := 0.0)
       return ADPT_Type_Enuméré;

-- Type discret
   type ADPT_Type_Discret is new ADPT_Type_Enuméré;
   function ADPT_Fonction (Info  : String   := "";
                           Durée : Duration := 0.0)
       return ADPT_Type_Discret;
   ADPT_Valeur_Discrète : constant ADPT_Type_Discret;

-- De même pour les type entiers, flottants, fixes,
-- tableau, articles, accès, étiquetés...

-- Autres déclarations
   ADPT_Condition : constant BOOLEAN := False;
   ADPT_Exception : exception;

   procedure ADPT_Procédure (Info  : String   := "";
                             Durée : Duration := 0.0);
   procedure ADPT_Actions  (Info  : String   := "";
                             Durée : Duration := 0.0);

-- Ajustement du comportement
   type ADPT_Comportement is (Ignorer, Tracer, Piéger);
   ADPT_Comportement_Courant : ADPT_Comportement:= Ignorer;

private
   ...
end ADPT;
```

Le paquetage complet (spécification et corps) est donné en annexe.

*Maquettage comportemental*

On fournit différents types ADPT, qui expriment la connaissance plus ou moins précise que l'on peut avoir du futur type. Chaque type possède une valeur, et une fonction fournissant une valeur du

type. Par exemple, si l'on n'a besoin que d'un type et que l'on n'a encore aucune idée du choix futur d'implémentation, on le fournit sous la forme :

```
type Mon_type is new ADPT_type;
```

Comme le type `ADPT_type` est en fait un type privé, aucune opération n'est encore disponible. Plus tard, on peut décider que ce type doit être un type discret. On change alors la définition en :

```
type Mon_type is new ADPT_type_discret;
```

On peut l'utiliser comme n'importe quel type discret, par exemple comme indice de tableau. On décide ensuite d'utiliser un type entier, mais l'on ne souhaite pas encore se préoccuper du problème des bornes. On définit alors :

```
type Mon_type is new ADPT_type_entier;
```

Ceci nous permet d'utiliser les opérations arithmétiques. Finalement, on définit les bornes réelles et l'on écrit :

```
type Mon_type is range 1..10;
```

De cette façon, on peut faire évoluer le programme depuis une structure très grossière jusqu'à l'implémentation finale, en faisant effectuer à chaque étape par le compilateur des vérifications sémantiques correspondant au niveau des choix d'implémentation qui ont (ou n'ont pas été) faits. Lorsque l'on pense que tout est défini dans un module, il suffit de retirer la clause `with ADPT;` et de recompiler. Si l'on a oublié quelque chose d'indéfini, le compilateur se chargera de le signaler. De même, on peut demander au gestionnaire de bibliothèque Ada la liste des unités qui dépendent encore de `ADPT` ; ceci procure une bonne idée de l'état d'avancement de la définition du projet. Lorsque cette liste est vide (et que l'on a fourni tous les corps nécessaires), on est prêt pour la première exécution en vraie grandeur !

En plus des déclarations de types, le paquetage procure une exception (à utiliser lorsque l'on ne sait pas encore quelle exception lever dans une situation donnée), une constante booléenne appelée `ADPT_condition` qui permet d'écrire :

```
if ADPT_Condition then...
```

ainsi que deux procédures : `ADPT_Procédure` et `ADPT_Actions`. La première sert à prendre la place d'un appel de procédure lorsque la vraie procédure n'a pas encore été conçue. La seconde sert à repérer un endroit où le langage exige des instructions, mais où l'on n'a pas encore décidé quoi mettre. Ne jamais utiliser une instruction **`null`** pour cela ! Il n'y aurait aucun moyen de vérifier que des actions indéfinies n'ont pas été oubliées par endroits.

On peut vouloir essayer le programme avant qu'il ne soit complètement défini. Pour cela, le comportement des sous-programmes peut être changé en modifiant la variable (publique) `ADPT_comportement`. Lorsqu'elle est mise à `Ignorer` (la valeur par défaut), les appels aux sous-programmes ADPT n'ont aucun effet. Lorsqu'elle est mise à `Tracer`, le message que l'on peut fournir (facultativement) à chacun des appels est imprimé, fournissant ainsi une trace du parcours du programme. Lorsqu'elle est mise à `Piéger`, une exception est levée si un sous-programme est appelé. Cette exception est déclarée à l'intérieur du corps de paquetage, de façon à interdire sa récupération (sauf par un traite-exception **`when others`** bien sûr). Ce dernier comportement sert à identifier des appels oubliés à des sous-programmes ADPT.

*Maquettage temporel*

Toutes les opérations fournies possèdent un paramètre supplémentaire facultatif, `Durée`, représentant la durée du traitement effectué. La valeur par défaut est bien entendu 0. Ceci permet de simuler les temps d'exécution des procédures. L'idée est de permettre de maquetter également le budget temps.

Lorsque l'on doit développer une application devant vérifier des contraintes temps réel, une façon de procéder consiste lors de la descente dans les niveaux d'abstraction à allouer à chaque procédure

un «budget temps», un temps maximum à ne pas dépasser pour toute exécution. Lors de la phase d'analyse suivante, le budget temps est réparti entre les différentes opérations utilisées. Bien entendu, comme dans un vrai budget, on gardera une petite marge de sécurité à chaque répartition, et il sera possible de transférer du temps alloué depuis une procédure qui s'avère plus rapide que prévu vers une autre qui a du mal à tenir ses contraintes. Le paramètre `Durée` permet de maquetter ce budget temps : lorsque l'on implémente réellement une opération (c'est-à-dire que l'on passe d'une maquette à une implémentation réelle, mais qui utilise des maquettes de plus bas niveau), on répartit la durée qui était allouée à la maquette entre les durées de toutes les maquettes utilisées. Comme l'ensemble reste exécutable, il est possible de vérifier, notamment en fonction de diverses valeurs d'entrée, que le budget temps n'est jamais dépassé. Bien entendu, il faudra réajuster le budget si l'on découvre des dépassements ! Mais ce n'est pas un moindre mérite de cette méthode que de permettre, *dès les couches hautes de la conception*, de vérifier au moins partiellement les propriétés temporelles du programme.

Notons enfin que l'utilisation du paquetage `ADPT` est très nettement préférable à une solution où l'on se contenterait de mettre des `PUT_LINE` (et des **`delay`**) un peu partout pour réaliser des corps prototypes. Cette dernière solution peut être extrêmement dangereuse : comment être sûr que tous les modules ont atteint leur état définitif et que l'on n'a pas oublié des `PUT_LINE` (ou pire des **`delay`** !) quelque part ? En utilisant `ADPT` en revanche, il suffit de demander au gestionnaire de programme la liste des modules qui en dépendent. S'il n'en existe plus, tout va bien. S'il en existe encore (mais que l'on pense qu'en fait tous les modules ont atteint leur état final), il suffit d'enlever les clauses **`with`** `ADPT` des modules concernés, et de recompiler. Si jamais il restait encore des appels aux fonctionnalités du paquetage, le compilateur aura l'extrême obligeance de vous les signaler...

Ce paquetage `ADPT` est un outil extrêmement puissant, à toujours posséder dans sa bibliothèque Ada. Evidemment, il n'y a aucune raison de le standardiser (au sens ISO) ; on doit plutôt l'adapter en fonction de son goût personnel et de ses habitudes de programmation.

## 11.5 Avantages et inconvénients du maquettage progressif

La notion de maquettage progressif constitue la mise en œuvre pratique de la notion de parcours horizontal du V de développement dont nous avions parlé en première partie (paragraphe. 6.4). Son avantage principal réside dans l'absence de phase d'intégration finale : le projet est en permanence vérifié dans sa globalité. Ceci donne l'impression d'avancer toujours sur du terrain solide, puisqu'on ne fait pas un pas en avant sans avoir auparavant vérifié que cette progression était cohérente et s'intégrait aux besoins du reste du projet.

Cette démarche favorise également le développement par équipes (relativement) indépendantes : on donne à un groupe la réalisation effective d'un composant dont on lui fournit la maquette. Celle-ci constitue donc une définition opérationnelle (mais réduite) de la fonctionnalité à réaliser. Elle constitue le contrat de réalisation, et il est toujours possible de comparer le comportement du module complet à celui défini par la maquette. On peut également définir (et mettre au point) les programmes de tests unitaires sur la maquette, avant d'aborder la réalisation complète ; la validation du module définitif consistera à exécuter correctement ces tests unitaires qui sont connus avant le début de la réalisation. On voit que dans ces conditions, il est aisé de conduire un développement indépendant, en ignorant le contexte global dans lequel le composant devra être intégré.

De plus, une fois définies les maquettes, chaque partie de l'équipe de développement peut s'occuper de sa partie en utilisant les corps maquettes des autres modules en cours de développement dont elle a besoin ; on évite ainsi les goulots d'étranglement qui se produisent lorsqu'une partie du projet dépend de la disponibilité d'un module particulier.

Mais le développement progressif pose également quelques difficultés. Le plus gros problème est celui du suivi. Dans un développement en «chute d'eau» classique, il est aisé d'établir des plannings de développement, avec des points précis (appelés jalons) à atteindre à des dates données. Souvent,

ces points sont vérifiés et ont des conséquences contractuelles et financières : on trouvera souvent dans des contrats de développement des clauses comme :

> *Le JJ/MM/AA, le fournisseur remettra au client les spécifications détaillées ; celui-ci disposera d'un délai d'un mois pour les accepter ou demander des modifications. Après acceptation par le client, une somme correspondant à 25% du montant du contrat sera payée.*

Avec la notion de parcours horizontal du V de développement, il n'est plus possible de définir de telles étapes. Tout au plus pourrait-on parler de pourcentage du projet implémenté «de façon définitive», par opposition aux éléments encore à l'état de maquette. Mais la définition du pourcentage et l'évaluation de l'état de la réalisation seraient fortement subjectives, et trop difficiles à définir pour permettre d'avoir des implications financières, comme le paiement d'avances sur contrat.

Indépendamment même des considérations financières, il est facile de répondre à un responsable qui s'enquiert de l'état de développement d'un projet : «Nous avons fini la conception générale, et nous sommes à environ 30 % des spécifications détaillées.» Le responsable pourra se faire une idée de l'état d'avancement du projet[1]. Il est plus difficile de répondre : «Nous avons implémenté le troisième niveau d'abstraction, et nous travaillons à l'implémentation du quatrième avec certaines maquettes de cinquième niveau opérationnelles.»

Une échappatoire consiste à définir une équivalence entre le développement de certains plans d'abstraction et les phases habituelles du développement, en particulier lorsque l'on dispose d'une méthode bien formalisée. On peut alors spécifier par contrat (en utilisant les critères de HOOD) :

> *La fourniture complète de toutes les rubriques H1 et H2 de la conception pour les objets de premier et deuxième niveaux constituera la conception générale, et donnera lieu au paiement de l'avance après acceptation par le client.*

Mais ceci ne constitue qu'une tentative de «raccrocher» le processus de développement progressif aux éléments classiques de la chute d'eau, dont on cherche au contraire à se séparer. On voit donc que le maquettage implique une remise en cause complète de tout le processus de gestion de projet, et l'on ne dispose pas encore de cadre bien admis permettant d'assurer ce suivi.

Cette difficulté à définir l'état d'avancement d'un projet par rapport à la réalisation finale peut paraître anecdotique, pour ne pas dire ridicule, aux développeurs individuels ou à ceux qui n'ont pas eu à développer des projets en milieu industriel ; elle est en fait suffisamment forte pour interdire totalement le développement progressif dans certains contextes.

## 11.6 Exercices

1. Analyser en composition le système de contrôle d'une centrale nucléaire : surveillance de la température des réacteurs, et déclenchement d'alarmes en cas de valeurs hors normes. On se limitera au premier niveau d'abstraction (!), mais l'on fournira une maquette opérationnelle.
2. Ecrire une maquette opérationnelle des fonctionnalités de haut niveau d'un système de gestion de bases de données, mais dont l'implémentation ne gère que quelques données dans un tableau.
3. Expliquer pourquoi le maquettage progressif, au contraire du maquettage rapide, doit s'effectuer dans le langage de l'implémentation définitive.

[1] Idée souvent considérablement faussée par rapport à l'état réel du projet, surtout si la mise au point se révèle difficile, ou que des difficultés apparaissent lors de l'intégration. Mais ceci est une autre histoire...

# Troisième partie

# Composants logiciels

La réutilisation est un élément principal de l'abaissement des coûts de développements logiciels ; il n'est plus acceptable de nos jours de récrire éternellement les mêmes morceaux de code. L'électronicien qui souhaite construire un circuit logique ne reconçoit pas à chaque fois les portes dont il a besoin : il achète des *composants* qui lui fournissent les fonctions logiques nécessaires. C'est d'ailleurs grâce à cette notion de composant réutilisable, à faible coût car produit à un grand nombre d'exemplaires, que l'industrie électronique a pu prendre le développement que nous lui connaissons aujourd'hui. Or il faut bien reconnaître que cette démarche est actuellement quasi inexistante dans le monde du logiciel. Combien de programmes de tri sont-ils recodés chaque année ? Il est temps que l'industrie du logiciel redécouvre les méthodes qui ont fait le succès de bien d'autres branches industrielles... avec quelques dizaines d'années de retard !

Il faut donc apprendre à développer en produisant et en utilisant des *composants logiciels réutilisables*. Mais là non plus, la bonne volonté n'est pas suffisante : il existe des techniques et des méthodes nécessaires à la bonne mise en place d'une politique de réutilisation dans l'entreprise.

Cette troisième partie va présenter ces techniques et méthodes pour le développement de composants logiciels réutilisables, et nous verrons qu'encore une fois, Ada fournit l'outil qui supporte *activement* ces méthodes.

# 12

# En guise d'introduction...

Ecrire un bon composant logiciel est loin d'être trivial. Avant de discuter de toutes les contraintes que cela impose, nous prendrons un exemple (un peu caricatural, peut-être, encore que...) et suivre les péripéties qui nous mènent de la vue naïve d'une fonctionnalité souhaitée à un réel composant logiciel.

Nous voulons réaliser une fonction `Permute` qui permute le début et la fin d'une chaîne de caractères, c'est-à-dire qui nous permette d'obtenir la chaîne de caractères «XYZABCD» à partir de la chaîne «ABCDXYZ». Nous exprimons cette spécification comme :

```
procedure Permute(La_Chaîne : in out String;
                  Coupure   : in     Integer);
```

Nous donnons le corps à écrire à un stagiaire[1] qui a une certaine expérience d'autres langages de programmation, mais pas tellement d'Ada et qui écrit :

```
procedure Permute (La_Chaîne : in out String;
                   Coupure   : in     Integer) is
   Pos_Char : Integer := 1;
begin
   for I in Coupure+1..La_Chaîne'LENGTH loop
      La_Chaîne (Pos_Char) := La_Chaîne (I);
      Pos_Char := Pos_Char + 1;
   end loop;

   for I in 1..Coupure loop
      La_Chaîne (Pos_Char) := La_Chaîne (I);
      Pos_Char := Pos_Char + 1;
   end loop;
end Permute;
```

Dès le premier essai, il s'aperçoit que ceci ne peut pas fonctionner, car le programme de test :

```
with Ada.Text_IO; use Ada.Text_IO;
with Permute;
procedure Test is
   S : String(1..7) := "ABCDXYZ";
begin
   Permute (S, 4);
   Put_Line (S);
end Test;
```

imprime «XYZDXYZ» ! En effet on va écraser le début de la chaîne alors que l'on en a encore besoin. Le stagiaire en déduit que la chaîne de sortie doit être différente de la chaîne d'entrée, et récrit la procédure comme suit :

[1] L'écriture de ce genre de petites fonctions est souvent utilisée pour occuper les stagiaires et les éloigner des points critiques du projet...

```
procedure Permute (Entrée  : in  String;
                   Sortie  : out String;
                   Coupure : in  Integer) is
  Pos_Char : Integer := 1;
begin
  for I in Coupure+1..Entrée'LENGTH loop
    Sortie (Pos_Char) := Entrée (I);
    Pos_Char := Pos_Char + 1;
  end loop;

  for I in 1..Coupure loop
    Sortie (Pos_Char) := Entrée (I);
    Pos_Char := Pos_Char + 1;
  end loop;
end Permute;
```

Le stagiaire procède alors à quelques tests et vient annoncer fièrement que «ça marche». Que peut-on dire alors de cette solution ?

> La spécification en a été changée à cause d'un problème d'implémentation.
> Le composant ne fonctionne que pour les chaînes dont la borne inférieure d'index est 1.
> Il ne fonctionne pas correctement si les variables `Entrée` et `Sortie` n'ont pas la même taille.
> Il risque de ne pas fonctionner correctement si la variable associée au paramètre `Sortie` est la même que celle associée au paramètre `Entrée`.
> La signification exacte de la variable `Coupure` n'a jamais été spécifiée. Désigne-t-elle le dernier caractère de la première chaîne ou le premier de la deuxième ?
> Le comportement de la procédure lorsque `Coupure` n'appartient pas à l'intervalle des bornes de la chaîne à couper n'a jamais été spécifié.

Le premier point est extrêmement grave : la première tâche de celui qui écrit un composant est de réaliser le composant demandé, pas un autre qui l'arrange mieux ! Ceci dit, une spécification n'est pas sacro-sainte non plus, et il se peut que l'implémenteur y découvre une faiblesse ; il doit alors proposer une nouvelle spécification, et n'implémenter qu'après accord du client. Ici, la spécification initiale ne prévoit qu'une permutation de la chaîne sur elle-même ; cela oblige à utiliser une variable pour l'appel, même si la chaîne intervient en fait dans une expression chaîne plus vaste.

Le deuxième point est une erreur classique des gens qui ne sont pas habitués à Ada : rien n'impose qu'un tableau soit indexé à partir de 1, *et ce ne sera en général pas le cas si l'on appelle le composant en donnant comme paramètres réels des tranches de tableau*. Une bonne utilisation des attributs `'First` et `'Last` permet de remédier aisément à ce problème.

> Ada permet de prendre des tranches de tableaux (unidimensionnels), qui sont elles-mêmes des tableaux. Ainsi, `S(5..10)` représente la tranche du tableau `S` commençant au 5e élément et se terminant au 10e ; la borne inférieure de ce tableau est alors 5, et sa borne supérieure 10. Une tranche de variable est une variable, et peut donc figurer en partie gauche d'une affectation :
>
> ```
> S(5..10) := (others => 0);
> ```
>
> remet les éléments 5 à 10 du tableau S à 0.

Les points 3 et 4 ne semblent pas dramatiques *a priori* : il semblerait qu'il suffise de prévenir l'utilisateur du sous-programme que les deux variables doivent être différentes et de même longueur. Il serait possible de vérifier l'égalité des longueurs et de lever une exception si ce n'était pas le cas ; en revanche, il n'existe aucun moyen de s'assurer que les variables fournies sont différentes. Pire : le composant peut fonctionner parfaitement, même avec deux fois la même variable, sur une implémentation qui passe les paramètres par copie, et donner des résultats faux si les paramètres sont passés par adresse. Le bon fonctionnement du composant repose donc sur des éléments extérieurs et incontrôlables.

Les deux derniers points proviennent clairement d'un manque de spécification au départ. Cela ne signifie pas que la solution soit mauvaise en soi ; ce que nous voulons dire, c'est que le comportement doit être exactement spécifié *a priori*, et non résulter des hasards de l'implémentation. Nous voyons hélas trop souvent l'application de la définition suivante :

*Spécification : description du comportement constaté d'une implémentation !*

Nous décidons donc que `Coupure` désigne le dernier caractère de la première chaîne, et que s'il n'appartient pas à l'intervalle des bornes, la chaîne fournie doit être identique à la chaîne en entrée. Une deuxième itération du problème conduit à améliorer la spécification ainsi :

```
function Permute (La_Chaîne : String;
                  Coupure   : Integer) return String;
```

De plus, nous expliquons au stagiaire la possibilité de déclarer des variables locales dont la taille dépend des paramètres d'entrée. Cette fois-ci, il produit l'implémentation suivante :

```
function Permute (La_Chaîne : String;
                  Coupure   : Integer) return String is
   Résultat : String (La_Chaîne'Range);
   Pos_Char : Integer := Résultat'First;
begin
   if Coupure not in La_Chaîne'Range then
      return La_Chaîne;
   end if;

   for I in Coupure+1..La_Chaîne'Last loop
      Résultat (Pos_Char) := La_Chaîne (I);
      Pos_Char := Pos_Char + 1;
   end loop;

   for I in La_Chaîne'First .. Coupure loop
      Résultat (Pos_Char) := La_Chaîne (I);
      Pos_Char := Pos_Char + 1;
   end loop;
   return Résultat;
end Permute;
```

Cette version paraît nettement plus satisfaisante : le cas où `Coupure` n'appartient pas à l'intervalle est traité conformément aux spécifications, et il n'y a plus de dépendance à une quelconque valeur des bornes. Pourtant, il y a encore un cas de figure qui ne fonctionne pas correctement. Si nous écrivons :

```
    S : String(Integer'Last-1 .. Integer'Last) := "AB";
begin
   S := Permute (S, S'First);
end;
```

la fonction `Permute` lèvera l'exception `Constraint_Error` ! En effet, l'instruction

```
Pos_Char := Pos_Char + 1;
```

sera effectuée pour une valeur de `Pos_Char` déjà égale à `Integer'Last`, et lèvera donc l'exception. Il faut faire un cas particulier si la borne supérieure de `La_Chaîne` est `Integer'Last`. Et comme l'expression `Coupure+1` apparaît également, il faut aussi faire un cas particulier si `Coupure` prend la valeur `Integer'Last`... Voilà un composant qui commence à devenir bien compliqué, juste pour un cas qui a toutes chances de ne jamais se produire en pratique ! La tentation est forte de laisser tomber, au besoin de documenter «qu'il ne faut pas appeler la fonction avec une chaîne dont la borne supérieure est `Integer'Last`»... C'est ainsi que des logiciels qui fonctionnent depuis des années deviennent parfois fous sur des cas particuliers.

La nécessité de fiabilité conduit donc à des complications supplémentaires. Ceci est fréquent lorsque l'on a écrit un composant sans tenir compte des cas limites, et que l'on essaie ensuite de le faire fonctionner même pour ces cas-là. Mais avons-nous utilisé toutes les possibilités du langage ? N'y aurait-il pas une autre implémentation qui traiterait tous les cas de façon égale ? Bien entendu, la réponse est oui[1]. Les tranches de tableaux constituent un outil remarquablement puissant, car leur sémantique dans les cas limites est très bien définie... ce qui veut dire que c'est le compilateur qui s'embête à notre place. Nous pouvons récrire le corps de notre fonction comme ceci :

[1] Sinon, nous n'aurions pas choisi cet exemple...

```
function Permute (La_Chaîne : String;
                  Coupure   : Integer) return String is
begin
   if Coupure not in La_Chaîne'Range then
     return La_Chaîne;
   else
     return La_Chaîne (Coupure+1..La_Chaîne'Last) &
            La_Chaîne (La_Chaîne'First .. Coupure);
   end if;
end Permute;
```

Cet énoncé traite correctement tous les cas de figure normaux, *y compris celui où Coupure vaut La_Chaîne'Last*, grâce à la sémantique d'Ada qui autorise les chaînes vides. Il est plus proche de la spécification, et il est même plus efficace : en spécifiant des opérations de tranches de tableau, on permet au compilateur d'utiliser des opérations de déplacement de chaînes d'octets, ce qu'il n'aurait pu faire quand on décrit le transfert caractère par caractère. Est-il parfait ? Non, car il existe encore un cas où il lèvera `Constraint_Error` : si `La_Chaîne'Last = Coupure = Integer'Last` ! Car alors, le calcul de `Coupure+1` débordera. Mais l'on peut remarquer que dans ce cas (ainsi que dans tous ceux où `Coupure = La_Chaîne'Last`), la chaîne à renvoyer est précisément la chaîne d'origine. On peut donc écrire :

```
function Permute (La_Chaîne : String;
                  Coupure   : Integer) return String is
begin
   if Coupure not in La_Chaîne'First .. La_Chaîne'Last-1
then
     return La_Chaîne;
   else
     return La_Chaîne (Coupure+1..La_Chaîne'Last) &
           La_Chaîne (La_Chaîne'First .. Coupure);
   end if;
end Permute;
```

Sommes-nous au bout de nos peines ? Non ! Il existe encore un cas de figure qui lève `Constraint_Error`. Le lecteur est prié de réfléchir avant de lire la suite... L'utilisateur a le droit de déclarer :

```
S : String(1..Integer'First);
```

Il s'agit bien sûr d'une chaîne vide, mais sa borne supérieure est effectivement `Integer'First`. Si notre fonction est appelée sur un tel monstre, l'expression `La_Chaîne'Last-1` lèvera `Constraint_Error`. Faut-il introduire une complication supplémentaire pour tester ce cas qui relève clairement de la pathologie ? Mais après tout, *faut-il vraiment tester les autres cas ?* Puisque le compilateur fait les tests pour nous, autant l'utiliser ! Nous pouvons écrire :

```
function Permute (La_Chaîne : String;
                  Coupure   : Integer) return String is
begin
   return La_Chaîne(Coupure+1..La_Chaîne'Last) &
          La_Chaîne (La_Chaîne'First .. Coupure);
exception
   when Constraint_Error =>
     return La_Chaîne;
end Permute;
```

Cette version, qui est satisfaisante du point de vue de la sécurité, présente encore un défaut : la borne inférieure de la chaîne renvoyée est `Coupure+1`. La chaîne subit donc un «glissement» vers la droite. Il y a peu de chances que l'utilisateur le remarque, sauf s'il utilise la fonction dans un contexte qui requiert une borne inférieure précise. En fait, nous n'avons pas spécifié la valeur de la borne inférieure de la valeur retournée ; les deux comportements possibles qui viennent à l'esprit sont soit de prendre la même valeur que la chaîne d'origine, soit de forcer la borne inférieure à 1. Mais quelle que soit la solution adoptée, il faut que l'invariant soit garanti. Or notre solution actuelle, non seulement ne correspond à aucune de ces solutions, mais de plus n'a pas une sémantique uniforme puisque dans les cas où l'on passe par le traite-exception, la borne inférieure

n'est pas modifiée. Décidons de toujours renvoyer en sortie les mêmes bornes qu'en entrée (cela paraît logique), et utilisons une conversion de sous-type pour remettre les bornes désirées :

```
function Permute (La_Chaîne : String;
                  Coupure   : Integer) return String is
   subtype Ajustement is
     String (La_Chaîne'First ..
             La_Chaîne'First + La_Chaîne'Last-Coupure-1);
begin
   return Ajustement(La_Chaîne(Coupure+1..La_Chaîne'Last))
        & La_Chaîne (La_Chaîne'First .. Coupure);
exception
   when Constraint_Error => return La_Chaîne;
end Permute;
```

On peut encore discuter de cette spécification. Il semble inutile d'autoriser des valeurs négatives pour `Coupure` ; pourquoi alors ne pas lui donner le sous-type `Natural` au lieu de `Integer` ? Certes, mais des valeurs négatives peuvent résulter de calculs intermédiaires, et alors il ne faudrait pas non plus autoriser de valeurs supérieures à `La_Chaîne'Last+1`. Pour être cohérent, il faut donc soit autoriser n'importe quelle valeur pour `Coupure` (exemple précédent), soit n'autoriser qu'un débordement d'une seule valeur (et lever `Constraint_Error` sinon). Nous en resterons donc à la version précédente qui a le mérite de l'uniformité et d'une sémantique claire des cas limites.

| **Test n°** | **(1)** | **(2)** | **(3)** | **(4)** |
|---|---|---|---|---|
| 1 (procédure) | 2,01 | 2,14 | 0,92 | 0,68 |
| 2 (fonction) | 3,70 | 3,28 | 5,23 | 1,74 |
| 3 (utilise "&", test) | 3,55 | 3,12 | 4,06 | 1,40 |
| 4 (meilleur test) | 3,55 | 3,12 | 4,12 | 1,43 |
| 5 (exception) | 3,68 | 3,24 | 4,06 | 1,48 |
| 6 (ajustement des bornes) | 3,97 | 3,46 | 4,73 | 1,43 |

(1) Compilateur 1, sans optimisation
(2) Compilateur 1, avec optimisation
(3) Compilateur 2, sans optimisation
(4) Compilateur 2, avec optimisation

**Figure 26 :** Performances des différentes versions de Permute

Le souci d'obtenir un composant tellement protégé ne va-t-il pas avoir des conséquences funestes sur les performances ? Nous avons effectué quelques mesures rapides sur les différentes versions présentées ici, avec deux compilateurs (un bon marché et un plus coûteux), avec et sans optimisation. Les temps d'exécution sont consignés dans le tableau de la figure 26.

Il apparaît clairement que le passage d'une procédure à une fonction retournant un type non contraint induit un surcoût ; mais ceci correspond également à un accroissement important de l'utilisabilité du composant. En revanche, *à condition de compiler avec les optimisations*, l'augmentation de sécurité des dernières versions n'induit aucun surcoût mesurable en termes d'efficacité. Ceci signifie que le surcroît de précision apporté dans la définition a été mis à profit par le compilateur pour optimiser le code généré. Il n'y a donc pas de raison de supposer *a priori* qu'une version sécurisée d'un composant doive être nécessairement moins efficace.

Que peut-on déduire de cet exemple ?

Qu'il y a un abîme entre un composant qui marche «presque» toujours et un composant qui marche toujours.

Qu'une bonne connaissance du langage permet de trouver des solutions plus simples, plus élégantes, plus lisibles *et* plus efficaces.

Qu'une fois que l'on a trouvé la bonne solution, elle paraîtra évidente à ceux qui la reliront. Ce qui ne signifie pas qu'elle était simple à trouver la première fois.

Qu'il faut un état d'esprit extrêmement rigoureux pour envisager tous les cas limites et définir une sémantique acceptable dans tous les cas.
Qu'il ne faut pas avoir d'*a priori* sur les constructions supposées être plus (ou moins) efficaces que d'autres tant qu'on n'a pas eu l'occasion d'effectuer des mesures.
... et qu'il ne faut pas confier l'écriture de composants logiciels à des stagiaires.

# 13

# Qu'est-ce qu'un composant logiciel ?

## 13.1 Définition

Qu'appelle-t-on composant logiciel ? Ce terme est né de l'analogie avec les composants matériels. Mais quest-ce qu'un composant matériel ? Dans le domaine voisin de l'électronique, nous trouvons des composants de base : résistances, condensateurs, transistors... On trouve aussi des composants de plus haut niveau : alimentations électriques, amplificateurs. Certains appareils (y compris des ordinateurs !) sont même des produits autonomes qui sont *utilisés* comme composants dans des systèmes de taille supérieure. Qu'y a-t-il de commun entre ces différentes sortes de composants qui justifie l'emploi d'un même terme et qui soit susceptible de s'appliquer aux composants logiciels ?

La première caractéristique est que ces éléments n'ont pas *a priori* d'utilité en eux-mêmes ; ils ne servent qu'à la réalisation d'ensembles plus vastes, résultant de la *composition* de ces éléments.

Une deuxième caractéristique est que le même composant peut être utilisé pour réaliser des systèmes très différents, n'ayant aucun rapport entre eux. Ce sont les mêmes résistances électroniques qui servent pour un lecteur de cassette, le contrôleur d'un four à micro-ondes ou le guidage d'un missile.

Une troisième caractéristique, conséquence des précédentes, est que le concepteur d'un composant ne sait pas dans quel contexte celui-ci sera utilisé. Il ne peut donc s'appuyer sur aucun élément extérieur à la nature même de son composant pour guider sa conception[1]. En revanche, c'est l'utilisateur du composant qui devra subir les contraintes propres à ce dernier. Ainsi par exemple, le fabricant d'un ventilateur prévoira quatre trous pour le fixer, également répartis sur la circonférence. Pourquoi également répartis ? Parce qu'en l'absence de connaissance sur les contraintes de fixation de son composant, il a dû prendre une décision arbitraire. Si la position des trous ne satisfait pas l'utilisateur, ce sera à ce dernier de mettre des cales, des équerres, etc., pour pouvoir utiliser le composant. Noter que l'utilisateur aurait pu avoir la réaction de dire qu'en l'absence d'un ventilateur répondant exactement à ses spécifications, il en faisait réaliser un sur mesure. Cela fait longtemps que l'on sait dans le domaine du matériel qu'il est plus économique d'adapter ses exigences aux composants existants.

Parfois, un composant peut être conçu dans un but spécifique, mais généralisé bien au-delà de son but initial. C'est ainsi que les microprocesseurs furent initialement développés pour les besoins de la conquête spatiale... domaine qui ne représente plus qu'une minuscule partie de leur marché actuel.

Enfin, un composant doit être standardisé, ou tout au moins suffisemment stable pour rester compatible dans le temps. La standardisation des interfaces, en permettant l'évolution technologique sans remise en cause des utilisations, en favorisant l'apparition d'un marché concurrentiel et en

[1] Cette phrase faisait partie de la première version de cet ouvrage, publié quelques mois avant l'accident d'Ariane 501, dû précisément au non-respect de ce bon principe.

entraînant la standardisation des éléments annexes,[1] est pour beaucoup dans l'essor de l'industrie électronique.

A partir de ces exemples, nous définirons donc un composant logiciel comme une entité logicielle *réutilisable*, définissable *indépendamment de son contexte d'utilisation*, et dont l'interface est stable et *figée*.

## 13.2 Contraintes supplémentaires pour les composants logiciels

Cette définition implique des contraintes supplémentaires pour un composant logiciel par rapport à un composant qui fait partie d'un projet donné.

Pour être réutilisable, le composant doit être général, c'est-à-dire correspondre à un besoin partagé par de nombreux développements. Ceci nécessite un effort d'abstraction supplémentaire au niveau de la conception, afin de rendre le composant indépendant de toute utilisation particulière. Il doit également être *complet,* mais non *surabondant* : il doit offrir toutes les fonctionnalités indispensables, et seulement celles-là. De plus, il doit être fortement cohérent : il ne doit pas être possible de le couper en deux tout en maintenant un faible couplage entre les deux parties. Une autre façon de formuler cette exigence est de dire qu'un composant doit traiter entièrement un seul aspect d'un problème. Ceci ne s'obtient pas en général du premier coup, et c'est au fil des réutilisations que le composant «décante» pour arriver à un juste équilibre dans les fonctionnalités offertes.

Le principe de l'indépendance du module implique que sa sémantique doit être autonome et définie. La connaissance d'une documentation minimale doit être suffisante pour l'utiliser. L'utilisateur doit pouvoir avoir vis-à-vis du composant une attitude de «tu te débrouilles, je ne veux pas le savoir».

Enfin le composant doit être robuste, notamment pour les cas limites. Un composant spécifique ne sera pratiquement jamais utilisé dans toute l'étendue de ses capacités, ce qui peut d'ailleurs permettre à des erreurs de rester cachées pendant longtemps, parfois même éternellement. Un composant réutilisé de nombreuses fois a toutes les chances de subir tous les cas de figure possibles, et doit donc pouvoir faire face à toutes les circonstances.

Terminons par une remarque : dans tout projet, il existe des parties nécessairement spécifiques, notamment celles qui sont bâties sur les composants logiciels et les font «jouer» pour obtenir le résultat escompté. Il existe donc forcément des composants qui ne correspondent pas aux critères ci-dessus, et ce serait une erreur d'imposer à un projet de *tout* faire sous forme réutilisable.

## 13.3 Coût de développement des composants

Compte tenu des contraintes que nous avons exposées, il n'est pas étonnant que le coût de développement d'un composant logiciel réutilisable soit supérieur à celui d'un composant spécifique. Le composant spécifique a en effet le droit de connaître (plus ou moins) par qui et comment il a été appelé ; il peut faire des hypothèses simplificatrices sur ses paramètres et il n'est pas toujours nécessaire de le rendre absolument fiable, face aux cas limites notamment.

On estime empiriquement que le développement d'un composant réutilisable coûte environ deux à trois fois plus cher que son équivalent spécifique. [Fav91] a publié une étude où il présente des mesures tenant compte non seulement du surcoût pour «écrire réutilisable», mais également du surcoût pour *utiliser* des composants (y compris la phase d'apprentissage) ; sur différents composants, le nombre de réutilisations pour amortir le développement varie de 1,33 à près de 13 dans un cas particulier ! Pour la plupart des exemples, le seuil de rentabilité est inférieur à 4 réutilisations. Cette constatation fondamentale a des conséquences très importantes :

[1] C'est parce que l'espacement des pattes des circuits est standardisé à 1/10" que les supports de circuits ont pu être standardisés et se développer.

Le surcoût n'est pas *énorme*, car il sera amorti au bout de quelques réutilisations du composant logiciel, une situation que nous envient ceux qui fabriquent des composants matériels !

Le surcoût est cependant suffisamment important pour ne pouvoir être justifié au niveau d'un projet individuel. Un chef de projet qui prendrait sur lui d'exiger de ses équipes de développement qu'elles définissent systématiquement les modules sous forme de composants réutilisables serait assuré de ne pas tenir ses budgets.

Une politique de réutilisation ne peut donc être rentable que si elle est établie à un échelon supérieur à celui du projet individuel : c'est au niveau de l'entreprise qu'une telle politique doit être mise en place, afin de permettre le partage des composants entre plusieurs projets. Il est donc nécessaire de mettre en place une structure transversale, coiffant les projets, pour la réutilisation. Des investissements supplémentaires, non affectables directement à un projet, doivent être consacrés à la réutilisation. Nous développerons ce point par la suite.

## 13.4 Exercices

1. Ecrire la spécification d'un composant d'interface graphique de base (définissant les coordonnées d'écran et des fonctions élémentaires, comme d'allumer un pixel dans une certaine couleur). Le composant doit être indépendant de la résolution et du nombre de couleurs disponibles à l'écran.
2. Spécifier complètement un composant permettant de traiter des dates et d'effectuer des calculs entre dates sur toute la période historique (de -3000 à +2000). Traiter soigneusement les cas exceptionnels, en particulier le passage du calendrier julien au calendrier grégorien.
3. Prendre plusieurs projets (comme des projets de fin d'étude d'élèves précédents) et analyser les parties communes qui auraient dû faire appel à des composants réutilisables.

# 14

# Organisation et classifications des composants logiciels

Il existe plusieurs façons de classer les composants logiciels. Nous avons déjà vu que les paquetages pouvaient être subdivisés en *collections de données*, *collections de sous-programmes*, *machines abstraites*, *types de données abstraits* et *gestionnaires de données*. Nous allons présenter maintenant d'autres critères d'organisation. Ces différentes classifications sont orthogonales : deux composants appartenant à la même classe selon un critère pourront parfaitement appartenir à des classes différentes selon un autre.

L'intérêt des classifications est double : d'abord, elles servent de base à la documentation des composants et aux moyens de recherche d'un composant répondant à un comportement souhaité ; ensuite, elles permettent une normalisation des espèces de composants. Un composant qui apparaîtrait «hors norme», inclassifiable, aurait toutes les chances de souffrir d'erreurs de conception, et d'être un mauvais candidat à la réutilisation.

## 14.1 Taxonomie comportementale

Booch a défini dans son second livre [Boo87] une taxonomie des composants fondée sur les propriétés de leur comportement. Cette taxonomie a depuis été reprise et complétée par Berard [Ber87].

### *14.1 .1 Comportement parallèle*

Le premier critère de classification se fonde sur le comportement du composant en cas d'utilisation par plusieurs tâches en parallèle. Le composant peut être :

*Séquentiel*. Le bon fonctionnement du composant n'est garanti qu'en cas d'utilisation par une seule tâche.

*Gardé*. L'objet fournit un moyen d'exclusivité d'accès (sémaphore), mais n'en vérifie pas le bon usage. Le respect du protocole en cas d'utilisation par plusieurs tâches est à la charge du client.

*Parallèle* (ou *protégé* selon Berard). L'objet assure lui-même l'exclusivité d'accès en sérialisant des demandes concurrentes. Il ne peut y avoir qu'une seule tâche à la fois dans le composant.

- *Multiple*. Le composant gère une politique de type «lecteurs-écrivains» pour assurer un comportement correct et un parallélisme maximal en cas d'accès multiple.

A première lecture, on peut se demander pourquoi tous les composants n'appartiennent pas à la catégorie «multiple». La raison en est que le gain en fiabilité se paie par l'efficacité. Un composant multiple est totalement inutile pour un programme purement séquentiel...

Nous déconseillerons fortement l'écriture de composants gardés, puisqu'ils ne peuvent assurer eux-mêmes leur propre sémantique : leur bon fonctionnement dépend du respect du protocole par l'utilisateur. Leur seul avantage est de permettre d'enchaîner plusieurs opérations sans relâcher le sémaphore. Un tel comportement est cependant parfois nécessaire pour fournir des fonctionnalités de base, uniquement destinées à écrire des fonctionnalités de plus haut niveau qui seront, elles, protégées ou multiples.

On obtient assez facilement un composant parallèle à partir d'un composant séquentiel en rajoutant un paquetage comportant une tâche ou un objet protégé assurant la sérialisation des accès. Par conséquent nous considérerons que les deux comportements fondamentaux (et devant faire l'objet de codages distincts) sont le séquentiel et le multiple.

Berard a apporté une précision supplémentaire, en divisant chacun des comportements parallèles en deux, selon que la protection s'effectue au niveau des *opérations* ou au niveau des *objets*. On distingue ainsi le cas des composants parallèles au niveau des opérations, où deux tâches effectuant des opérations sur des objets différents sont sérialisées, et les composants parallèles au niveau des objets, où la sérialisation n'intervient que si les deux tâches effectuent des opérations sur le même objet.

Noter qu'Ada est par définition un langage réentrant. Dans le cas d'un type de donnée abstrait, un comportement séquentiel autorise tout de même plusieurs tâches à travailler simultanément sur des objets distincts. Ce n'est exclu que s'il existe une information globale commune à deux objets distincts, et donc que ceux-ci ne sont pas de «purs» types de données abstraits.

### *14.1 .2 Contenance*

Le deuxième critère de classification se fonde sur la capacité du composant de manipuler un nombre quelconque d'entités. Ce critère n'est en principe significatif que pour les gestionnaires de données. Le composant peut être :

*Limité*. La taille (ou la contenance) de l'objet est fixée pour toute sa durée de vie.
*Non limité*. La taille (ou la contenance) de l'objet peut varier au cours de sa vie, et n'est limitée que par la mémoire disponible sur la machine.
*Borné* (Berard). Le composant est doté d'une taille maximale lors de sa création. La taille utile est variable, mais ne peut envahir toute la mémoire.

> Noter que Booch utilise le terme «statique» pour caractériser la taille de l'objet, ce qui est trop limitatif. Ada permet de créer des objets dont la taille est déterminée lors de leur élaboration, mais néammoins dynamique.

Cet aspect concerne les choix fondamentaux de représentation. Typiquement, une liste limitée sera représentée par un tableau, alors qu'une liste non limitée sera représentée par une chaîne de variables pointées. La forme bornée est adaptée au cas où l'on connaît *a priori* le nombre maximum d'éléments, tout en sachant qu'il est très peu probable que ce nombre soit atteint. Elle peut correspondre à un type à discriminant. Ici encore, une plus grande généralité se paie généralement en termes de rapidité d'exécution.

### *14.1 .3 Récupération*

Le troisième critère de classification se fonde sur les propriétés de la gestion de la mémoire par le composant. Ce critère est fondamental pour les structures de données, mais peut être applicable à

d'autres sortes de composants. Il n'a de signification que pour les composants de type «non limité» selon le critère précédent. La mémoire utilisée par le composant peut être :

*Non gérée*. Le composant ne se préoccupe pas de la récupération de l'espace mémoire devenu disponible. La mémoire ne sera récupérée que si l'exécutif a prévu un dispositif «ramasse-miettes» (*garbage collector*), sinon elle sera perdue.

*Gérée*. Le composant gère et récupère la mémoire libérée.

- *Contrôlée*. Le composant, de type séquentiel, gère et récupère la mémoire libérée. Bien que séquentiel, le processus de récupération de la mémoire est tout de même protégé contre les accès simultanés, de façon à garantir un comportement correct en cas d'utilisation par plusieurs tâches travaillant sur des objets distincts.

Ici encore, on pourrait espérer que les composants non limités soient du type géré ou contrôlé. Cependant, le surcoût peut être important, et non justifié dans certains cas (par exemple si les structures ne décroissent jamais).

> Les types contrôlés rendent l'écriture de composants contrôlés beaucoup plus facile en Ada 95 qu'en Ada 83. Ces types permettent de définir une procédure de finalisation qui est appelée automatiquement lors de la destruction d'un objet; il est alors possible de rendre automatiquement tout l'espace mémoire associé.

### *14.1 .4 Parcours*

Ce critère de classification qui figure dans [Boo87] n'en est pas vraiment un ; sa présence provient de ce que Booch n'a pas considéré les gestionnaires de données comme une classe à part de type de donnée abstrait. Nous le mentionnons ici par complétude.

*Non itérateur*. Le composant ne comporte pas d'itérateur.

- *Itérateur*. Le composant comporte un itérateur.

Si l'on sépare les structures de données, il est évident que celles-ci, et seulement celles-ci, doivent avoir un itérateur... sauf exception (les piles sont typiquement des structures de données sans itérateur).

## 14.2 Relations entre composants

Tous les composants ne naissent pas égaux. Certains sont totalement généraux et indépendants de leur contexte, d'autres très spécifiques. Nous allons maintenant caractériser les composants en termes de généralité d'usage.

### *14.2 .1 Composants indépendants*

Ces composants sont caractérisés par le fait qu'ils ne référencent que les éléments standard du langage : types et paquetages prédéfinis. Ils sont donc *a priori* portables sans problèmes. Ces composants sont l'analogie logicielle des composants électroniques les plus élémentaires : résistances, condensateurs, transistors. Ils peuvent être combinés entre eux sans difficulté. Autant que possible, on essayera de rendre les composants indépendants, car ce sont les plus généraux et les plus portables. La plupart des composants «classiques» commercialement disponibles appartiennent à cette catégorie : gestionnaires de données, bibliothèques... L'utilisation de composants généraux est en principe autorisée sans contrainte, c'est-à-dire que le choix de tels composants ne constitue pas une décision de conception engageant le projet.

On pourra souvent rendre indépendant un composant qui semble dépendre d'un type de donnée particulier en rendant le composant générique et en important le type en question en tant que

paramètre générique. Supposons par exemple une fonction qui dépendrait d'un type `Vecteur` et qui fournirait la somme des éléments :

```
with Définition_Vecteurs; use Définition_Vecteurs;
function Somme (Item : Vecteur) return Float;
```

On pourrait rendre sa spécification indépendante en la transformant en générique comme ceci :

```
generic
   type Composant is private;
   with function "+" (Gauche, Droite : Composant)
     return Composant is <>;
   type Index is (<>);
   type Vecteur is array (Index range <>) of Composant;
function Somme (Item : Vecteur) return Composant;
```

## *14.2 .2 Sous-systèmes*

La notion de sous-système a été développée par Booch [Boo87]. Elle correspond à la nécessité de diminuer la complexité d'un composant en le décomposant hiérarchiquement en un certain nombre de modules. L'un d'entre eux joue le rôle de *point d'entrée*, et c'est le seul utilisé par les entités clientes du composant. Les autres sont *spécifiques* du composant. Il s'agit donc bien d'une décomposition hiérarchique du composant, et non de l'utilisation par un composant d'autres composants généraux. Bien sûr, rien n'empêche les éléments du sous-système d'utiliser également des composants réutilisables, mais ceux-ci n'appartiendront pas *logiquement* au sous-système.

Il n'existait pas en Ada 83 de moyen pour contrôler au niveau du langage que les composants d'un sous-système n'étaient pas utilisés depuis l'extérieur du sous-système. C'est pourquoi Ada 95 a introduit la notion d'unités *enfants privées* : de telles unités ne sont utilisables que par leur père et leurs frères. Par exemple :

```
package Parent is
   ...
end Parent;

private package Parent.Enfant_1 is
   ...
end Parent.Enfant_1;

private package Parent.Enfant_2 is
   ...
end Parent.Enfant_2;

private package Parent.Enfant_2.Petit_fils is
   ...
end Parent.Enfant_2.Petit_fils;
```

Le corps du paquetage `Parent` peut référencer les paquetages `Parent.Enfant_1` et `Parent.Enfant_2`. Chacun des deux enfants peut référencer l'autre, mais seul le corps de `Parent.Enfant_2` peut référencer `Parent.Enfant_2.Petit_fils`. En revanche, le `Petit_Fils` peut référencer son «oncle» `Parent.Enfant_1`. Noter que seul le *corps* d'un parent peut référencer son enfant, car la spécification du parent doit être compilée avant celle de l'enfant. Aucune unité en dehors de celles qui descendent de `Parent` ne peut référencer les enfants privés.

Avec quelques différences dans la façon de les formaliser, ces sous-systèmes correspondent à la notion d'objet non terminal de la méthode HOOD [Hoo93], qui sert également à structurer hiérarchiquement les niveaux d'abstraction des objets. L'objet parent joue alors le rôle de point d'entrée, les autres modules constituant les objets enfants. Une hiérarchie HOOD se représente très bien au moyen d'unités hiérarchiques, bien que ceci ne fasse pas partie de la norme HOOD actuelle qui se limite aux possibilités d'Ada 83.

## *14.2 .3 Familles de composants*

Nous ne le répéterons jamais assez : il n'existe jamais de solution unique à un problème d'informatique. Même une simple gestion de chaînes de caractères peut être réalisée de plusieurs façons[1]. Lorsque des composants quelque peu évolués doivent être développés, ils utilisent souvent les services d'autres composants. Pour pouvoir coopérer et échanger des données entre eux, il est nécessaire qu'ils utilisent les mêmes abstractions de base. Nous dirons que des composants appartiennent à une même famille s'ils utilisent les mêmes types de base et qu'ils sont donc combinables entre eux sans conversion de données. Ces composants sont l'analogie logicielle des familles de circuits intégrés, TTL, ECL, etc. où, de façon similaire, le branchement de deux composants de la même famille ne pose aucun problème, alors que la connexion de deux circuits de familles différentes est possible, mais nécessite une adaptation.

On trouvera de telles familles lorsque des composants dépendent d'un choix de représentation des abstractions de base : chaînes de caractères, vecteurs, accès aux bases de données, systèmes de communication sur réseaux. On en trouvera également pour les interfaces utilisateurs (terminaux virtuels, systèmes de fenêtrage), ainsi que pour des conventions générales (comme le mécanisme de traitement et de gestion des erreurs). Lors du développement d'un composant logiciel, s'il n'est pas possible de le rendre indépendant, il faudra identifier clairement sa famille, c'est-à-dire les autres composants *généraux* qui sont «entraînés» par ce composant. On essayera bien entendu d'en limiter le nombre.

> Les nouveaux paquetages prédéfinis prédéfinis d'Ada 95 (chaînes, nombres complexes, bibliothèques mathématiques...) permettent de rendre indépendants des composants qui ne l'étaient pas auparavant.

Lors des phases préliminaires de la conception, le chef de projet doit faire le choix des familles de composants à utiliser, comme l'électronicien choisit une famille technologique. Il doit tenir compte des contraintes de son projet, mais aussi des familles auxquelles appartiennent les composants qu'il souhaite utiliser, pour minimiser les conversions de données. Dans une entreprise, on peut imposer l'utilisation d'un petit nombre de familles de composants afin de maintenir une interopérabilité maximale des modules. Un tel ensemble de familles «homologuées» constitue une véritable culture d'entreprise.

## *14.2 .4 Composants liés*

Il est courant que plusieurs niveaux de services gravitent autour d'une même abstraction. En particulier, lorsque l'on définit un type de donnée abstrait, on trouve des fonctionnalités fondamentales et d'autres qui, quoique très utiles, peuvent être construites par combinaison des fonctionnalités fondamentales. Plus ces fonctionnalités annexes sont évoluées, plus elles tendent à être spécifiques d'un besoin particulier, et leur nombre risque d'exploser rapidement. Aussi est-il préférable de *limiter* les fonctionnalités fournies dans la définition d'un type de donnée abstrait aux seules opérations fondamentales. Il se pose donc la question de la façon de définir les opérations complémentaires lorsque l'abstraction fondamentale ne fournit pas directement les services nécessaires. Trois solutions sont alors possibles[2] : définir une nouvelle entité indépendante, définir un type englobant ou enrichir les fonctionnalités.

### a) Définir une nouvelle entité indépendante

On définit un nouveau type de donnée abstrait, indépendant du type original, mais muni des fonctionnalités supplémentaires. Il est possible d'utiliser simultanément les deux types de données abstraits, qui n'ont aucun lien conceptuel. Il y a duplication d'un certain nombre de fonctionnalités. Cette situation est illustrée par la figure 27. Ce cas se produit lorsqu'il existe plusieurs notions

[1] C'est si vrai qu'Ada 95 fournit trois paquetages de gestion de chaînes, sensiblement différents.

[2] Cette classification a été établie dans cadre des activités d'Ada-France.

différentes auxquelles on se réfère sous un nom commun. Le besoin de nouvelles fonctionnalités provient de ce que les propriétés naturelles du nouveau type ne correspondent pas à celles de l'ancien.

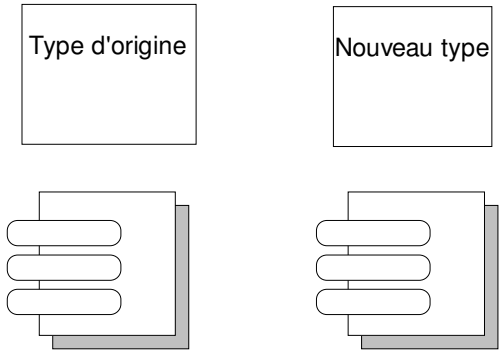


**Figure 27 :** Composants indépendants

Par exemple, nous avons un paquetage gérant les aspects liés au temps (`Calendar`). Si l'on a besoin de la notion de «temps virtuel» (pour faire un programme de simulation à événements discrets par exemple), il ne faut surtout pas tenter d'utiliser le type `Calendar.Time` ; la notion de temps simulé est conceptuellement différente de celle de temps «réel». Pourtant, les deux notions ont des choses en commun : la notion d'heure courante, les opérations arithmétiques entre temps et durée... On définira donc un nouveau type, indépendant de `Calendar.Time`, mais on fournira des opérations identiques dans leurs spécifications à celles de `Calendar`, de façon à éviter à l'utilisateur d'avoir à apprendre deux formes d'interfaces différentes.

### b) Définir un type englobant

Le nouveau type de donnée abstrait est différent du type d'origine, doté de nouvelles fonctionnalités et situé à un plus haut niveau d'abstraction ; son implémentation utilise le type de plus bas niveau. On interdit l'utilisation simultanée des deux niveaux. Cette situation est illustrée par la figure 28.

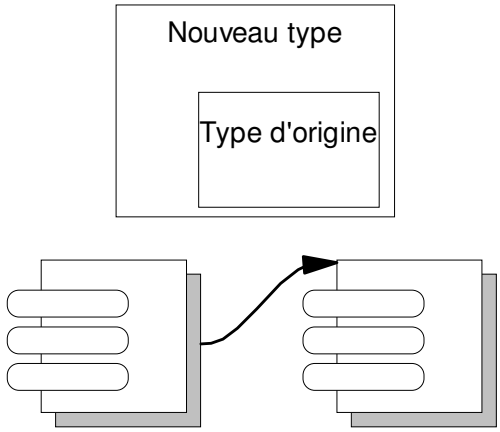


**Figure 28 :** Composant englobant

Par exemple, lorsque l'on conçoit des interfaces utilisateur, il est nécessaire d'afficher des messages dans différentes couleurs. Mais il existe deux niveaux de couleur : les couleurs physiques (rouge, bleu, jaune, ...) et les couleurs «logiques» (couleur d'un message d'erreur, couleur de fond d'une boîte de dialogue, couleur de bordure...). Pour permettre à l'utilisateur de changer ses couleurs à volonté, le programme ne devra utiliser que des couleurs logiques, que l'implémentation se contentera de traduire en couleurs physiques.

### c) Enrichir les fonctionnalités

On fournit un paquetage de type «collection de sous-programmes» qui procure des fonctionnalités complémentaires, comme illustré par la figure 29.

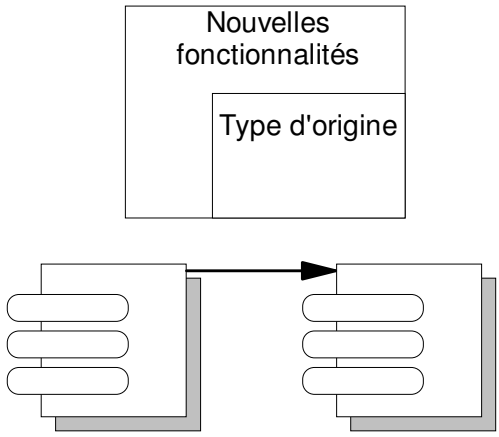


**Figure 29 :** Composant complémentaire

En Ada 83, il y aurait une dépendance de la spécification du nouveau paquetage vers celle du paquetage contenant le type d'origine et le paquetage complémentaire ne pourrait utiliser les informations privées du paquetage d'origine ; les fonctionnalités fournies devraient donc être construites par combinaison des fonctionnalités de base. En Ada 95, on utilise des unités *enfants publics*. Celles-ci s'écrivent comme les enfants privés, mais sans mettre le mot **`private`** en tête. Ces unités sont alors accessibles depuis n'importe quelle autre unité, comme si ce n'étaient pas des enfants ; l'avantage pour l'implémentation est qu'elles ont accès depuis leur propre partie privée et leur corps à la partie privée de leur parent : les fonctionnalités complémentaires peuvent donc être écrites de façon beaucoup plus efficace.

L'utilisateur doit employer le type d'origine en même temps que le paquetage de fonctionnalités complémentaires ; les deux niveaux cohabitent et doivent être utilisés ensemble. Ceci se fait automatiquement en Ada 95, car une clause de la forme **`with`** `Parent.Enfant` implique l'utilisation du parent aussi bien que de l'enfant.

Cette structure est fréquente avec les types de données abstraits. Ceux-ci possèdent généralement quelques fonctionnalités réellement fondamentales et indépendantes de l'application, et d'autres plus accessoires. Il est ainsi possible de séparer clairement ces deux aspects. Par exemple, un paquetage de nombres complexes fournira directement les opérations habituelles ; en revanche, les entrées-sorties ne font pas *fondamentalement* partie de l'abstraction : il est préférable de mettre celles-ci dans un paquetage séparé. Noter qu'il est également possible de fournir plusieurs variantes du même service : il suffit d'avoir plusieurs enfants avec exactement les mêmes spécifications (mais bien sûr des corps différents).

## 14.3 Classification des spécifications et des implémentations

La spécification et le corps d'un même composant n'appartiennent pas nécessairement à la même classe. Par exemple, une spécification peut ne faire référence à aucun autre paquetage, alors que son implémentation utilise une «technologie» qui la fait appartenir à une famille. Dans ce cas, le composant sera «indépendant» du point de vue de l'utilisateur et appartiendra à la famille pour la maintenance. L'appartenance de l'implémentation à une famille doit cependant être documentée dans les caractéristiques annexes d'implémentation, car il y a un risque d'incompatibilité entre la famille utilisée pour l'implémentation et les choix du projet. Par exemple, si le projet a choisi une certaine famille de terminal virtuel, il ne pourra utiliser un composant dont l'implémentation exigerait une famille de terminal virtuel différent.

Certains composants dont la nature exige une implémentation dépendant d'une famille peuvent fournir plusieurs variantes, correspondant aux implémentations selon différentes familles. Un système de fenêtrage peut ainsi avoir une variante correspondant à un terminal virtuel ANSI, une autre à une gestion d'écran IBM-PC et une troisième à une implémentation X Window.

## 14.4 Variantes, versions et systèmes de compilation

En général, il n'existera pas un seul exemplaire d'un composant logiciel. Tout d'abord, pour un même service (abstrait) rendu, on peut disposer de plusieurs implémentations, par exemple en fonction des différentes espèces définies dans la taxonomie de Booch (protection ou non contre les accès concurrents, types de gestion de la mémoire...), de différentes conventions de présentation des données à l'écran... Nous proposons d'appeler *variantes* un ensemble d'unités partageant la même spécification, mais différant dans leurs contraintes de réalisation. Idéalement, toutes les variantes devraient avoir des spécifications Ada identiques, sauf pour le nom des unités. En pratique, elles différeront au moins par leur partie privée. De plus, des sous-programmes supplémentaires peuvent être introduits, par exemple pour la gestion des variantes «contrôlées». Il faut donc considérer que la spécification d'un paquetage comporte logiquement deux parties, que nous appellerons spécification *principale* et spécification *annexe*. La spécification principale n'est dictée que par les contraintes de l'abstraction considérée, sa documentation peut (et doit) être commune à toutes les variantes. La spécification annexe regroupe des fonctionnalités liées à des contraintes d'implémentation particulières ; une documentation spécifique par variante doit être fournie. La lecture de la documentation de la spécification annexe ne doit en aucun cas être nécessaire pour comprendre le fonctionnement abstrait du composant.

Une entreprise développera en général ses composants sur une grande variété de triplets hôte-cible-compilateur. Nous appellerons un tel triplet un *système de compilation*. On peut le définir en disant que deux programmes d'un même système de compilation peuvent toujours être compilés dans une même bibliothèque de programme, et donc utiliser les mêmes composants sans recompilation. L'idéal est bien entendu de disposer de toutes les variantes de tous les composants logiciels sur tous les systèmes de compilation utilisés dans l'entreprise. En pratique, ce sera rarement le cas, car certains composants peuvent être spécifiques de certaines configurations (l'accès à un service de courrier électronique a peu de chances d'être utile avec un système pour carte autonome embarquée...). De plus, l'écriture de certains corps[1] peut faire appel à des éléments dépendant de l'implémentation, et donc nécessiter que l'on conserve des sources différents selon le système de compilation.

Enfin, comme tout logiciel, un composant peut être amené à évoluer dans le temps : il peut donc y avoir différentes *versions* successives d'un même composant. Ces trois degrés de liberté d'un composant logiciel sont orthogonaux : chaque composant devra être repéré par ses caractéristiques selon les composantes «variantes», «système de compilation», «version».

## 14.5 Exercices

1. Prendre un livre décrivant des composants matériels (comme un *TTL Data Book*) et chercher les analogies avec les classifications établies dans ce chapitre.
2. Quels sont les points importants de l'*implémentation* d'un composant logiciel qui doivent être portés à la connaissance de l'utilisateur ? Penser aux problèmes liés au parallélisme et à la gestion mémoire.
3. Etudiez le système de gestion de bibliothèques multiples de votre compilateur Ada favori et proposez une façon de l'utiliser pour faciliter l'organisation des différentes familles de composants logiciels.

[1] Jamais des spécifications!

# 15

# Règles pour l'écriture des composants logiciels

Nous allons présenter ici quelques règles particulièrement importantes pour l'écriture des composants logiciels. Bien sûr, elles sont également applicables aux modules non spécifiquement réutilisables, mais c'est avec les composants réutilisables qu'elles doivent être appliquées le plus rigoureusement.

## 15.1 Définition du comportement

Comme nous l'avons vu, une caractéristique du composant logiciel est que l'auteur ne dispose d'aucun moyen de pression sur l'utilisateur. Le comportement doit donc être totalement défini pour toute combinaison possible des valeurs d'entrée. Imaginons par exemple une procédure qui utilise un de ses paramètres comme quotient dans son algorithme :

```
procedure P (X : Integer) is
begin
   ...
   Y := 1/X;
   ...
end P;
```

Généralement, on documente que «la procédure ne fonctionne pas si elle est appelée avec la valeur 0». Cependant, une partie du calcul a pu être effectuée, éventuellement avec modification partielle de l'état global, avant le point où l'exception `Constraint_Error` est levée à cause de la division par 0. Le risque d'obtenir un état incorrect est donc sérieux. Par conséquent, si une valeur n'est pas autorisée, le sous-programme doit effectuer un test de validité avant toute autre opération. Dans l'exemple, on débuterait la procédure par :

```
begin
   if X = 0 then
       raise Constraint_Error;
   end if;
   ...
```

L'effet extérieur est apparemment le même, sauf qu'il est maintenant possible de documenter précisément que «l'appel avec la valeur 0 lève l'exception `Constraint_Error` et n'a aucun autre effet». D'une mention d'un non-fonctionnement de la procédure nous sommes passés à une spécification de comportement pour une valeur particulière. Si 0 est la seule valeur interdite, il n'est pas possible de faire mieux. Mais supposons maintenant que les valeurs négatives soient également interdites. La spécification peut alors devenir :

```
procedure P (X : Positive);
```

Il n'est désormais plus nécessaire de faire de test explicite, puisque celui-ci est assuré par les règles du langage. Mieux : il n'est plus nécessaire de *documenter* la levée de l'exception, puisque ce

comportement résulte de la seule spécification syntaxique du sous-programme. Notons au passage que le test engendré automatiquement par le compilateur a de bonnes chances d'être plus efficace qu'un test explicite. Tirons quelques règles de cet exemple :

Le comportement d'un sous-programme doit être *défini* pour toute combinaison possible des valeurs des paramètres. Nous appelons ici combinaison *possible* les valeurs de paramètres résultant de la seule application des règles du langage.
Une spécification de comportement peut être la levée d'une exception. Le diagnostic des valeurs incorrectes doit être effectué le plus tôt possible, et en tout état de cause la levée d'une exception doit être le seul effet de l'appel du sous-programme sur l'environnement.

- Il est préférable d'exprimer toute contrainte sur les paramètres au moyen des règles du langage, notamment par une utilisation judicieuse des sous-types. Le test explicite ne doit être utilisé que si le langage ne permet pas d'exprimer directement la contrainte.

Notons que pour l'application de la première règle, deux solutions sont possibles dans le cas de valeurs pour lesquelles le comportement du sous-programme est non spécifié : il faut soit *restreindre plus*, c'est-à-dire empêcher l'appel avec la valeur incorrecte, soit *spécifier plus*, c'est-à-dire lui donner une sémantique. Cette dernière solution est par exemple celle que nous avons choisie dans notre exemple d'introduction (la procédure `Permute`), lorsque la valeur de `Coupure` n'appartient pas à l'intervalle de définition de la chaîne : nous avons alors décidé de renvoyer la chaîne inchangée. Nous avons donc complété la sémantique du sous-programme. Nous aurions aussi bien pu décider de lever une exception dans ce cas.

Un possibilité intéressante d'Ada concerne les unités génériques. Ce que nous avons dit de la vérification des paramètres formels de sous-programmes s'applique également à la vérification des paramètres formels génériques. Le langage nous permet bien sûr de donner des sous-types appropriés aux paramètres génériques. Mais que faire si les contraintes ne peuvent s'exprimer au moyen des règles du langage ? Supposons par exemple un paquetage générique de génération de nombres aléatoires :

```
generic
   type Flottant is digits <>;
   Germe : in Positive;
package Aleat is
   ...
```

Certains algorithmes de génération exigent la présence d'un germe strictement positif (ce que nous avons exprimé par l'utilisation du sous-type `Positive`) impair. Cette dernière condition n'est pas exprimable dans le langage, *il est donc nécessaire de la vérifier par programme*. Il suffit de se rappeler que les instructions du corps d'un paquetage générique sont exécutées au moment de l'instanciation. Nous pouvons donc mettre tous les tests nécessaires dans la partie instruction du corps de paquetage :

```
package body Aleat is
   ...
begin
   if Germe mod 2 = 0 then
       raise Constraint_Error;
   end if;
end Aleat;
```

Notons que pour l'utilisateur qui tenterait d'instancier le générique avec une valeur négative, `Constraint_Error` serait levée par le compilateur, alors qu'avec une valeur paire elle serait levée par le test du corps de paquetage ; cependant, ces deux comportements sont indiscernables extérieurement, et il suffit de documenter que toute instanciation avec des valeurs incorrectes lève l'exception. Les règles du langage garantissent alors qu'aucun paquetage instancié avec des paramètres incorrects ne peut exister. Retenons que :

*En aucun cas il ne faut imposer un comportement à l'utilisateur sans le vérifier dans le composant.*

## 15.2 Gestion des situations exceptionnelles

Comme nous venons de le voir, un composant logiciel doit diagnostiquer un certain nombre de situations exceptionnelles. Bien que plusieurs techniques soient disponibles (nous les passerons en revue dans la quatrième partie), le mécanisme d'exceptions doit être utilisé dans ce cas ; mieux, l'on peut dire que les exceptions ont été spécialement conçues pour permettre l'écriture de composants logiciels.

Pour bien comprendre ce point, rappelons ce qui se passe par exemple en FORTRAN lorsqu'une erreur d'exécution survient. Nous avons connu une bibliothèque mathématique FORTRAN qui, lors d'une erreur de paramètre (ARCSIN(2.0) par exemple), envoyait un message d'erreur sur le terminal. Or rien ne prouve qu'il y ait un terminal ! Ou plus vraisemblablement, le terminal peut être dans un état (mode graphique) qui ne lui permette pas d'imprimer un message. Le résultat peut être catastrophique, car le message met le terminal dans un état «bizarre»... et le programme appelant n'est pas prévenu que la fonction n'a pu réaliser le travail demandé. En fait, le problème vient de ce que le composant *diagnostique* une faute de paramètres, mais que l'*origine du problème* se trouve dans l'appelant ; le composant n'est donc pas à même de prendre une *décision* quant au traitement de l'erreur : par définition, le composant logiciel ignore tout du contexte dans lequel il est appelé. Il est donc nécessaire de signaler à l'appelant de façon non équivoque que le traitement demandé n'a pu avoir lieu, et c'est bien la raison d'être des exceptions.

Si cet appelant est lui-même un autre composant logiciel, il ne sait ni pourquoi l'exception a été levée, ni quel traitement effectuer : il doit donc *laisser filer* l'exception ou, à la rigueur, la *transformer* en une exception à lui. Lorsque l'exception pénètre dans un module spécifique du projet, elle doit être rattrapée et traitée selon les conventions du projet. On peut donc résumer ces principes au moyen des règles suivantes :

*Un composant indépendant doit soit lever une exception, soit rendre exactement le service demandé.*
*Un composant indépendant ne doit pas faire disparaître une exception.*
*Le premier niveau non indépendant est chargé du traitement de l'exception.*

## 15.3 Initialisation

### *15.3 .1 Initialisation du composant*

Les bibliothèques, les machines abstraites et beaucoup d'autres composants nécessitent une initialisation avant de pouvoir être utilisés de façon correcte. Avec la plupart des langages de programmation, il existe une procédure d'initialisation, et la documentation avertit généralement qu'«il faut l'appeler avant toute utilisation». Bien entendu, l'utilisateur ne sait rien de ce qui se passe si jamais il omet de l'appeler, mais il se doute que cela ne pourrait apporter qu'un mauvais fonctionnement. Et que se passe-t-il si la procédure d'initialisation est appelée deux fois ? Mystère, bien qu'en général ce soit aussi nocif que de ne pas l'appeler du tout.

Tout le problème est que dès que l'on utilise des composants un peu évolués, ils font eux-mêmes appel à d'autres composants, qui doivent également être initialisés. Si un composant A utilise un composant B, il paraîtra logique d'inclure l'appel à l'initialisation de B dans la procédure d'initialisation de A.

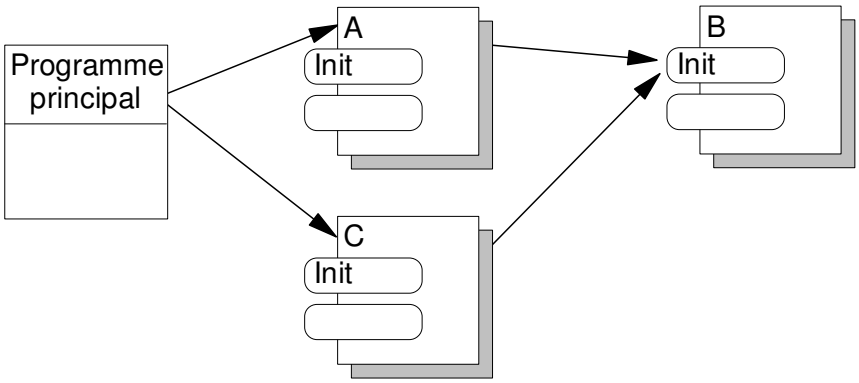


**Figure 30 :** Dépendances à l'initialisation

Hélas, un composant général est susceptible d'être utilisé par plusieurs autres composants. Supposons un schéma comme celui de la figure 30. Le composant B est utilisé par A et C, eux-mêmes utilisés par le programme principal. Logiquement, les procédures d'initialisation de A et de C doivent faire appel à l'initialisation de B, et la procédure d'initialisation de C sera appelée deux fois. Bien sûr, A ne connaît pas C, et réciproquement, donc il ne peut supposer que l'initialisation de B est effectuée par le collègue. La seule solution est de *ne pas effectuer l'initialisation des composants utilisés*, et de se reposer sur l'utilisateur en documentant qu'avant d'appeler l'initialisation de A et de C, il faut appeler celle de B... Outre que cette solution fragilise le système, elle rend une partie de l'implémentation visible : l'utilisateur de A et de C n'était pas nécessairement au courant de l'existence de B, et la nécessité d'appeler cette initialisation va créer des liens de dépendance pour cette seule fonction, qui dénaturent l'architecture du projet.

La meilleure solution consiste donc en l'auto-initialisation des composants. Pour les bibliothèques et les machines abstraites, il est facile de faire effectuer l'initialisation par les instructions du corps de paquetage. Si l'on souhaite que l'initialisation ait lieu plus tardivement que l'élaboration du corps, il faut garder simplement une variable globale spécifiant que la bibliothèque n'est pas initialisée. Cette variable sera testée par tous les points d'entrée publics, et l'on déclenchera l'initialisation lors du premier appel d'un module.

Quant aux types de données abstraits, ils sont généralement représentés par des types articles, dont les composants cruciaux peuvent contenir des valeurs d'initialisation. N'oublions pas que l'initialisation d'un composant de type article peut parfaitement être dynamique et contenir n'importe quelle expression ! Supposons par exemple un type de donnée dont chaque instance doit se faire attribuer un numéro d'identification unique. Habituellement, la documentation dirait qu'il faut appeler une procédure d'initialisation après avoir déclaré chaque variable. Il existe un risque sérieux de mauvais fonctionnement si une variable n'est pas initialisée ou initialisée deux fois. En Ada, nous avons la possibilité de déclarer l'objet comme ceci :

```
package Objet_Identifié is
   type Objet is private;
   -- Opérations sur l'objet...

private
   type ID_Type is new Positive;
   function Identification return ID_Type;
   type Objet is
      record
         ID : ID_Type := Identification;
         -- autres composants ...
      end record;
end Objet_Identifié;
```

Les mécanismes du langage vont *garantir* que la fonction d'initialisation sera appelée une fois pour chaque objet, et qu'il n'est pas possible de l'appeler autrement que pour l'initialisation d'un objet (puisqu'elle est déclarée dans la partie privée).

> Pour des besoins plus complexes, Ada 95 permet de définir des procédures d'initialisation, de finalisation et de contrôle de l'affectation au moyen du paquetage `Ada.Finalization`.

Dans certains cas, il n'est pas possible d'effectuer une initialisation automatique, et l'on doit recourir à l'utilisation de sous-programmes explicites. Certaines méthodes de conception (HOOD) imposent également des initialisations explicites. Dans ce cas, il faut prendre des précautions supplémentaires pour garantir le bon fonctionnement. Une technique courante est l'utilisation de *jetons* : lors de l'appel à une procédure d'une bibliothèque, on doit présenter une valeur (généralement d'un type limité pour éviter les contrefaçons), valeur fournie par la procédure d'initialisation. La technique utilisée pour la gestion des fichiers («ouverture» du fichier, suivie de la présentation à chaque procédure d'entrée-sortie du fichier «ouvert») n'est qu'une variante de la technique du jeton. Noter que pour être parfaitement sûre, cette technique nécessite une auto-initialisation du jeton, qui ne peut être assurée que par les techniques précédentes. L'initialisation automatique reste donc nécessaire pour amorcer le mécanisme.

Si aucune des techniques précédentes n'est applicable, on est obligé de s'en remettre à la méthode classique de la procédure d'initialisation. Le composant doit alors mettre en place un mécanisme pour *vérifier* qu'il est initialisé correctement, grâce à une variable globale interne. La procédure d'initialisation lèvera une exception si elle est appelée sur un module déjà initialisé, de même que l'appel de toute autre procédure sur un module non initialisé. Ceci impose de séparer la fonction d'initialisation de la fonction de réinitialisation, séparation qui est d'ailleurs de toute façon souhaitable.

Les mêmes principes et les mêmes techniques sont applicables, en dehors des cas d'initialisation, chaque fois qu'il existe une dépendance temporelle dans l'ordre d'appel des sous-programmes. Les solutions seront choisies de préférence dans l'ordre suivant :

Ne pas introduire de dépendance temporelle, ou faire en sorte qu'elle soit cachée (ajustement automatique du comportement grâce à une utilisation judicieuse du langage).
Mettre des sécurités au niveau de l'objet : utilisation de jetons, «ouverture» d'objets...

- Mettre des sécurités globales et rejeter les utilisations incorrectes.

Encore une fois, notons qu'en aucun cas la sécurité ne doit être assurée en supposant simplement que l'utilisateur a suivi les règles du mode d'emploi.

## *15.3 .2 Elaboration du composant*

Si l'on utilise des initialisations explicites, l'ordre dans lequel elles s'effectuent est déterminé par l'utilisateur. Dans le cas d'initialisations automatiques, les règles du langage ne déterminent qu'un ordre *partiel* ; l'ordre total dépend en partie de l'implémentation. Il est possible de contrôler plus finement cet ordre, appelé ordre *d'élaboration*, au moyen du pragma `Elaborate_All` qui demande que certains modules (ainsi que ceux dont ils dépendent de façon transitive) soient élaborés avant le module où le pragma apparaît.

> En Ada 83 n'existait que le pragma `Elaborate`, qui n'était pas transitif : si le paquetage utilisé dépendait lui-même d'autres paquetages dont l'élaboration était nécessaire, il fallait les nommer explicitement. Le pragma `Elaborate_All` a remédié à ce problème et doit normalement être utilisé à la place de l'ancien pragma `Elaborate`, sauf dans quelques cas d'ordre d'élaboration très spéciaux.

Bien que ce soit rare, il existe des cas où un ordre d'élaboration incorrect peut conduire à la levée de l'exception `Program_Error`[1]. Il faut alors faire preuve d'une vigilance particulière, car il est parfaitement possible d'obtenir un comportement correct sur les programmes de tests, et qu'une fois en service le composant refuse de fonctionner. Le risque est cependant limité, car le programme ne démarre même pas.

Un autre cas de dépendance à l'élaboration est fourni par l'exemple de `l'Objet_Identifié` du paragraphe précédent : toute déclaration d'un objet du type `Objet` *par l'utilisateur* lèvera l'exception

[1] Ne jetons pas la pierre à Ada : ceci est dû à la présence d'initialisations dynamiques qui n'existent pas dans les autres langages de programmation.

`Program_Error` si le corps de `Objet_Identifié` n'a pas été élaboré. Or cette dépendance n'est pas visible par l'utilisateur si on ne lui communique pas le contenu de la partie privée du paquetage. Il convient donc de *documenter* la nécessité de mettre un pragma `Elaborate_All` dans tout module déclarant des objets de ce type.

## 15.4 Définition de génériques

La notion de générique est intimement liée à celle de composant logiciel, car elle permet de *généraliser* un algorithme en le rendant applicable à tout un ensemble de types de données.

### *15.4 .1 Développement de générique*

Supposons que nous écrivions une fonction pour trouver le plus grand de deux nombres entiers :

```
function Max (Left, Right : Integer) return Integer is
begin
   if Left < Right then
      return Right;
   else
      return Left;
   end if;
end Max;
```

Ne pouvons-nous écrire cette fonction que pour le type `Integer` ? Bien entendu non. La seule propriété nécessaire est l'existence d'une relation d'ordre (opérateur `"<"`). Nous pouvons donc généraliser cette fonction à tout type de donnée muni d'un opérateur `"<"` :

```
generic
   type Donnée(<>) is limited private;
   with function "<"(Left, Right : Donnée) return Boolean
      is <>;
function Max (Left, Right : Donnée) return Donnée;
```

> La boîte utilisée dans `Donnée(<>)` est une nouveauté Ada 95 qui exprime que le composant peut être instancié même avec des types non contraints.

Nous avons ici généralisé l'algorithme en analysant quelles étaient les propriétés *minimales* nécessaires pour l'écrire. Ces propriétés sont exprimées par les paramètres génériques de l'unité. Les règles d'Ada vérifieront que le type effectif utilisé pour l'instanciation possède *au moins* ces propriétés minimales, garantissant la validité de l'instanciation.

On commencera donc rarement par écrire un composant générique. Un projet éprouvera le besoin de développer un certain algorithme, généralement pour un type de donnée particulier. Par la suite, on accroîtra la réutilisabilité du composant en analysant les propriétés du type de donnée réellement utilisées, et en passant le type et ses propriétés nécessaires en générique.

Bien sûr, tous les composants ne sont pas candidats à la transformation en générique. En premier lieu, on cherchera à rendre génériques les types de données abstraits de «deuxième niveau», c'est-à-dire ceux qui utilisent un autre type pour leur implémentation (type numérique en particulier).

### *15.4 .2 Autres types de génériques*

Deux sortes d'unités génériques échappent à ce schéma général d'évolution, et sont conçues en génériques dès le début : les unités paramétrables et les structures de données.

Les unités paramétrables ne sont génériques que pour permettre de passer des facteurs de dimensionnement, comme des tailles maximales de tables. Ce sont en quelque sortes de «faux» génériques, en ce sens qu'elles ne correspondent pas à la notion de type de donnée abstrait construit

«par-dessus» un autre type. Ces unités ne comportent en principe que des paramètres **`in`**, et ne sont en général instanciées qu'une seule fois dans un projet.

Inversement, les gestionnaires de données sont fondamentalement prévus pour manipuler d'autres types de données dont on ignore tout *a priori*. Ils seront donc conçus comme génériques dès le début.

### *15.4 .3 Génériques et exceptions*

Il existe plusieurs points spécifiques à l'utilisation des exceptions dans/par les composants génériques.

Tout d'abord, lorsqu'une exception est déclarée dans la partie visible d'un paquetage générique, chaque instanciation provoque la création d'une exception *différente* ; ceci n'est en général pas souhaitable. On regroupera donc souvent les exceptions dans un paquetage annexe *non générique*, suivant l'exemple du paquetage `Ada.IO_Exceptions` pour les entrées-sorties.

Ensuite, lorsqu'un sous-programme est passé en paramètre générique, l'unité générique doit considérer la possibilité de levée d'une exception par le sous-programme effectif, sous-programme sur lequel elle n'a aucun pouvoir, car il est fourni par l'utilisateur. En général, ces sous-programmes participent à la réalisation d'une fonctionnalité de l'unité générique ; un échec de ce sous-programme ne peut signifier qu'un échec de la fonctionnalité utilisatrice. Il faudra en général laisser «filer» l'exception, et donc exclure tout traite-exception «**`when others`**» dans l'appelant, sauf si celui-ci relève soit la même exception, soit une exception spécifique du paquetage.

Enfin, on ressent parfois le besoin de faire lever par le générique une exception appartenant au domaine de l'utilisateur. Pourquoi ? Parce que l'on se trouve dans le cas où un générique diagnostique une erreur, mais où la décision de l'action à prendre appartient à l'utilisateur. En fait, lever une exception de l'utilisateur est bien trop restrictif : rien ne dit que la réaction souhaitée soit justement une simple levée d'exception ! Ceci nous conduit à une meilleure solution : importer une procédure de traitement d'erreur. Celle-ci pourra effectivement lever une exception[1], ou faire tout autre traitement approprié. L'utilisation des valeurs par défaut des procédures est tout à fait utile dans ce cas. Par exemple, une bibliothèque effectuant des traitements mathématiques pourrait se présenter ainsi :

```
procedure Action_Par_Défaut (Valeur_Fournie : out Float) is
begin
   raise Constraint_Error;
end Action_par_Défaut;

with Action_Par_Défaut;
generic
   with procedure Si_Débordement(Valeur_Fournie: out Float)
       is Action_Par_Défaut;
package Fonctions_Mathématiques is ...
```

En cas de débordement, on appelle `Si_débordement`. Par défaut, ceci provoque la levée de `Constraint_Error`, mais l'utilisateur qui ne souhaite pas interrompre les calculs peut fournir une procédure `Si_débordement` explicite qui fournira une valeur de substitution (`Float'LARGE` ou 0.0 par exemple).

## 15.5 Portabilités et dépendances à l'implémentation

La portabilité est un point fondamental pour un composant logiciel. Il existe pourtant des fonctionnalités qui dépendent nécessairement de l'implémentation. Mieux : de nombreux composants ont précisément pour but d'*encapsuler* ces dépendances, afin de rendre le reste de

[1] Que le composant logiciel ne devra bien entendu surtout pas traiter !

l'application totalement portable. Il importe donc d'identifier correctement ce qui est, ou n'est pas, portable, et de savoir de quelle portabilité il s'agit.

### *15.5 .1 Portabilités a priori et a posteriori*

Nous parlerons de portabilité *a priori* lorsque le résultat de l'exécution du programme peut être prédit d'après la seule lecture du code, de façon indépendante de l'implémentation. C'est en principe le cas le plus fréquent.

Nous parlerons de portabilité *a posteriori* lorsque le résultat ne peut être déterminé indépendamment de l'exécution, mais que le logiciel a été prévu pour tenir compte des variations dues à l'implémentation. Il utilisera les attributs des types ou les constantes du paquetage `System` pour ajuster son fonctionnement aux particularités du système d'exécution.

> Le paquetage `System` contient (par définition) les déclarations de tous les éléments qui dépendent de l'implémentation.

Les deux formes de portabilité sont acceptables, à condition que l'utilisation de l'une ou de l'autre résulte d'un choix délibéré, et non d'une insuffisance de spécification... qui conduit généralement à n'avoir aucune des deux ! Nous allons illustrer ce point par un exemple. Supposons qu'un programme ait besoin de stocker des données sous forme de chaînes de caractères. Une taille limite doit être choisie pour ces chaînes. Une erreur fréquente serait de déclarer le type comme suit :

```
Max : constant := 40_000;  -- par exemple
type Donnée is new String(1..Max);
```

Une telle déclaration n'assure en effet aucune des deux portabilités ! Il n'y a pas portabilité *a priori*, car le type `String` dépend de l'implémentation, et rien ne dit qu'il soit possible de déclarer une chaîne de 40 000 caractères[1]. Il n'y a pas non plus portabilité *a posteriori*, car il n'existe aucun paramétrage en fonction de l'implémentation. Il faut donc déclarer le type comme ceci (portabilité *a priori*) :

```
Max : constant := 40_000;
type Index_Donnée is range 1..Max;
type Donnée is array (Index_Donnée) of Character;
```

ou bien comme ceci (portabilité *a posteriori*) :

```
type Donnée is new String;
Max : constant := Donnée'Last;
```

Noter que dans ce dernier cas, la constante `Max` est déterminée à partir du type `Donnée`, et non le contraire. Le programme s'ajuste aux possibilités de l'implémentation. On trouvera fréquemment cette distinction entre les deux types de portabilité dans les modules à nature numérique : on a le choix entre imposer une précision *a priori* en définissant les types de façon absolue :

```
type Donnée is digits 7;
```

auquel cas on peut déterminer la précision des calculs indépendamment de l'implémentation, ou bien utiliser les nombres «normaux» du matériel :

```
type Donnée is new Float;
```

La précision des calculs dépend alors de la machine, mais peut être déterminée après coup de façon portable à partir de l'attribut `Donnée'Digits`.

[1] Si le type `Integer` est sur 16 bits, les chaînes sont limitées à 32 767 caractères.

## *15.5 .2 Dépendances à l'implémentation*

Nous dirons d'une unité de compilation qu'elle *dépend de l'implémentation* si l'on a prévu, dès la conception, qu'il puisse être nécessaire de la récrire en cas de changement de l'un des éléments du système de compilation. Nous avons bien dit «récrire» et non «modifier» ou «adapter» : tout changement, même minime, dû à une différence d'implémentation doit être considéré comme une nouvelle variante du composant, indépendante de celle qui lui a donné naissance. Faute de quoi, on risque d'assister à des «effets de yo-yo[1]» menant tout droit à des modules qui ne fonctionnent plus sur aucune implémentation... Noter que des génériques du genre «paramétrable» peuvent permettre de ne maintenir qu'une seule variante pour des systèmes suffisamment voisins : on garde les parties communes dans le générique, et on fournit les parties spécifiques sous forme de sous-programmes importés.

On trouve des dépendances à l'implémentation pour réaliser des interfaces avec le matériel, avec le système d'exploitation hôte, ou avec des bibliothèques écrites dans d'autres langages. Dans tous les cas, il existe un objet réel sous-jacent : périphérique, service système, service de la bibliothèque. La meilleure solution consiste à définir trois couches d'interfaçage. On commence par définir un composant représentant aussi fidèlement que possible une abstraction de l'objet réel. Sa spécification, comme son corps, comportent des éléments dépendant de l'implémentation. Le but de cette couche est de libérer les couches suivantes de tous les éléments «pénibles» de l'interface. On trouve ensuite une couche représentant le *service* fourni par l'objet réel : sa spécification est indépendante de l'implémentation, mais pas son corps. Enfin, on trouve un composant représentant le *besoin abstrait* satisfait par le service : spécification et corps sont alors indépendants de l'implémentation.

Illustrons cette structure par un exemple : nous voulons accéder aux fonctions du haut-parleur d'un IBM-PC. Celui-ci est piloté par un port 8253. Pour y accéder, nous devons réaliser l'abstraction du plus bas niveau des instructions IN et OUT des processeurs INTEL :

```
package In_Out is
   type Périphérique  is range 16#00# .. 16#FF#;
   type Octet         is mod 256;
   for Octet'SIZE use 8;

   procedure In_8086  (Depuis : in  Périphérique;
                       Donnée : out Octet);
   procedure Out_8086 (Vers   : in  Périphérique;
                       Donnée : in  Octet);
end In_Out;
```

Le corps sera écrit en assembleur ou fera appel au paquetage `Ada.Machine_Code`. Ce paquetage permet d'accéder à la fonction «son» :

```
package Contrôle_Son is
   type Fréquence is range 20..20_000;
   procedure Emettre (Note :    in Fréquence);
   procedure Emettre (Note :    in Fréquence;
                      Pendant : in Duration);
   procedure Arrêter_Son;
end Contrôle_Son;
```

Le corps de ce paquetage dépendra de l'implémentation, car il faudra notamment connaître les adresses physiques du port d'entrée-sortie. Cette deuxième couche nous permet de réaliser un paquetage totalement abstrait :

```
package Musique is
   type Gamme is
       (Silence,  Ut,  Ut_Dièze,  Ré, Ré_Dièze, Mi, Fa,
        Fa_Dièze, Sol, Sol_Dièze, La, La_Dièze, Si);

   type Numéro_Gamme is range -4 .. + 4;
```

[1] Deux programmeurs effectuant alternativement des modifications sur un module pour l'adapter chacun à son besoin... et cassant ce qui vient d'être fait par l'autre.

```
      type Rythme is range 10..240;
      Rythme_Courant : Rythme := 60;

      type Durée_Note is delta 1.0/16.0 range 0.00 .. 4.00;
      Noire : constant Durée_Note := 1.0;

      type Note_A_Jouer is
          record
              Note    : Gamme;
              Hauteur : Numéro_Gamme;
              Durée   : Durée_Note;
          end record;

      type Partition is array (Positive range <>)
          of Note_A_Jouer;

     procedure Jouer (Note : in Note_A_Jouer);
     procedure Jouer (Air  : in Partition);
     procedure Arrêter_Musique;
     function  Musique_En_Cours return Boolean;
  end Musique;
```

Le corps de ce dernier paquetage ne fait appel qu'aux fonctionnalités du paquetage `Contrôle_Son`. Nous avons ainsi isolé et empaqueté les dépendances : en cas de portage sur une autre machine disposant d'instructions similaires, nous n'aurons à récrire que le paquetage `In_Out`. En cas de portage sur un système très différent, les deux couches inférieures devront être changées, mais le volume de récriture sera en pratique extrêmement réduit. Bien sûr, l'application ne devra utiliser que le paquetage `Musique`, de façon à ne pas introduire de modifications importantes.

On imagine bien comment appliquer ce processus à d'autres exemples : une interface avec un système graphique aura une première couche représentant l'interface avec la bibliothèque système (appel de sous-programmes C en général), une deuxième couche fournira les fonctionnalités abstraites de base (positionnement à l'écran, dessins élémentaires) et la troisième couche fournira les éléments réellement utiles (menus, boîtes de dialogue, etc.).

Insistons sur le fait que pour les utilisateurs, seule la troisième couche est effectivement utilisable, et représente en fait le point d'entrée d'un sous-système local. Si on réalise plusieurs implémentations différentes, elles ne doivent différer ni sur la spécification syntaxique (sauf modification de partie **`private`**, ou peut-être extensions ne fournissant qu'une compatibilité à sens unique), ni sur la spécification sémantique, sauf pour les performances, le comportement vis-à-vis du parallélisme et des éléments non visibles extérieurement (comme l'utilisation ou non de la souris).

## 15.6 Exercices

1. Définir une interface en trois couches comme au paragraphe précédent pour gérer l'accès à une imprimante.
2. Imaginer un cas de dépendances à l'initialisation qui requiérerait l'utilisation du pragma `Elaborate`, mais ne pourrait être résolu avec le pragma `Elaborate_All`.
3. Généraliser l'exemple du chapitre 12 en le passant en générique pour lui permettre de permuter les éléments de n'importe quel tableau monodimensionnel.

# 16

# Etablissement et gestion d'une bibliothèque de composants

Jusqu'à présent, nous avons vu comment développer des composants logiciels. Mais il ne s'agit là que d'une partie du problème : une fois que l'on possède des composants, que va-t-on en faire ? A la limite, il est possible d'acheter des composants tout faits : peu importe alors comment ils ont été développés. Mais il faudra tout de même mettre en place une structure permettant de les retrouver et de promouvoir leur utilisation : c'est ce que nous allons étudier maintenant.

## 16.1 Responsable composants logiciels

L'utilisation de composants logiciels n'est rentable que s'ils sont réutilisés par plusieurs projets, qui peuvent avoir des intérêts divergents. Pour conserver la cohérence, il est nécessaire de mettre en place une structure *transversale*, dirigée par un responsable composants logiciels.

Le rôle du responsable composants logiciels n'est *pas* d'écrire de nouveaux composants. Si la taille de l'entreprise le permet, il pourra diriger une équipe entièrement consacrée aux composants, mais en général les composants seront développés par les équipes de projet. Le responsable composants logiciels est chargé de *gérer* les composants ; il doit veiller au bon respect des règles d'uniformité, au suivi des différentes versions et de la documentation associée. Il lui faut maintenir une base de données des utilisations des composants, faire circuler les mises à jour des fiches descriptives, et aider les utilisateurs à rechercher le composant correspondant à leurs besoins.

Le responsable composants logiciels est également chargé de veiller à la promotion de l'utilisation de composants standard et d'empêcher la prolifération des adaptations «spécifiques» de ces composants. En particulier, il doit conserver les sources et *refuser* de fournir ceux d'un composant réutilisable à une équipe qui souhaiterait l'adapter à un cas particulier, si cette modification va dans le sens d'une perte de généralité.

Compte tenu de ses missions, la personne la plus apte à devenir responsable composants logiciels est un responsable qualité. Dans une période transitoire, ou si la quantité de composants ne justifie pas une personne à plein temps, la gestion des composants peut très bien s'effectuer sous le contrôle de la structure d'assurance qualité.

## 16.2 Documentation

La documentation est un point crucial pour le développement d'une approche «composant logiciel». Imagine-t-on les composants électroniques sans les nombreux guides, catalogues et autres *data book* permettant de connaître leurs caractéristiques et de retrouver la référence du ou des composants susceptibles de répondre à un besoin ?

Il convient de distinguer tout de suite deux sortes de documentations : d'une part une documentation «utilisateur», équivalent des catalogues de matériel, destinée exclusivement aux

utilisateurs des composants ; d'autre part, une documentation de conception, de suivi et de maintenance, destinée aux concepteurs, gérants et mainteneurs du composant, et qui ressemble beaucoup plus à la documentation habituelle de tout projet logiciel. Nous allons examiner successivement ces deux aspects.

## *16.2 .1 Documentation externe*

Cette documentation s'adresse à l'utilisateur de composants logiciels. Le système le plus commode, éprouvé de longue date avec les composants matériels, consiste à établir un modèle de fiche préimprimée standard, décrivant les différents aspects du composant. Nous allons présenter un tel modèle de fiche adapté au cas des composants logiciels (le modèle complet est donné en annexe). Ce modèle ne doit pas être pris comme absolu, chaque entreprise est invitée à développer le sien propre en fonction de ses habitudes.

Nous avons identifié trois niveaux de documentation nécessaires à l'utilisateur pour choisir et utiliser un composant logiciel. A chacun de ces niveaux correspond une rubrique particulière de la fiche : «Identification», «Spécification» et «Implémentation». Des études aux Etats-Unis ont abouti à une conclusion quasi similaire, connue sous le nom de modèle «3c» : Concept, Contenu, Contexte [Tra89].

### a) Identification

La fiche doit d'abord permettre à l'utilisateur de sélectionner rapidement un petit nombre de composants susceptibles de répondre à son besoin. Pour cela, il est nécessaire bien entendu que le composant réponde au besoin, mais également que le composant soit disponible pour le système de compilation, et même que les conditions de licence soient acceptables compte tenu des particularités du projet. La partie «identification» de la fiche regroupe donc une brève description du composant et des variantes disponibles, la classification du composant par rapport aux critères du chapitre 14, les modalités d'accès éventuelles (conditions de licence, droits d'accès) ainsi que les informations sur la disponibilité et l'historique du composant. Au vu de cette partie, l'utilisateur doit être à même de prendre l'une des décisions suivantes :

*Utiliser le composant en l'état*. C'est évidemment la décision préférentielle, sous réserve que le composant soit disponible pour le système de compilation désiré. La nécessité de modifier légèrement la structure du projet pour pouvoir utiliser le composant en l'état n'est *pas* une raison suffisante pour rejeter cette solution.

*Porter le composant*. Ceci correspond au cas où un composant paraît satisfaisant, mais n'est pas disponible sur le système de compilation désiré. Il convient de prendre immédiatement contact avec le responsable composants logiciels, 1) pour s'assurer que la non-disponibilité provient effectivement d'une absence de besoin antérieur et non d'une impossibilité technique cachée, 2) que le portage n'est pas déjà en cours par une autre équipe, et 3) pour demander éventuellement des crédits supplémentaires pour effectuer le portage. Si le portage est effectué par le demandeur, ses droits de modification sur le source seront extrêmement restreints, et toute modification allant au-delà de la simple adaptation au nouveau système de compilation doit être effectuée sous le contrôle du responsable composants logiciels.

*Développer une nouvelle variante*. Ceci correspond au cas où la spécification abstraite d'un composant paraît satisfaisante, mais où la variante correspondant aux besoins du projet n'existe pas encore. Les contraintes sont similaires au cas du portage du composant.

*Demander une modification du composant*. Ceci correspond au cas où un composant existant ne correspond qu'imparfaitement aux besoins, et où un perfectionnement semble possible, aboutissant à une nouvelle version du composant. Une demande en ce sens doit être déposée auprès du responsable composants logiciels, qui n'est recevable qu'à condition que la

modification soit compatible de façon ascendante avec la version précédente et qu'elle aille dans le sens d'une plus grande généralité du composant. En aucun cas la modification ne doit être effectuée par celui qui est à l'origine de la demande, mais par l'équipe support du composant logiciel. Celle-ci proposera en retour une modification effective qui ne correspondra pas nécessairement à la demande originale. En effet, la demande peut révéler une insuffisance de conception d'un composant, mais l'analyse peut découvrir une possibilité de modification plus générale que ce qui était proposé initialement.

- *Proposer la définition d'un nouveau composant*. Le responsable composants logiciels doit en être immédiatement informé, comme pour les propositions de modification. L'écriture d'un nouveau composant ne doit être acceptée que s'il est suffisamment différent des composants existant dans la base ; sinon les autres possibilités doivent être envisagées en priorité. Le développement initial du nouveau composant peut être effectué par le demandeur, mais la définition des spécifications et l'intégration du composant doivent être effectuées sous le contrôle du responsable composants logiciels.

### b) Spécification

Ayant identifié le bon composant, l'utilisateur doit connaître le mode d'emploi externe : c'est le but de la partie «Spécifications», qui reprend les éléments principaux de la spécification Ada de l'unité, en rajoutant les informations sémantiques qui ne peuvent être déduites de la seule spécification syntaxique. On séparera à ce niveau les éléments principaux, indépendants de la variante, des éléments annexes, particuliers à certaines variantes.

### c) Implémentation

Enfin, l'utilisation effective nécessitera des informations complémentaires. C'est le rôle de la partie «implémentation», qui précisera également des éléments d'information n'ayant rien à voir avec la spécification abstraite, mais qui peuvent être importants pour l'utilisateur : utilisation d'éléments particuliers du langage (points flottants, parallélisme, gestion dynamique de mémoire), performances, conditions d'élaboration et d'initialisation...

## *16.2 .2 Documentation des performances*

Bien que faisant logiquement partie de la documentation externe, la documentation des performances constitue un point suffisamment délicat pour que nous lui consacrions un paragraphe spécial. L'utilisateur d'un composant logiciel a besoin de connaître, au moins approximativement, les performances qu'il est en droit d'attendre d'un composant pour l'établissement d'un «budget de temps». Or les temps d'exécution peuvent être extrêmement variables pour un même module, parfois même en fonction de facteurs qui échappent totalement aussi bien au concepteur du composant qu'à l'utilisateur.

Il existe une autre raison, généralement méconnue, et pourtant fondamentale, d'essayer de donner une idée des performances d'un module. L'utilisateur est en général obsédé par la question des performances, et si plusieurs structures utilisant différents composants sont possibles, il choisira bien souvent en fonction des performances des différents composants. En l'absence de documentation, même approximative, l'utilisateur devrait faire des mesures de performances avant de décider de la meilleure solution. Hélas ! C'est rarement le cas. L'utilisateur choisit le composant qui lui semble le plus rapide, *généralement sans aucun support objectif* pour justifier son choix. Or s'il est un cas où l'on se trompe souvent, c'est sur l'évaluation des performances d'un module. Toute indication, même approximative et relative, des performances peut donc éviter de monumentales erreurs de conception.

Un premier point peut être relativement facilement documenté : la complexité de l'algorithme, c'est-à-dire la variation des temps d'exécution en fonction de la taille des données manipulées. Ceci est indispensable pour les structures de données, mais d'autres composants peuvent être susceptibles de bénéficier de ce type d'indication. Cependant, cette indication ne fournit qu'une mesure relative, c'est-à-dire que connaissant le temps d'exécution du sous-programme sur une structure de taille *N*, elle permet d'évaluer le temps nécessaire pour une structure de taille *2xN*. Ces éléments, relativement bien connus dans la littérature, doivent être mentionnés dans les différents cas de figure : pour une recherche, préciser s'il s'agit d'un temps moyen ou d'un temps maximal, les variations selon que la recherche aboutit ou échoue ; pour un tri, mentionner si le fait que les données en entrée sont (presque) triées ou non a une influence sur les performances, etc. Ne pas oublier non plus les contraintes en espace mémoire.

Reste le point sensible de la documentation des performances absolues. Nous ne connaissons pas de publications qui référencent ce problème, et les fabricants de composants «prêts à porter» semblent souvent discrets... Nous proposons la démarche suivante :

> Définir un système de compilation de référence unique dans l'entreprise. Il s'agit d'une machine bien définie, avec les caractéristiques de modèle d'unité centrale, d'horloge, de mémoire...
>
> Définir un ensemble de programmes de test standard. Cet ensemble de tests doit présenter un large panorama d'utilisation des fonctionnalités d'Ada. Se méfier des «benchmarks» connus (Whetstone, Dhrystone) qui ont généralement été adaptés à partir d'autres langages de programmation et ne sont donc généralement pas représentatifs du style de programmation Ada. Des suites d'évaluations spécifiques d'Ada (ACEC[1]...) peuvent être utilisées. Mesurer le temps d'exécution des tests standard sur le système standard.
>
> Mesurer le temps d'exécution des tests standard sur le système de compilation du projet. Le rapport avec le temps d'exécution standard donnera une idée du *rapport de puissance* des machines. Il n'est nécessaire d'effectuer cette mesure qu'une fois pour chaque système de compilation.

- Pour chaque composant, définir un programme de test standard en précisant bien les conditions, et documenter son temps d'exécution sur le système de compilation standard.

On obtiendra une première idée des performances du module en multipliant le temps d'exécution du test standard par le rapport des puissances des machines. Ceci permet d'éliminer de façon quasi certaine des composants dont le temps d'exécution est beaucoup trop important, ou d'accepter sans trop de risques des modules dont le temps d'exécution est manifestement faible.

Une meilleure idée des performances peut être obtenue en mesurant la vitesse d'exécution du test standard sur le système de compilation de l'utilisateur. Cette mesure est plus réaliste, car effectuée sur le système cible lui-même, donc en tenant compte des particularités du système de compilation, sans pour autant nécessiter d'écrire du code spécifique. Elle devrait permettre en particulier de choisir entre plusieurs composants de performances voisines.

Si une grande précision est requise, notamment dans le cas de systèmes critiques à «budget de temps» serré, il faudra alors que l'utilisateur développe et mesure son propre jeu de tests représentatif de son application. Noter que l'adaptation du test standard peut faciliter l'écriture de ces tests qui doivent alors être caractéristiques de l'utilisation faite du composant par le logiciel.

Ce système n'a pas la prétention de résoudre totalement le problème de la documentation des performances, et il est important de bien en comprendre les limites. Tout d'abord, le rapport des puissances des machines n'est pas uniforme, et peut varier pour une même machine selon les compilateurs : tel système, particulièrement performant pour la commutation des tâches, peut avoir des performances déplorables sur la génération de code des agrégats tableau ! Le profil, le style d'utilisation du langage peuvent avoir une influence sur l'efficacité relative des systèmes. C'est pourquoi le programme de test standard doit offrir un large éventail d'utilisation des fonctionnalités

[1] Ada Compiler Evaluation Capability

d'Ada. Ensuite, des phénomènes liés au matériel peuvent mettre en défaut l'hypothèse de linéarité. Dans le cas d'un logiciel qui boucle sur une structure de données qui tient entièrement en mémoire cache, la première boucle peut être plusieurs ordres de grandeur plus lente que les suivantes. Si la structure se met à déborder de la mémoire cache, les performances peuvent s'effondrer brusquement. Quant aux processeurs vectoriels, les surprises peuvent être encore plus importantes, selon qu'ils sont capables de «chaîner» ou non...

En résumé, la démarche que nous proposons ne dispense pas de l'écriture de tests spécifiques dans les cas critiques ; mais elle devrait permettre d'obtenir une base de décision plus objective *dans les cas où l'écriture de tests spécifiques ne se justifie pas*. Notons que l'essentiel du travail sera effectué par le responsable composants logiciels, de façon à fournir aux décideurs un maximum d'informations pertinentes sans leur imposer de coût supplémentaire.

### *16.2 .3 Exemple d'utilisation*

Il est bon de fournir un programme d'exemple associé à chaque composant logiciel, montrant la «bonne» façon de l'utiliser. Attention : celui-ci doit être différent du programme de test. Ce dernier doit servir à mesurer les performances, alors que l'exemple sert de modèle pour les utilisateurs potentiels.

### *16.2 .4 Documentation interne*

La documentation interne n'est destinée à être lue que par le responsable composants logiciels et son équipe. Elle est proche de la documentation interne de tout développement logiciel en ce qui concerne les documents de conception. Plus que partout ailleurs, il importe de mentionner clairement les décisions de conception qui ont été prises, et les différents compromis possibles avec les raisons des choix qui ont conduit à la solution finalement adoptée. Nous avons vu sur des exemples que les raisons des choix sont souvent complexes, et il faut éviter que des personnes nouvellement arrivées réessayent des solutions qui avaient été envisagées au début, puis abandonnées.

Ces documents doivent être complétés par une structure appropriée permettant de suivre et de contrôler les utilisations des différents composants.

## 16.3 Administration de la base de composants

### *16.3 .1 Validation de composants*

Tout composant doit être muni d'une batterie de tests destinés à vérifier son comportement, que nous appellerons sa *suite de validation*. Un seul test peut être communiqué aux utilisateurs : le test de performance «standard».

Les autres tests, gérés par le responsable composants logiciels, sont essentiellement destinés à vérifier le bon comportement du composant. L'assurance de non-régression en cas de nouvelle version est encore plus importante pour des composants logiciels que pour des logiciels fermés : en cas d'introduction d'une erreur, tous les projets utilisateurs risquent d'être affectés ! Tout rapport d'anomalie concernant un composant doit conduire à l'incorporation dans la suite de validation du test qui aurait dû diagnostiquer l'erreur, et un test ne doit *jamais* être retiré de la suite. En pratique, il est nécessaire d'associer à chaque composant une procédure automatisée (fichier *batch*) lançant tous les tests correspondants. La suite de validation doit obligatoirement être passée entièrement avant d'accepter une nouvelle version d'un composant.

## *16.3 .2 Gestion de configuration et suivi d'utilisation*

L'utilisation de composants implique que l'on soit capable, à partir du numéro de version d'un programme livré au client, de connaître exactement la version de chacun des composants logiciels utilisés. Il s'agit donc typiquement d'un problème de gestion de configuration, et de nombreux outils sont disponibles pour effectuer cette tâche.

Mais le partage de ressources qu'entraîne l'utilisation de composants logiciels pose le problème inverse. Si une erreur est diagnostiquée et corrigée dans un composant, cela affectera non seulement le programme qui aura détecté l'erreur, mais aussi tous les autres programmes utilisateurs du composant. Il faut donc être capable de retrouver les systèmes utilisateurs d'un composant donné. Cet aspect n'est pas toujours pris en compte par les systèmes de gestion de configuration, et nécessite une politique propre. Elle exclut en particulier de permettre aux utilisateurs d'employer «librement» une bibliothèque de composants, car on perdrait alors la trace des utilisations. Il faut donc mettre en place une politique de *droits d'accès* obligeant les utilisateurs à demander au responsable composants logiciels la mise à disposition de modules, et donc à se déclarer explicitement à lui. Plus difficile à obtenir (et à vérifier) : si un composant a été demandé, mais n'est finalement pas utilisé dans un système, il faut retirer le demandeur de la base des utilisateurs.

Une solution alternative serait d'obliger en fin de projet les développeurs à faire une liste des composants utilisés. Noter qu'encore une fois, cette pratique est courante dans les composants matériels : après avoir réalisé une carte électronique, l'ingénieur récapitule les éléments utilisés pour permettre la fabrication. Le problème est que cette liste n'est pas nécessaire pour la «fabrication» en série du logiciel, et risque donc d'être plus difficile à obtenir et à vérifier. On peut imaginer qu'à la fin d'un projet, le responsable effectue une recompilation finale dans une bibliothèque «vierge» dans laquelle on n'aurait mis que les composants effectivement déclarés par les concepteurs.

## *16.3 .3 Support des environnements de programmation*

La mise en œuvre des composants logiciels sera plus ou moins facile et nécessitera différents niveaux de contrôles manuels selon les fonctionnalités fournies par l'environnement de programmation. Un élément que l'on peut avantageusement mettre à profit est la notion de sous-bibliothèque. La plupart des environnements proposent plus ou moins cette fonctionnalité permettant de séparer une bibliothèque logique en plusieurs bibliothèques physiques. Selon les cas, il s'agira de simples liens entre bibliothèques ou de bibliothèques imbriquées avec des règles de visibilité sur les unités compilées similaires à celles des langages à structure de blocs.

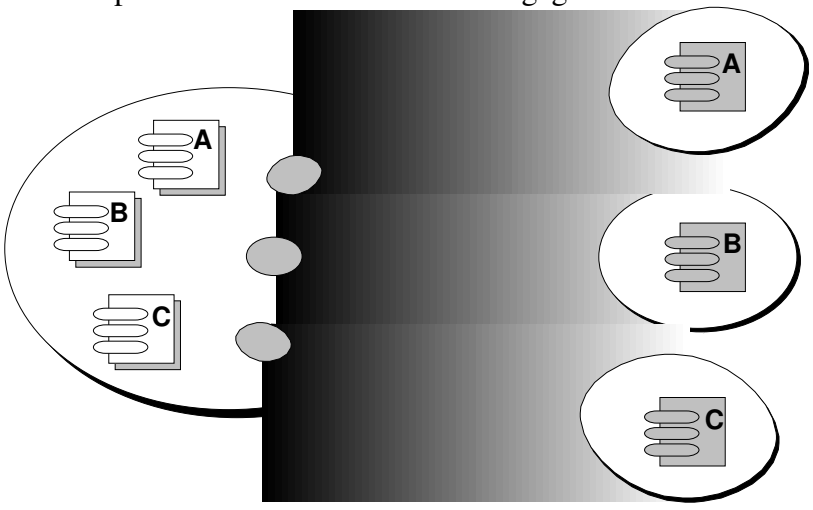


**Figure 31 :** Sous-bibliothèques

Par exemple, on peut utiliser une structure de bibliothèque imbriquée pour gérer différentes variantes dépendant de la cible, comme dans la figure 31. La bibliothèque principale contient les

spécifications communes aux différentes variantes, ainsi éventuellement qu'une implémentation portable. Des sous-bibliothèques contiennent des formes différentes d'implémentation pour des cibles particulières. En compilant depuis la sous-bibliothèque appopriée, on récupère la version la plus efficace selon la cible. Suivant le même principe, on pourrait gérer différentes versions des composants. Ce sera au responsable composants logiciels de déterminer la structure la plus appropriée, compte tenu des possibilités de l'environnement, du nombre de composants et de leurs variantes, et des besoins des projets. On ne peut établir de règle générale, si ce n'est qu'il faut utiliser toutes les possibilités de l'environnement.

Ce genre de possibilité a été extrêmement étendue dans certains environnements (Rational [Ler90]) : à chaque projet est associée une *vue* spécifiant les versions et les familles de chaque composant. La notion de vue permet alors de sélectionner automatiquement l'unité adaptée à chaque projet.

## *16.3 .4 Evolution des composants*

Comme tout logiciel, les composants sont amenés à évoluer dans le temps. Compte tenu de l'impact d'une modification de composant (surtout si la spécification en est affectée), il convient de maîtriser soigneusement cette évolution. Notons que dans leur enfance, les composants tendent à évoluer fortement : les besoins sont rarement bien définis au début, et ce n'est qu'avec l'expérience que l'on parvient à définir la «bonne» forme d'un composant. Avec le temps, ils tendent à se stabiliser. Une fois qu'un composant a été réutilisé plusieurs fois, en particulier s'il a pu être repris sans modification par des applications très différentes, il convient de le considérer comme figé et d'accueillir avec beaucoup de méfiance toute demande de modification.

### a) Evolutions compatibles

Ce type d'évolution ne remet pas en cause les applications existantes ; tout au plus risque-t-on des recompilations plus ou moins importantes, selon la chaîne des dépendances et les facilités de l'environnement de programmation. Le cas le plus fréquent est l'ajout de fonctionnalités : on rajoute des éléments dans une spécification de paquetage. Ceci *peut* créer des erreurs de compilation dans les modules plus anciens, si les nouveaux noms entrent en conflit avec des noms existants : cela doit attirer l'attention du responsable, car un conflit de nom peut révéler une duplication d'abstractions.

Des besoins nouveaux peuvent conduire à l'ajout de paramètres à une spécification de sous-programme. On maintiendra la compatibilité en utilisant une valeur par défaut qui corresponde à l'ancienne utilisation. Par exemple, si l'on disposait d'une fonction d'effacement d'écran :

```
procedure Effacer_Ecran;
```

et que l'on souhaite pouvoir choisir la couleur de l'écran à l'effacement, on peut modifier la spécification ainsi :

```
procedure Effacer_Ecran (En_Couleur : Couleurs := Noir);
```

Enfin, toute modification portant sur le corps des unités *sans changer la sémantique* (en particulier les améliorations d'efficacité) sera compatible.

### b) Evolutions incompatibles

Des modifications plus importantes de la spécification pourront entraîner des incompatibilités nécessitant la modification des programmes utilisateurs. Plus dangereux : des modifications des corps qui changent la *sémantique* de l'unité peuvent entraîner des incompatibilités d'exécution non détectées à la compilation. On évitera ceci en provoquant systématiquement une modification de la spécification (changement de nom...) lors d'une modification incompatible du corps. Toute tentative de recompilation provoquera des erreurs, qui préviendront les utilisateurs de la modification du

comportement. Il importe donc d'avertir les utilisateurs qu'une erreur de compilation provenant d'une modification d'un composant doit les inciter à se renseigner sur la cause de l'évolution, au lieu de modifier leur code «pour que ça passe».

Noter que plus les possibilités du typage Ada auront été soigneusement utilisées pour modéliser l'abstraction considérée, plus il est vraisemblable qu'une modification sémantique entraînera des modifications syntaxiques, donc des incompatibilités. Encore une fois, ces incompatibilités ne doivent *pas* être évitées, car elles procurent un degré de sécurité supplémentaire.

## 16.4 Recherche de composants

La recherche de composants logiciels s'apparente à la recherche documentaire... et souffre des mêmes difficultés. Il importe de fournir au développeur un moyen d'investigation puissant, sélectif mais pas trop, de la base de composants. Si une recherche n'est pas assez sélective, le nombre de composants identifié est trop grand et risque de décourager l'utilisateur ; une recherche trop sélective dépendra trop de la formulation de la demande, et courra le risque de ne pas identifier le composant alors même qu'il existe.

A la base de tout système de recherche de composants se trouve le problème de la classification. Selon [Pri91], un schéma de classification des composants logiciels doit répondre aux points suivants :

Il doit pouvoir gérer un nombre toujours croissant de composants.
Il doit permettre de retrouver des composants *similaires*, ne correspondant pas exactement à la demande.
Il doit permettre de retrouver des fonctionnalités équivalentes pour des domaines différents.
Il doit être très précis et puissamment descriptif.
Il doit être facile à maintenir ; la mise à jour et la redéfinition du vocabulaire ne doivent pas entraîner de reclassification totale.
Il doit être facile à utiliser par le responsable comme par l'utilisateur.

- Il doit être possible à automatiser.

Selon le rapport du groupe de l'ACM qui étudie ce problème [Sol91], différents systèmes ont été mis en œuvre pour la recherche de composants :

*Utilisation du système de hiérarchie de fichiers*. Des répertoires sont dédiés aux grands thèmes, des sous-répertoires aux sous-thèmes, des sous-sous-répertoires aux variantes d'implémentation. Cette solution, utilisée dans la PAL (*Public Ada Library*, bibliothèque de composants publics), permet d'obtenir une certaine organisation en l'absence d'outils spécifiques. Elle ne représente évidemment qu'un pis-aller.
*Système de recherche par mots clés et index*. Cette technique est directement inspirée de la recherche documentaire dont elle peut reprendre les outils. Le système SAIDA de CISI-Ingénierie appartient à cette catégorie. La difficulté réside surtout dans la définition des mots clés : l'idée que se fait l'utilisateur potentiel d'un composant ne correspond pas forcément à la présentation qu'en a fait le concepteur, ce qui entraîne le risque d'ensembles de mots clés disjoints, donc d'échec de la recherche alors même que le composant est disponible.
*Systèmes à facettes*. Il s'agit d'une recherche multicritères, selon des points de vue (facettes) orthogonaux. On a ainsi une description de l'élément recherché selon les divers modes de classification que nous avons étudiés. Cette méthode, qui correspond le mieux aux critères ci-dessus, a été utilisée aux Etats-Unis dans le cadre du projet STARS et du catalogue RAPID.
*Systèmes de base de connaissance*. Il s'agit de véritables systèmes experts fondés sur des réseaux sémantiques. La bibliothèque de réutilisation de Unisys utilise ce principe.

- *Hypertexte*, utilisé par Westinghouse.

Ces systèmes sont souvent expérimentaux, et la question de l'utilité de l'outil par rapport aux résultats (marteau-pilon pour écraser une mouche !) reste ouverte. Certains (comme Rose-Ada [Moi91]) visent également la gestion des éléments réutilisables autres que les modules Ada (conceptions, documentation). En l'absence d'outils exclusivement dédiés aux composants, la meilleure solution est d'utiliser conjointement plusieurs techniques :

Un système d'indexation par thèmes, mots clés, etc. Un système de gestion documentaire est parfaitement apte à jouer ce rôle. Une organisation hiérarchique peut suppléer ce système, surtout au début.

Une documentation papier, de type *data book*, régulièrement mise à jour et largement disséminée dans l'entreprise.

- Un responsable composants logiciels, connaissant très bien sa base de données et disponible pour conseiller les utilisateurs.

Seules l'expérience pratique et la mise à disposition de nouveaux outils permettraient de perfectionner ce premier dispositif.

## 16.5 Analyses de réutilisabilité

Développer une approche de conception fondée sur les composants logiciels nécessite d'intégrer des phases d'analyse de réutilisabilité en plus des étapes classiques. En fait, deux phases bien distinctes sont nécessaires : au début et à la fin du développement.

### *16.5 .1 Analyse a priori*

La phase d'analyse de réutilisabilité *a priori* a pour but de déterminer les composants disponibles utilisables par le projet. La première condition pour obtenir une bonne efficacité de cette phase est la bonne connaissance par les équipes de développement des composants disponibles. Le rôle du responsable composants logiciels est à cet égard déterminant.

La deuxième condition est un changement d'état d'esprit de l'équipe de programmation : il faut passer du «je réutilise si par hasard il y un composant qui convient» au «je n'écris du code nouveau que s'il n'y a pas moyen de faire autrement». Le programmeur doit voir son code comme une «colle» destinée à faire fonctionner des composants, et non comme un ensemble dont il maîtrise tous les niveaux d'abstraction. La formation joue donc un rôle prédominant pour l'évolution de l'état d'esprit.

En fait, l'analyse de réutilisabilité peut s'effectuer à deux niveaux. Lors des choix initiaux, les décisions de conception sont guidées par la disponibilité des composants. Il s'agit d'une influence globale des composants sur la structure du projet. En particulier, c'est à ce niveau que s'effectue le choix d'une «famille» dont l'utilisation sera imposée à tout le projet. Ensuite, lorsque l'on identifie un objet lors de la descente dans les niveaux d'abstraction, il faut chercher dans la bibliothèque si l'on dispose déjà de l'abstraction nécessaire, *ou d'une abstraction suffisamment voisine*, qui éviterait un développement à partir de rien. Attention : il ne s'agit aucunement de prôner une démarche ascendante de la conception de logiciel. Ce que nous disons ici, c'est qu'il faut *orienter* la démarche descendante pour lui permettre d'«atterrir» sur des composants existants plutôt que sur de nouveaux développements. Le mieux pour comprendre cette démarche est de reprendre l'analogie avec l'électronique.

Si l'on doit concevoir une carte logique complexe, on la décompose en unités logiques, elles-mêmes composées de sous-ensembles, puis de circuits. Il s'agit bien d'une analyse descendante. Il n'empêche qu'une connaissance des composants de base est nécessaire dès le début : on fait par exemple le choix d'utiliser une «famille» TTL, ce qui aura une influence sur les spécifications de l'alimentation électrique. De même, la structure globale est influencée par le fait qu'au bout du compte, les composants élémentaires seront des portes logiques. Une fois arrivé au

niveau des composants, l'électronicien fera son possible pour utiliser des composants existants *même s'ils ne correspondent pas exactement à ses besoins*. Si par exemple il a besoin de la porte indiquée à la figure 32(a) (porte AND inversant une des entrées, non disponible directement), plutôt que de commencer à fondre du silicium pour réaliser un composant spécifique, il préférera la réaliser comme indiqué à la figure 32(b) (une porte AND et un inverseur) ou même, solution moins évidente, mais utilisant une porte NOR plus fréquente qu'une AND, comme à la figure 32(c).

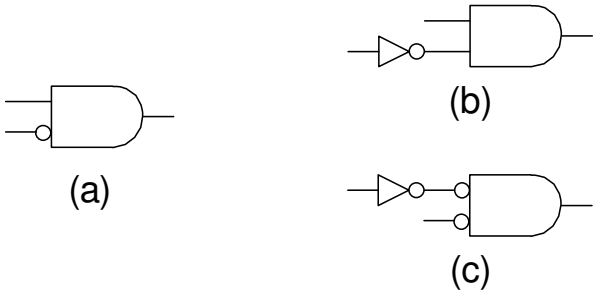


**Figure 32 :** Un composant électronique non standard

## *16.5 .2 Analyse a posteriori*

La phase d'analyse de réutilisabilité *a posteriori* a pour but de déterminer dans un projet terminé les modules susceptibles d'être récupérés et intégrés à la base de composants logiciels. L'identification de tels modules n'est pas évidente, d'autant plus qu'ils auront été développés dans un contexte particulier, et ne se présentent donc pas encore sous forme de composants réutilisables.

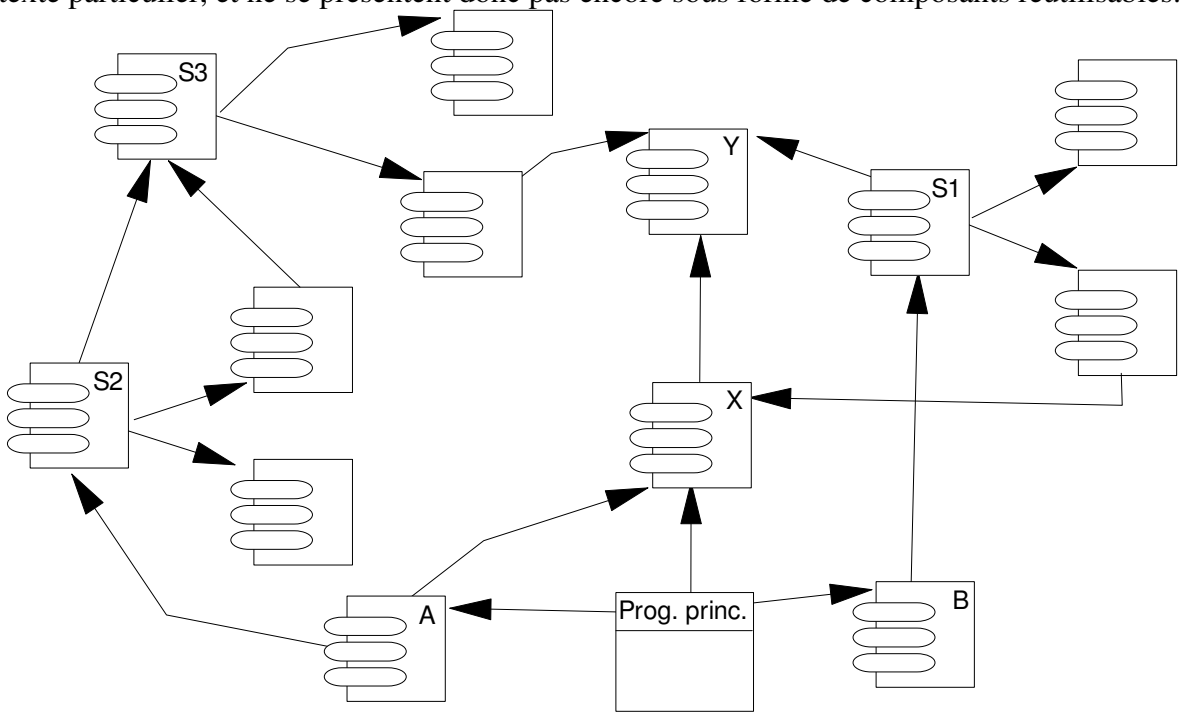


**Figure 33 :** Topologie de projet

Un bon moyen de les identifier est d'analyser la topologie du projet, c'est-à-dire le graphe des clauses **`with`**. Cette analyse est grandement facilitée si l'on dispose d'un outil graphique adéquat. Le graphe est en général très complexe, d'autant plus que les outils de représentation graphique des unités Ada se contentent souvent de tirer des flèches entre des boîtes sans souci de l'architecture sous-jacente. Il convient donc d'*organiser* la représentation graphique. La figure 33 représente une organisation (simplifiée) caractéristique d'un projet. Il est difficile *a priori* de reconnaître une structure dans un tel graphe. Cependant, on constate la présence d'un certain nombre de modules (comme X et Y) qui sont utilisés un peu partout dans le projet, sans aucun rapport avec les

différents niveaux d'abstraction. Ces composants seront habituellement des abstractions générales (chaînes, bibliothèques mathématiques), pouvant appartenir au domaine de problème particulier du logiciel, mais ayant une nature «fondamentale». Nous appellerons de tels modules des «bus logiciels», par analogie avec les bus d'alimentation des cartes électroniques qui alimentent tous les circuits, indépendamment de leur structure logique. De même, ces bus logiciels se caractérisent par le fait qu'ils sont utilisés dans tout le projet, indépendamment du découpage en niveaux d'abstraction.

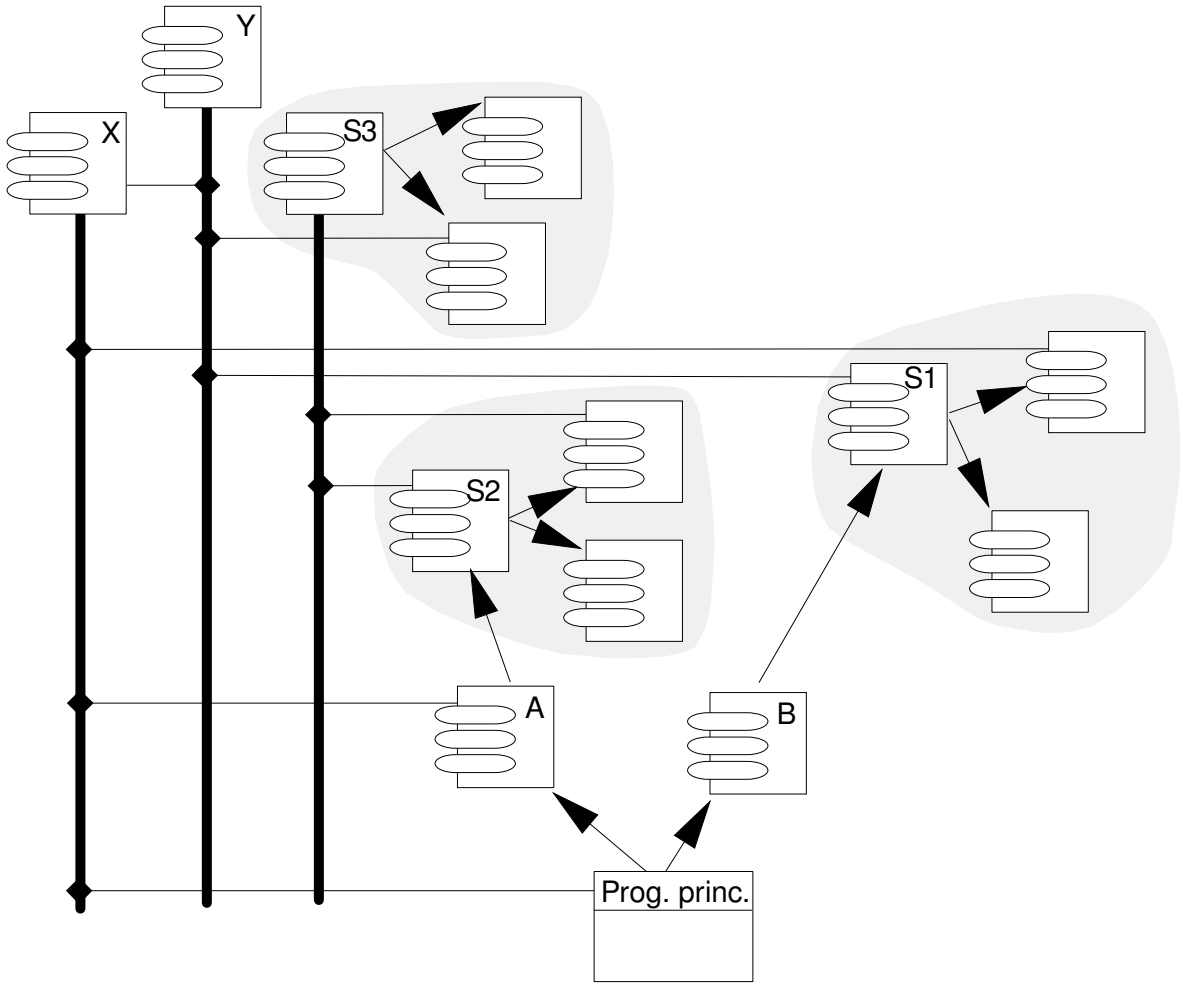


**Figure 34 :** Topologie de programme réordonnée

La représentation du graphe se simplifie grandement si, comme sur la figure 34, on représente effectivement l'utilisation de tels composants comme un «bus» sur lequel les autres composants viennent s'alimenter. De tels composants sont proches de la notion d'«objet d'environnement» que l'on trouve dans HOOD. S'ils ont été développés spécifiquement par le projet, ils sont de très bons candidats à la réutilisation. Noter que les composants d'une «famille» se reconnaissent aisément sur une telle représentation, puisqu'ils sont tous connectés sur le même bus. Inversement, le programme principal dépend d'un petit nombre de modules (hors composants bus), qui correspondent au premier niveau d'abstraction de la décomposition (A,B). Ces modules sont en général spécifiques de l'application et ont peu de chances d'être réutilisables.

Ces modules premiers utilisent des modules (S1, S2) qui se comportent comme des points d'entrée de sous-arborescences qui ne sont pas utilisés par des modules d'une autre sous-arborescence ; il s'agit là typiquement de sous-systèmes. Il est difficile de généraliser à ce niveau, mais ces sous-systèmes peuvent ou non être candidats à la réutilisation. S'ils sont réutilisables, ils constituent en général des abstractions de haut niveau. En tout état de cause, l'analyse de réutilisabilité doit porter sur le seul module d'entrée du sous-système. On remarque

qu'un composant bus peut être lui-même en fait un sous-système local (S3), et qu'il peut exister des dépendances entre bus (X vers Y).

A partir de cette représentation, il est possible d'identifier un nombre pas trop important de candidats à la réutilisation. Il faut inspecter ces modules pour déterminer s'ils correspondent bien à des abstractions clairement identifiées d'objets du monde réel et si ces abstractions ont un sens en dehors du projet qui les a développées. Si la réponse à ces deux questions est positive, il faut transformer le module en composant réutilisable. Le module de départ est souvent incomplet : le projet n'a développé que les fonctionnalités dont il avait besoin, alors que l'abstraction devrait logiquement en fournir d'autres. Le code doit être retravaillé comme nous l'avons vu pour parfaire la définition sémantique et le rendre plus robuste ; on parle de *durcissement* du composant. Ensuite, il faut étudier si la réutilisabilité du composant peut être accrue en le rendant générique. Il y a là un travail d'industrialisation tout à fait spécifique. Une fois le composant ainsi modifié, il risque d'être devenu incompatible avec le projet dont il est issu ! A ce point, il est en général trop tard pour faire les adaptations qui permettraient au logiciel d'utiliser la forme industrialisée du composant. Il faut conserver trace de cet état de fait et profiter de la prochaine révision du logiciel pour le réaligner sur le composant standard.

## 16.6 Exercices

1. Etudier comment organiser une base de composants logiciels destinés à l'enseignement. Adapter les conseils donnés dans ce chapitre au contexte particulier du milieu universitaire.
2. Tracer le graphe de dépendances d'un projet existant et le réordonner avec des bus logiciels.
3. Etablir les fiches descriptives, selon le modèle donné en annexe, des paquetages prédéfinis faisant partie de l'environnement standard Ada. Pour les parties dépendant de l'implémentation, se référer aux paquetages fournis avec le GNAT.

# 17

# Avantages et difficultés d'une politique de réutilisation

## 17.1 Avantages

Nous avons essayé tout au long de cette partie d'exposer les tenants et les aboutissants, les difficultés, les risques et les contraintes d'une approche du développement employant des composants logiciels réutilisables, et de montrer comment Ada fournit des outils de nature à simplifier cette tâche. Arrivé à ce point, le lecteur peut être saisi d'une crainte : tous ces efforts valent-ils réellement la peine ? La réponse est un *oui* franc et massif, si l'on considère non le seul coût, mais ce que l'on obtient par rapport à ce qui a été dépensé.

Une fois la structure mise en place et les composants disponibles, on peut considérer que 50 à 75 % du code «brut» d'un programme peuvent être obtenus à partir de composants réutilisés[1]. La raison en est que les aspects de présentation et d'interface utilisateur deviennent de plus en plus importants, et sont d'énormes consommateurs de code. Ajoutez à cela la gestion des structures de données, méthodes d'accès et manipulations courantes, et vous vous apercevrez que la partie réellement «noble» et nouvelle d'un projet logiciel est loin de représenter la majorité du code.

Non seulement le code réutilisé n'est plus à écrire, mais il a pu être développé avec plus de soins, puisque l'effort nécessaire pour le supplément de qualité est amorti sur plusieurs projets. Combien de projets nécessitant un petit tri pas vraiment critique utilisent-ils encore le *quick-sort*, qui n'a de «quick» que le temps pour l'écrire ? Un tri générique utilisera un algorithme de tri tournoi, notablement plus compliqué à programmer, mais tellement plus efficace. Quelle importance, puisque précisément on ne le récrit pas à chaque fois ? De plus, les modules ayant été utilisés, donc testés, dans de nombreuses configurations, le risque d'erreurs résiduelles est considérablement amoindri, pour ne pas dire supprimé à partir d'une dizaine d'utilisations. Enfin, en cas de problème, la maintenance et la réparation sont centralisées, et la correction bénéficiera automatiquement à toutes les applications utilisatrices[2].

Encore les remarques précédentes ne s'appliquent-elles principalement qu'aux composants développés par l'entreprise. Mais de nombreux composants sont disponibles aujourd'hui commercialement, pour un prix certes parfois élevé, mais faible devant ce qu'il faudrait investir pour développer soi-même l'équivalent. La présence de ces composants peut même conditionner la faisabilité d'un projet : il ne serait pas raisonnable de développer un projet sous X Window si l'on devait récrire toute l'interface.

---

1 Estimation personnelle.

2 A condition bien entendu qu'il s'agisse de réutilisation «telle quelle», et non d'une recopie avec modification d'un module dans le programme.

## 17.2 Difficultés

### *17.2 .1 Réticences psychologiques*

La première, et peut-être la plus grande difficulté à l'introduction d'une politique de réutilisation, n'est pas d'ordre technique, mais psychologique : le syndrome NIH (*Not Invented Here*). Le programmeur tend à se méfier de tout composant acheté à l'extérieur. Est-il aussi bien que ce qu'il aurait réalisé lui-même ? Même une bonne documentation peut ne pas être suffisante pour entraîner la confiance [Bau91]. S'il ne lui viendrait pas à l'idée de récrire une bibliothèque mathématique (parce qu'il ne s'en sent pas capable), acheter des composants à l'extérieur lui paraît gaspiller de l'argent... même si ceux-ci coûtent considérablement moins cher que le temps qu'il lui faudrait pour les développer. Parfois, la récriture peut être un prétexte pour utiliser un autre langage [Bau91]. La nécessaire discipline consistant à adapter la conception aux composants existants plutôt que l'inverse est une démarche nouvelle, peu répandue dans le domaine du logiciel. Enfin, l'approche fonctionnelle descendante permet mal l'identification de composants réutilisables ; une démarche objet, qui est encore loin d'être universellement acceptée, est indispensable pour permettre d'identifier ces composants.

Du côté positif, on note que les craintes qu'ont les programmeurs à réutiliser, qui relèvent beaucoup du fantasme, se dissipent lors de l'utilisation effective de composants de haute qualité. On peut même assister à un retournement psychologique : le programmeur peut avoir l'impression d'être en charge de la partie «noble» du projet, sans avoir à se préoccuper de l'«intendance». Lorsqu'une politique globale de réutilisation est effectivement mise en place, on voit l'apparition d'un effet majoritaire [Car91] : puisque tout le monde utilise les composants, les nouveaux arrivants sont beaucoup plus enclins à suivre la pratique générale. [Fav91] note cependant qu'il peut y avoir une déception lorsque la mise en œuvre des composants n'est pas très facile, avec un risque de rejet en bloc. Formation, valorisation et information forment donc la base de la préparation à l'introduction d'une démarche «composants»... avec la mise à disposition d'une bibliothèque de base de qualité.

Inversement, il faut veiller à ne pas dévaloriser non plus l'écriture de composants. Comme nous l'avons vu, cette écriture relève d'une spécialité, nécessitant une connaissance approfondie du langage et une excellente faculté d'abstraction. Il convient donc d'identifier les concepteurs de composants comme des spécialistes à part, indépendants des développeurs d'application. Ils n'ont pas à connaître les éléments du domaine de problème d'un développement particulier, mais doivent être capables au contraire de transcender les cas particuliers pour en tirer des éléments généraux. Développeurs de composants et développeurs d'application doivent ainsi s'estimer réciproquement comme des spécialistes compétents dans des domaines différents, dont les uns comme les autres sont indispensables à la bonne fin des projets.

### *17.2 .2 Problèmes structurels de l'entreprise*

Si les problèmes de personnes sont une facette importante de la mise en œuvre d'une politique de réutilisation, ils ne sont pas les seuls. C'est toute la hiérarchie de l'équipe de développement qui se trouve mise en cause. Le rôle du responsable composants doit être bien compris, en particulier par rapport au responsable qualité : comme lui, il apparaîtra transversal aux différents développements, mais plus directement responsable de modules codés. Le responsable qualité en revanche devra veiller à l'*utilisation* effective de composants réutilisables de préférence à des développements nouveaux ; il sera également responsable de la qualité des composants, mais non de leur écriture. L'équipe «composants» apparaît donc comme une sorte de projet permanent, transversal et délocalisé par rapport aux développements spécifiques.

Il faut établir de nouveaux circuits de communication : l'équipe composants doit être capable de conseiller rapidement le concepteur et de fournir une documentation à jour des éléments disponibles. Si l'accès aux composants est lent ou difficile, le concepteur risque d'être rapidement découragé.

Enfin, l'utilisation des composants est très directement liée à une méthode de conception orientée objet. Il importe donc d'accompagner l'introduction des composants par des cours de méthodologie appropriés et par une sensibilisation à la rentabilité que la réutilisation peut procurer.

### *17.2 .3 Outillage*

Aux besoins spécifiques de la réutilisation doivent répondre des outils spécialisés. Force est de reconnaître que l'on ne dispose actuellement que de peu d'outils spécifiques. Les gestionnaires de configuration ont été conçus dans une optique «projet» plutôt que «composants». Les systèmes d'archivage et de recherche qui prennent en compte les besoins particuliers des composants sont encore rares.

Les différents responsables devront donc utiliser des outils (systèmes d'archivage, gestionnaires de documents, ...) qui ne sont pas forcément parfaitement adaptés. Ceci ne signifie pas bien sûr qu'il ne faille pas utiliser d'outils, mais simplement que leur mise en œuvre dans un contexte de composants logiciels risque d'être moins aisée et moins adaptée que dans d'autres contextes. Ce manque d'outils risque donc de venir compliquer la tâche de ceux qui sont chargés de mettre en place une politique de réutilisation.

Mais avec la multiplication des composants commerciaux, le besoin d'outils devient de plus en plus explicite. Des projets comme STARS aux Etats-Unis ont commencé à s'attaquer au problème. Divers systèmes commerciaux commencent à voir le jour, et on peut espérer que ce problème s'atténuera dans un avenir relativement proche.

## 17.3 Conclusion

La réussite d'une politique de promotion de composants logiciels est un processus complexe qui nécessite la mise en œuvre conjointe de nombreux éléments. Comme le rappelait Jeffrey Sutherland (cité dans [Rem94]) :

> *Cinq conditions sont nécessaires : spécifier l'unité de réutilisation (composant logiciel) et l'encapsuler ; associer conception, documentation et code, et en maintenir la cohérence ; indexer les composants et les stocker dans un référentiel objet ; former à la réutilisation ; enfin préparer l'organisation à l'appliquer.*

L'utilisation d'un langage de programmation adapté comme Ada est un élément nécessaire, indispensable même, mais non suffisant. Il faut aussi réorganiser la structure des équipes de développement pour intégrer l'approche «composants» de façon globale. La présence d'un responsable spécialisé pour introduire le processus de réutilisation, mettre en place une structure appropriée et aider les développeurs est nécessaire. Les facteurs psychologiques ne doivent pas être négligés, et une formation aux techniques spécifiques de la réutilisation, en particulier à l'approche objet, est indispensable.

La documentation et la recherche des composants forment une part importante du succès ; reprenons une citation d'un rapport de la NASA sur la réutilisabilité :

> *Il faut qu'il soit plus facile de trouver, d'identifier et de comprendre un composant que de le construire à partir de rien.*

Cette partie du livre ne s'est intéressée qu'à l'aspect «composants» de la réutilisation, mais la réutilisation en général couvre d'autres concepts : réutilisation de spécifications, de conceptions, de

documentation... De gros efforts sont entrepris actuellement dans le domaine des environnements de développement, qui tentent d'intégrer dans une structure commune toutes les formes de réutilisation.

La démarche par composants logiciels est une évolution nécessaire (mais, encore une fois, non suffisante) pour le développement de logiciels fiables de grande taille. Il ne faut s'en cacher ni le coût, ni l'impact sur l'organisation même du développement de logiciel ; il s'agit d'un investissement réel, mais d'un investissement rentable.

# Quatrième partie

# Organisation de projet et choix fondamentaux

Conduire un projet, ce n'est pas seulement adopter une méthode de conception et écrire du code. Un développement s'effectue en équipe, et même en utilisant une méthode rigoureuse, des choix fondamentaux dont dépendra le succès du futur produit doivent être effectués, dès avant le démarrage du projet et jusqu'aux dernières étapes de la réalisation.

En fait, il faut être convaincu que la solution à un problème informatique n'est *jamais* unique, et que toute décision a des avantages et des inconvénients. L'informaticien doit donc être avant tout une personne capable d'arbitrer des compromis. Hélas, trop souvent il se précipite sur la première idée qu'il trouve au lieu d'évaluer les différentes possibilités. Souvent, il ne se rend même pas compte qu'il a fait un choix, tant la solution adoptée lui paraît «évidente».

Dans cette partie, nous allons donc expliquer l'organisation générale d'une équipe de programmation, puis détailler les principaux choix intervenant à différents niveaux du développement, afin de faire prendre conscience de tous les points qu'il importe d'étudier soigneusement avant de se lancer sur son clavier !

# 18
# L'équipe de développement

## 18.1 Les rôles à remplir

La réalisation d'un projet logiciel quelque peu important ne peut se faire qu'en équipe. Trop souvent, celle-ci ne comporte qu'un chef de projet (qui met parfois la main à la pâte) et des développeurs qui font la conception, le codage et la mise au point. En fait, dans toute équipe de développement, il existe un certain nombre de *rôles* qui doivent être remplis. Bien qu'ils puissent varier légèrement, on trouve généralement :

*Un chef de projet*, qui doit coordonner l'équipe et à qui reviendra la responsabilité de trancher lorsque plusieurs choix techniques sont possibles et que l'unanimité ne peut être obtenue parmi les concepteurs.
*Un responsable financier*, qui suivra l'état d'avancement du projet et l'évolution des dépenses par rapport au planning prévu (qu'il aura lui-même établi).
*Un gestionnaire de configuration*, chargé de la réception des modules, de leur intégration, et de l'historique des différentes versions ainsi que de leur archivage.
*Un responsable documentation*, pour suivre la production de la documentation, s'assurer de sa compréhensibilité et de son exactitude. Il sera également chargé de la rédaction de la documentation «utilisateur» du produit.
*Un responsable qualité*, qui doit définir les critères de qualité du projet et vérifier leur bonne application.
*Un responsable composants logiciels*, chargé aussi bien de mettre à disposition des autres membres les composants qui peuvent leur être nécessaires que d'identifier les éléments susceptibles de devenir des composants logiciels réutilisables par la suite.
*Un responsable communication* qui assurera la bonne circulation de l'information dans l'équipe, qui tiendra à jour les comptes rendus des réunions, et auprès duquel les autres membres devront retrouver les informations dont ils auront besoin.

- Et bien entendu les *développeurs* proprement dits, chargés de la réalisation informatique du projet.

La taille de l'équipe de développement ne permettra pas en général d'affecter une personne différente à chacun de ces rôles. Il n'empêche que chacun d'entre eux *doit* être rempli par quelqu'un ; plusieurs rôles peuvent donc être dévolus à la même personne.

## 18.2 Affectation des différents rôles

Le chef de projet est généralement choisi par des instances supérieures, ainsi qu'un ensemble de personnes chargées de réaliser le projet. C'est la première responsabilité du chef de projet d'affecter les rôles aux différents membres de l'équipe. Bien que ceci dépende fortement de la personnalité des

individus, on peut donner quelques directives générales à ce niveau, surtout quand la taille de l'équipe ne permet pas d'affecter une personne par rôle.

Le rôle du responsable financier peut être tenu par le chef de projet lui-même : surveillant «d'en haut» la marche générale du développement, il est en bonne position pour cela.

Le gestionnaire de configuration a un rôle très important, et souvent sous-estimé. Ce rôle peut souvent être tenu par le responsable qualité, qui peut également s'occuper de la gestion des composants logiciels. On évitera en revanche autant que possible de faire tenir ce rôle par quelqu'un qui serait également activement impliqué dans le développement : un certain recul par rapport à la conception est en effet nécessaire.

Le rôle du responsable documentation sera également impérativement tenu par quelqu'un qui n'est pas directement un développeur ; en particulier, on prendra soin que la documentation utilisateur soit rédigée par quelqu'un qui ne connaît *pas* directement les modalités techniques de la réalisation. Trop souvent, les contraintes imposées aux utilisateurs résultent des problèmes techniques rencontrés par la réalisation. Le responsable documentation devra en particulier veiller à refuser des solutions «simplificatrices» au niveau de l'implémentation qui apporteraient des complications au niveau de l'utilisation. Ce rôle peut également être tenu par le responsable qualité... sous réserve qu'on ne lui ait pas déjà affecté trop d'autres rôles.

Selon la taille de l'équipe, le rôle du responsable communication est attribué au chef de projet, à sa (son) secrétaire, au responsable qualité...

En tout état de cause, c'est la responsabilité du chef de projet de s'assurer avant tout que chacun des rôles est tenu par quelqu'un de compétent pour ce faire, qu'aucun rôle n'est présenté ni perçu comme moins «noble» qu'un autre, et que chaque responsable a effectivement compris l'étendue de sa mission.

## 18.3 Fonctionnement de l'équipe

Toute la difficulté du travail en équipe vient de ce qu'il est nécessaire d'établir des règles, des modalités de travail, des procédures à respecter, etc. Or le but final doit être l'efficacité de l'équipe dans son ensemble. Il importe donc de faire en sorte que le mode de travail établi soit une aide pour tous, et non une gêne. La première condition est que chacun des rôles soit rempli par quelqu'un de compétent. Ceci doit être fait de façon positive ; en particulier, si l'équipe de développement a été déterminée *a priori*, cela nécessite d'identifier les compétences de chacun et de les ventiler au mieux sur les différents rôles, sans établir pour cela de hiérarchie de valeurs : tel qui se révéle fort piètre développeur peut faire un excellent gestionnaire de configuration.

De même, les différentes étapes nécessaires à l'acceptation d'un module ne doivent pas être présentées comme des sanctions, mais comme une reconnaissance d'un travail bien fait. Le responsable qualité doit en particulier faire preuve de psychologie, notamment lorsqu'un module ne peut être accepté : il doit fournir son aide au réalisateur en cause, non le pénaliser. En tout état de cause, il faut être sûr que dans l'équipe, tout le monde sait *qui* est responsable de *quoi*. L'équipe fonctionnera au mieux si chaque membre perçoit les rôles de chacun des autres membres comme des services complémentaires concourant au succès commun, s'il s'adresse au responsable concerné pour résoudre les problèmes de son ressort (sans chercher à les résoudre lui-même), et si réciproquement il est prêt à aider les autres membres pour les services qui relèvent de ses compétences propres.

# 19
# Le choix de la méthode

Dans la deuxième partie de cet ouvrage, nous avons vu différentes méthodes de développement. Comme il n'existe jamais de remède miracle en informatique (pas plus qu'ailleurs), aucune méthode ne peut, à elle seule, résoudre toutes les classes de problèmes. Lors d'un nouveau projet se pose donc inévitablement le problème du choix de la méthode la plus appropriée. Ce choix est d'autant plus crucial qu'il conditionnera pour une grande part le succès ou l'échec du système. Enfin, choisir la «bonne» méthode n'est pas suffisant : il faut aussi choisir une méthodologie associée et s'assurer qu'elle est acceptée et respectée par ceux qui doivent l'utiliser.

Le choix d'une méthodologie doit donc s'effectuer en deux étapes : déterminer tout d'abord la grande classe de méthode : structurée, entités-relations, orientée objet par composition ou classification ; choisir ensuite la méthodologie proprement dite pour mettre en œuvre la méthode. Il existe deux possibilités à ce niveau : soit adopter une méthodologie existante, soit concevoir une méthodologie «maison». Quelle que soit la solution choisie, il faudra veiller à bien définir la méthodologie adoptée et à l'appliquer rigoureusement. Le but étant bien entendu d'éviter la troisième solution : annoncer que l'on suit telle ou telle méthodologie, mais laisser chaque membre de l'équipe l'interpréter ou apporter ses propres «variations» à sa guise (même si c'est pour des motifs - apparemment - tout à fait justifiés) ; car alors, c'est la porte ouverte aux incohérences et aux difficultés de maintenance.

## 19.1 Critères de choix d'une méthode

Il existe certes des critères objectifs de choix de méthode ; mais là plus qu'ailleurs, d'autres aspects, notamment psychologiques, entrent en ligne de compte : habitudes ou compétences existantes, disponibilité d'outils, formation des personnels... et mode[1]. Nous ne présenterons ici que les critères objectifs ; il appartient au chef de projet de les pondérer en fonction des autres facteurs, notamment de la *culture d'entreprise*. Certaines sociétés par exemple ont un savoir-faire issu directement de l'automatique. Une méthode par machines abstraites, soutenue par une méthodologie telle que Buhr, sera plus adaptée. D'autres ont une longue expérience en composition ou en classification : la tendance naturelle sera de poursuivre dans cette voie, surtout qu'il y a alors plus de chances de pouvoir réutiliser des parties de logiciels ou des composants provenant de développements antérieurs.

### *19.1 .1 Programmation structurée*

Un des principaux avantages des méthodes structurées est qu'il y est plus facile de prévoir les temps d'exécution, puisque l'organisation du logiciel reflète l'ordre d'exécution. On les préférera

[1] Le seul critère qu'il soit bon d'ignorer. Hélas, il est parfois plus important dans certaines décisions techniques qu'on ne pourrait le croire.

donc pour des logiciels ayant des fortes contraintes de prédictibilité des temps de réponse. On pourra également être amené à les utiliser lorsque le cahier des charges est purement fonctionnel : il peut alors être plus simple d'implémenter directement celui-ci que de l'adapter à la structure requise par d'autres méthodes. De toute façon, on veillera à ne les utiliser qu'avec des logiciels de taille modeste et pour lesquels la réutilisabilité n'est pas un critère primordial.

### *19.1 .2 Méthode orientée objet*

Les méthodes orientées objet sont les méthodes favorites actuellement de par leur aptitude à maîtriser la complexité et à produire des modules réutilisables. On les préférera sauf raison particulière justifiant une autre méthode.

Reste le problème du critère de décomposition verticale principal : composition ou classification. L'effet de mode conduisant à des «guerres de religion», il est particulièrement délicat de démêler les critères de choix réellement objectifs. Les considérations sur la complexité des dépendances font préférer la composition pour les logiciels de grande taille et à longue durée de vie. La composition se prête également mieux au développement par maquettage progressif que nous avons exposé précédemment, puisque celui-ci est fondé sur la notion de plans d'abstraction. Inversement, si le projet nécessite une écriture rapide et relève du domaine de la recherche, la plus grande dynamicité et la facilité de modification des couches hautes peuvent faire préférer la classification. En fait, il semblerait que les facteurs psychologiques soient prédominants dans ce choix : certaines personnes semblent raisonner spontanément par composition, alors que d'autres adoptent d'instinct une approche par classification. Nous avons pu ainsi assister à de véritables dialogues de sourds entre concepteurs qui se représentaient mentalement leurs objets de façon opposée... Quelle que soit la solution adoptée, il est donc primordial que chacun des membres de l'équipe comprenne l'approche choisie *et l'adopte*, même si cela ne correspond pas à son mode naturel de pensée.

### *19.1 .3 Entités-relations*

Les méthodes entités-relations tendent à donner la place prépondérante dans la conception aux *données*, les traitements n'intervenant ensuite que par rapport aux données. De plus, le modèle implicite (mais non obligatoire) est celui des bases de données relationnelles. On utilisera donc plutôt ces méthodes lorsque les données sont prépondérantes, ou lorsqu'une base de données doit constituer le cœur du système informatique.

Ces caractéristiques correspondent souvent aux applications de gestion, mais pas nécessairement. C'est le profil de l'application, et non le fait qu'elle soit classée (ou non) «gestion», qui doit amener à décider d'utiliser ces méthodes.

## 19.2 Choix d'une méthodologie existante

La solution la plus simple, une fois une méthode choisie, consiste à adopter une méthodologie existante. La méthodologie est publiée, on dispose donc de manuels de référence. On trouvera facilement des stages de formation pour les membres de l'équipe qui ne la connaîtraient pas ; enfin, il existe des outils de génie logiciel disponibles commercialement pour la mise en œuvre.

La taille du projet joue un rôle prépondérant dans le choix. D'une façon générale, plus une méthodologie est rigoureuse, détaillée et précise, plus elle est également lourde à mettre en œuvre. La quantité de documentation exigée par HOOD par exemple ne se justifierait pas pour un projet de 10 000 lignes de code (une méthode de Booch «de base» est alors suffisante), alors qu'elle devient indispensable au-delà de 100 000 lignes. Certaines méthodologies sont très (trop ?) générales. Par exemple, OMT peut exprimer aussi bien des diagrammes de composition que de classification. Il est

parfois souhaitable de fixer des limites à l'utilisation de certaines méthodologies afin de restreindre l'étendue des possibilités, donc de renforcer les contrôles. On préférera, surtout dans de grands projets, des méthodes définissant clairement une dimension verticale, telles que Rose ou HOOD, à d'autres qui tendent à représenter tous les objets du programme «à plat» et offrent moins de possibilités de contrôler la complexité par une découpe en niveaux étanches.

N'oublions pas non plus que si l'on peut choisir une méthodologie au niveau d'un projet, il est souvent nécessaire d'acheter les outils correspondants, et que la direction financière souhaiterait pouvoir les amortir sur plusieurs projets... Il convient donc de pondérer les exigences d'un projet particulier par celles des autres projets typiques de l'entreprise, afin de permettre de réutiliser aussi le matériel... et les conceptions elles-mêmes.

## 19.3 Définition d'une méthodologie d'entreprise

L'absence d'une méthodologie correspondant bien aux besoins spécifiques d'un projet ou de l'entreprise en général, parfois le prix des outils, peuvent conduire à la décision de définir une méthodologie propre. Bien sûr, cela suppose d'avoir un ou des projets de taille suffisante pour amortir l'effort supplémentaire de définition de la méthode. Dans ce cas, la meilleure solution consiste souvent à repartir d'une «grande» méthodologie bien établie, et à l'adapter localement aux besoins ou contraintes particuliers. Il s'agira donc généralement non de méthodologies complètement nouvelles, mais de «panachages» d'idées empruntées à plusieurs d'entre elles. Nous montrerons dans la cinquième partie comment l'on peut définir une telle méthodologie.

Si le fonds de départ est suffisamment proche d'une méthodologie commerciale, il est parfois possible d'utiliser ou d'adapter des outils existants. Sinon, il faudra développer ses propres outils, ce qui viendra grever lourdement le budget destiné à la définition de la méthodologie. Il existe cependant des «méta-ateliers» de génie logiciel, qui fournissent une base commune permettant à l'utilisateur de créer les outils supports de n'importe quelle méthode.

## 19.4 Réticences et difficultés

Quelle que soit la méthodologie adoptée, elle doit être comprise et acceptée par ses utilisateurs, sous peine de mauvaise mise en œuvre, avec des résultats inverses de ceux que l'on en attendait. Citons quelques raisons pouvant conduire à un problème d'acceptation :

*La méthode n'est pas bien comprise*. Elle exige par exemple le remplissage de documents de conception dont le concepteur ne perçoit pas toujours l'utilité. On court alors le risque que les documents soient remplis machinalement, parce que le chef l'exige, mais ils seront alors de peu d'aide pour les programmeurs de maintenance qui les reliront par la suite. La solution à ce problème réside bien entendu dans une formation adéquate. Il peut arriver également que le chef ait «trop» bien compris la méthode, et insiste sur son application au-delà du raisonnable et aux dépens des autres aspects du projet.

*La méthode ne correspond pas aux besoins*. Une méthode orientée plutôt temps réel mettra l'accent sur les propriétés temporelles, alors que les méthodes orientées gestion accordent plus d'importance aux données. Une méthode mal choisie va donc focaliser l'attention du programmeur sur des points qui ne correspondent pas à ses préoccupations principales. Une méthode trop lourde par rapport à la taille du projet sera également difficilement acceptée, mais inversement une méthode insuffisante pour un projet complexe risque de mettre en danger la faisabilité même du programme.

*L'utilité de la méthode n'est pas perçue*. En particulier, les programmeurs qui ont eu l'habitude de travailler sans méthode bien définie, dans des petites équipes (quand ce ne sont pas des programmeurs individuels), ont du mal à se plier à la démarche contraignante d'une méthode

formalisée[1]. Il convient donc de leur en faire sentir la nécessité, par exemple en les faisant travailler quelque temps à la maintenance d'un projet existant.

- *La méthode est mal utilisée*. Les gens formés à une ancienne méthode tendent à propager leurs habitudes précédentes dans la nouvelle[2]. Il est fréquent de rencontrer ainsi des dérives fonctionnelles dans des projets censés être orientés objet. Cela provoque des problèmes de conception, ou rend les outils incapables de fournir le service escompté. Là encore, une formation appropriée est le meilleur antidote.

En conclusion, mieux vaut une méthode moins bonne sur le papier, mais comprise, appliquée et acceptée par l'équipe de développement, qu'une méthode théoriquement parfaite, mais mise en œuvre de façon inappropriée.

## 19.5 Exercices

1. Quelle méthode serait la plus appropriée pour développer des interfaces graphiques ? Un logiciel de transactions bancaires ? Un programme de simulation de physique nucléaire ? Défendez vos réponses.
2. Le choix d'une méthodologie ressemble au choix de composants logiciels. Appliquer le raisonnement du paragraphe 16.2.1.a à la décision de choisir une méthodologie existante ou, au contraire, d'en définir une nouvelle.

[1] C'est souvent le cas des débutants frais émoulus de l'université.

[2] C'est également vrai des langages de programmation. On dit qu'un bon programmeur FORTRAN est capable d'écrire du FORTRAN dans n'importe quel langage...

# 20
# Politiques de projet

Dans ce chapitre, nous allons aborder les différents éléments qui interviennent dans la phase préliminaire que l'on pourrait baptiser *le projet avant le démarrage du projet*. Soulignons que dans les projets réels, celle-ci est souvent négligée : les choix correspondants sont alors effectués au hasard, sans réelle étude des différentes possibilités et des compromis nécessaires... avec le risque d'aboutir à des solutions fort éloignées de l'optimum.

## 20.1 Choix du langage de programmation

Le langage de programmation utilisé dans un projet doit résulter d'un choix délibéré. Même si l'auteur de ces lignes a une préférence particulière[1], il faut bien reconnaître que de nombreuses contraintes peuvent influencer le choix du langage : temps de développement par rapport à la durée de vie du projet, compatibilité avec l'existant ou héritage provenant de projets plus anciens... L'important étant (encore une fois) d'éviter les choix implicites : ceux qui ne sont motivés par aucune décision rationnelle et dont l'origine ne peut être tracée. Nous discutions ainsi un jour avec un chef de projet dont le développement s'effectuait en C++. Le dialogue s'établit à peu près comme ceci :

*Moi :* Pourquoi avez-vous choisi C++ pour ce développement ?
*Lui :* Parce que c'est ce que M. Xxx avait mis dans la proposition.
*Moi :* Et pourquoi M. Xxx l'avait-il mis dans la proposition ?
*Lui :* ??? Oh, il l'avait mis dans la proposition...

Je ne pus obtenir aucune autre justification pour ce choix technologique qui gouvernait tout l'avenir du projet (au demeurant fort important) que le fait que M. Xxx avait estimé que cela ferait bien dans la proposition !

### *20.1 .1 Le langage principal*

En général, il est préférable de n'utiliser qu'un seul langage pour le développement : moins de compétences nécessaires, uniformité du projet, pas besoin d'outils multilangages... Cependant, certaines contraintes particulières peuvent nécessiter d'utiliser plusieurs langages, et nous y reviendrons dans la section suivante. Mais même dans ce cas, l'un d'entre eux doit être choisi comme langage principal, les autres n'ayant qu'un rôle annexe. En effet, si du point de vue technique rien ne s'opposerait au mélange des langages[2], nous avons vu en première partie que le langage de

[1] Le langage Ada, vous n'aviez pas remarqué ?

[2] Des systèmes comme VMS, ou des environnements de compilation comme GCC mélangent très facilement les langages. Seul UNIX privilégie abusivement un langage (C) au détriment des autres.

développement amène avec lui une certaine philosophie, et le mélange des philosophies serait beaucoup plus difficile à gérer que le mélange des langages.

Unique ou principal, un langage doit être choisi, qui imposera sa façon de faire à l'ensemble du projet. Nous avons présenté tout au long de ce livre les avantages d'Ada pour la mise en œuvre des principes du génie logiciel. Voyons donc quelques raisons qui peuvent conduire à *ne pas* choisir Ada.

### a) Logiciels jetables

Ce terme désigne de façon imagée les logiciels que l'on utilise une fois, mais qui n'offrent plus d'intérêt une fois le résultat obtenu : on les appelle parfois logiciels «Kleenex», puisqu'on les jette après un seul usage. Pour de tels logiciels, la facilité et la rapidité de développement priment sur la maintenance (par définition, il n'y en a pas). La rapidité d'exécution est également sans importance, car le logiciel n'est exécuté qu'une fois, ainsi que la lisibilité puisque le programme ne sera jamais relu que par son auteur, et uniquement pendant le développement. Tout ceci désigne des langages interprétés, très interactifs, tels que APL dans le domaine des calculs mathématiques, ou SmallTalk pour maquetter des interfaces utilisateur.

C'est également le cas des *maquettes rapides*, dont nous rappellerons qu'elles *doivent* être jetées après usage : le risque est que l'auteur soit tenté de conserver le logiciel, puis de le développer, et l'on termine avec un logiciel devant être maintenu alors que rien dans la structure initiale (ni dans le choix du langage) n'a été prévu pour cela. Si un logiciel jetable doit donner naissance à un logiciel plus complet, il est plus rentable de le recoder entièrement en fonction des nouvelles exigences que d'essayer d'adapter l'existant ! Lorsqu'il s'agit d'explorer un domaine nouveau, il est donc raisonnable d'écrire une première version dans un langage adapté au développement rapide[1].

### b) Domaines très spécialisés

Certains langages ont été conçus spécifiquement pour certains domaines : c'est le cas de Lisp en intelligence artificielle, de Prolog pour les systèmes experts et de SQL pour les bases de données. Si l'application concerne massivement un de ces domaines, le langage spécialisé sera certainement le plus performant. D'ailleurs, coder une application typiquement «lispique» en Ada reviendrait quasiment à récrire un interpréteur Lisp en Ada (ce qui d'ailleurs se fait très bien, mais pourquoi refaire ce qui a déjà été fait ?).

Attention cependant : ces langages très performants dans leur domaine deviennent souvent catastrophiques dès qu'ils en sortent. Ecrire une interface utilisateur correcte en Prolog relève de l'acrobatie ou de l'exercice de style ! Il est donc particulièrement important avec ces langages de sous-traiter les éléments qui n'appartiennent pas à leur domaine à des modules écrits dans d'autres langages.

### c) Utilisation d'éléments ou d'outils existants

Parfois, on prescrit l'utilisation d'outils (générateurs d'interface, vérificateurs formels...) qui ne savent traiter qu'un seul langage, qui se trouve donc imposé d'office. Il arrive même que le client prescrive son langage pour des raisons qui lui sont propres. Le développeur n'a plus aucun choix, encore qu'il puisse parfois discuter de ces contraintes avec son client s'il estime que le choix imposé est de nature à mettre en péril la réussite du projet. Noter que la contrainte imposée par le DoD (et de plus en plus affirmée) sur ses fournisseurs d'utiliser Ada est de cette nature. Mais elle est renforcée par le fait que les études ont montré le bien-fondé de cette directive en termes de maintenance à long terme.

---

[1] Et donc fondé sur des principes opposés à ceux d'Ada : rapidité d'écriture plutôt que facilité de lecture, aucune considération pour la maintenance ni pour l'efficacité...

De façon plus discutable, le projet peut être amené à choisir un langage pour des raisons d'utilisation de composants logiciels écrits dans le langage considéré. Si l'utilisation de composants logiciels peut (et parfois doit) avoir un impact sur la structure du projet, il est possible d'écrire des interfaces permettant de récupérer des bibliothèques écrites dans d'autres langages. Notons qu'Ada 95 a fait un effort tout particulier pour faciliter l'utilisation de modules écrits dans d'autres langages, notamment en C. Il faut donc évaluer le gain provenant de l'interfaçage direct avec les composants contre le coût du choix d'un langage non nécessairement le plus adapté au projet lui-même. En particulier, compte tenu de l'existence d'excellentes bibliothèques d'interfaçage et d'un standard Ada/POSIX[1], le fait de développer sous UNIX n'est certainement pas une raison suffisante pour justifier l'utilisation du langage C !

## *20.1 .2 Cohabitation de plusieurs langages*

Il existe des cas où il est raisonnable d'utiliser plusieurs langages pour un même développement logiciel. Citons par exemple :

Reprise de bibliothèques ou de composants logiciels existants, développés dans d'autres langages. En particulier, il se peut que certains algorithmes critiques de sécurité aient été validés (souvent au moyen de méthodes formelles). On préférera alors les réutiliser plutôt que de les récrire et de devoir passer par toutes les phases de certification, toujours longues et coûteuses.

- Présence d'algorithmes relevant d'une technique particulière : interfaçage avec des bases de données, raisonnement formel ou déductif.

Dans ces cas, on a un langage principal qui fait appel à des services écrits dans d'autres langages. Dans le cas où ce langage principal est Ada, on connaît deux techniques pour assurer la liaison : l'interfaçage direct et les bulles.

L'interfaçage direct consiste à appeler directement des sous-programmes écrits dans les autres langages. On donne une spécification Ada, mais on déclare simplement au moyen d'un **`pragma`** `Import` que les corps correspondants ont été écrits dans l'autre langage. Ceci règle les problèmes de convention d'appel, mais ne garantit pas nécessairement la cohérence des données. Aussi Ada fournit-il le paquetage `Interfaces`, qui sert de base à des paquetages enfants, un par langage, décrivant en termes Ada les types de données prédéfinis de ces autres langages. La norme a standardisé les paquetages `Interfaces.C`, `Interfaces.FORTRAN` et `Interfaces.COBOL`. Grâce à cela, l'écriture des spécifications Ada correspondant à des bibliothèques d'autres langages est grandement facilitée.

Le pragma `Import` remplace le pragma `Interface` d'Ada 83. Son principal avantage est de s'appliquer non seulement à des sous-programmes, mais aussi à d'autres entités : variables, constantes, classes... Le pragma `Export` permet symétriquement de mettre des entités Ada à disposition d'autres langages.

Le terme de bulle a été avancé par Pitette [Pit87,Pit88] pour décrire la technique d'interfaçage du système AdLog (CR2A), visant à permettre à des applications Ada d'accéder à de la programmation logique. L'idée de base est que certains problèmes particuliers relevaient de techniques déductives, et que le langage Prolog était alors le plus approprié pour les résoudre. Cependant, un programme réel se limite rarement à ses seules parties déductives : il faut au moins une interface utilisateur. De même, certains systèmes de nature «temps réel» font appel par endroits à des techniques déductives : l'exemple typique étant un système de contrôle aérien utilisant une base de règles pour reconnaître les avions d'après leur signature radar. AdLog permet d'écrire ces algorithmes déductifs en Prolog, et un précompilateur associé les traduit en structures de données Ada. Celles-ci sont alors exécutées par un interpréteur Prolog, lui-même écrit en Ada. Ainsi, un programme physiquement

[1] Ainsi que d'interfaces Ada/X Window/Motif, Ada/CORBA/IDL....

«tout Ada» peut par moments faire appel à des résolutions isolées, les «bulles», écrites en Prolog (et mises au point par un interpréteur Prolog classiques).

D'autres interfaçages utilisent des techniques voisines sans employer la même dénomination, notamment SAME [Iso94]. L'idée de base est qu'une application de gestion comporte des parties purement procédurales couplées à des requêtes à des bases de données. SAME définit un langage, SAMEdL (*SAME Definition Language*), dont la syntaxe est proche d'Ada, mais la sémantique dérivée de SQL. Ce langage est facilement accessible à des spécialistes des bases de données chargés de réaliser la partie «requêtes». Un outil génère automatiquement l'interface vers le système de base de données et produit des paquetages dont les spécifications constituent des vues «Ada» des requêtes. La partie impérative peut alors être réalisée en Ada par des concepteurs ne connaissant pas nécessairement les subtilités des bases de données.

## 20.2 Organisation des bibliothèques de programme

La bibliothèque de programme joue un rôle central pour le développement en Ada. Son organisation est de la responsabilité du gestionnaire de configuration. Il devra étudier les différentes possibilités offertes par le compilateur et décider d'une structure adaptée au projet. Par exemple, si le compilateur dispose à la fois des notions de liens et de sous-bibliothèque, on peut décider d'avoir une bibliothèque principale pour les développements spécifiques du projet, et une autre (qui sera liée à la première) pour les composants logiciels stables et immuables. De plus, la bibliothèque principale contiendra les modules officiellement acceptés (après contrôle par le responsable qualité), alors que chaque développeur bénéficiera d'une sous-bibliothèque pour ses développements propres. Ainsi, chacun peut bénéficier des modules «officiels» de ses voisins et travailler sur ses propres modules en développement sans perturber les autres membres.

Bien entendu, ceci n'est qu'un exemple, et d'autres organisations sont possibles selon les besoins du projet et les possibilités du compilateur ; en particulier, le mécanisme des liens, des familles et/ou des sous-bibliothèques peut être mis à profit pour gérer les versions successives du logiciel, les différentes options de configuration (ou les différentes cibles)... Certains environnements de développement offrent des possibilités extrêmement évoluées en la matière. Il n'existe qu'un seul point absolu à retenir : l'utilisation des possibilités doit être *décidée*, dès le début du projet, et appliquée de façon uniforme par tous les membres de l'équipe. Bien entendu, comme tout choix fondamental, ces décisions d'organisation doivent être justifiées et référencées dans le document conservant les raisons des choix de conception.

## 20.3 Style et restrictions d'usage

Le style d'un projet se doit d'être uniforme : cela facilite la maintenance en permettant à n'importe qui d'intervenir sur n'importe quelle partie d'un projet sans risquer d'être dépaysé. De nombreuses études ont été publiées sur le sujet ; en ce qui concerne Ada, la meilleure base de départ est l'«Ada Quality and Style Guide» [Spc94], édité aux Etats-Unis par le Software Productivity Consortium et mis dans le domaine public, ce qui permet de l'utiliser sans problème. Il donne de nombreux conseils et présente les solutions les plus fréquemment retenues. Il convient cependant de l'«instancier», c'est-à-dire de choisir les modalités pratiques à utiliser en fonction des grandes lignes qu'il présente. Quelques points qu'il importe de fixer sont :

les détails de la présentation : indentation, retraits... Le plus simple est d'adopter la présentation du manuel de référence.

la convention d'utilisation des minuscules et des majuscules : généralement, on met les mots-clés en minuscules et les identificateurs avec une majuscule initiale. On peut décider de mettre les identificateurs globaux, ou tout au moins ceux reconnus comme particulièrement importants, entièrement en majuscules.

- les conventions de nommage. Il est par exemple fréquent de nommer les types avec des identificateurs commençant par «`T_`», ou se terminant par «`_type`».
- la décision d'utiliser (ou d'interdire) les clauses **`use`**. Ce point est l'objet d'un vif débat dans la communauté Ada, aussi y reviendrons-nous dans la partie concernant les règles de nommage.
- la forme des commentaires et leurs normes d'utilisation : commentaires standardisés en tête ou en fin de module rappelant les grandes fonctionnalités, l'auteur, l'historique, etc. ;
  commentaires utilisés pour repérer ou séparer les différents sous-programmes ; placement des commentaires par rapport à la séquence de code à laquelle ils se rapportent.

Suivant le domaine du projet, il peut également être nécessaire d'interdire l'utilisation de telle ou telle possibilité du langage. Par exemple, un logiciel devant fonctionner 24 heures sur 24 interdira l'usage de toute allocation dynamique, car même si l'on peut assurer que toute variable allouée est rendue au bout d'un temps fini, cela ne garantit *pas* que la mémoire ne s'épuisera pas, à cause des effets de fragmentation mémoire. Une telle restriction ne s'applique parfois pas aux phases de démarrage de l'application, où l'on peut créer des variables de façon dynamique, mais ces variables dureront de façon permanente. On rencontre également des restrictions sur l'emploi du parallélisme, totales (pas de tasking) ou partielles (pas de rendez-vous, ou au contraire pas de types protégés). Certaines contraintes de validabilité peuvent exiger également l'interdiction de fonctionnalités de haut niveau. Depuis Ada 95, il existe un **`pragma`** `Restriction` qui permet de faire vérifier par le compilateur la bonne application de ces restrictions d'usage. On peut ainsi écrire :

```
pragma RESTRICTION (No_Tasking);
   -- Pas de parallélisme
pragma RESTRICTION (No_Dynamic_Allocation);
   -- Pas de pointeurs

-- etc.
```

Il existe également des outils permettant de vérifier la bonne application des règles de style, ou de remettre un code «aux normes». La forme la plus courante de ce genre d'outil est le «reformateur» (ou «*pretty printer*»), fourni avec la plupart des compilateurs Ada, qui redonne une présentation correcte à un programme. Comme en matière de style l'uniformité est plus importante que les détails du style adopté, il est parfaitement raisonnable d'adopter la politique des outils dont on dispose. Le compilateur GNAT possède par exemple une option de compilation qui refuse toute unité dont la présentation ne correspond pas au style... adopté par les auteurs du compilateur. Bien qu'initialement destinée à l'écriture du compilateur lui-même, on peut parfaitement décider d'adopter cette présentation, d'ailleurs fort acceptable, pour la seule raison que l'on dispose du moyen de la faire vérifier automatiquement.

## 20.4 Politiques de gestion des erreurs

### *20.4 .1 Principes généraux*

Lorsque l'on parle de gestion des erreurs, l'on pense immédiatement «exceptions». C'est bien normal, puisqu'Ada a popularisé cette puissante fonctionnalité qui fait défaut à beaucoup de langages, même si le concept figurait déjà dans des langages antérieurs. Notons au passage que la plupart des langages ultérieurs (notamment C++ et Eiffel) ont également adopté des fonctionnalités de déroutement sur événements exceptionnels. On trouvera dans [Goo88] un résumé des principales contraintes liées à l'utilisation des exceptions. Il ne faudrait cependant pas croire que les exceptions soient *suffisantes* pour gérer les erreurs d'exécution : elles constituent seulement un mécanisme linguistique indispensable pour *implémenter* des politiques de gestion d'erreurs, qui doivent être définies en fonction du contexte du projet.

Pour bien comprendre le problème, revenons sur ce qu'*est* une situation exceptionnelle. Un sous-programme est chargé de rendre un certain *service*, pour un *client* (le sous-programme qui l'appelle). Or il est parfoirs impossible de rendre le service demandé : lecture s'il n'y a plus de données, inversion de matrice singulière, panne de périphérique... Ce sont ces cas que nous appelons des cas exceptionnels. Noter que ce ne sont pas nécessairement des *erreurs* : le fait d'atteindre une fin de fichier, par exemple, est une condition *exceptionnelle* (la boucle normale de traitement ne peut se poursuivre) sans être forcément une *erreur* (le cas est prévu dans le logiciel et constitue une situation normale). On peut dire qu'une condition anormale cesse d'être une erreur pour n'être plus qu'une exception lorsque la condition est propagée à un niveau qui l'a prévue et qui sait comment réagir pour la corriger ou la faire disparaître. La gestion des cas exceptionnels comporte trois aspects distincts : le diagnostic, le signalement et le traitement.

### a) Diagnostic

Le diagnostic consiste à reconnaître qu'une situation exceptionnelle s'est produite. A l'origine du diagnostic, il y a toujours un *test de condition logique*, mais selon les cas ce test peut se produire à divers niveaux d'abstraction : niveau matériel pour une division par zéro, test généré par le compilateur (débordements de tableau par exemple), test programmé par l'utilisateur.

Le diagnostic peut être *interne,* quand le sous-programme effectue lui-même le test, ou *externe*, lorsque l'on fournit à l'utilisateur une fonction d'interrogation permettant de savoir si le service pourra être rendu ou non. Par exemple, la fonction `End_Of_File` est un diagnostic externe permettant de savoir s'il reste encore des données dans un fichier, et donc s'il est possible d'effectuer une lecture. Si l'on effectue malgré tout une lecture alors que l'on est en bout de fichier, la procédure de lecture effectuera un test interne, qui aboutira à la levée de l'exception `End_Error`.

### b) Signalement

Le signalement désigne le mécanisme utilisé, une fois qu'une condition exceptionnelle a été reconnue, pour avertir le client que le service demandé n'a pu être rendu. Il peut être *synchrone* si le client détermine lui-même le moment où la condition anormale lui est connue, *asynchrone* sinon.

Les mécanismes synchrones présentent l'avantage que le client maîtrise mieux le déroulement de son programme ; en revanche, ils créent le risque de l'omission du test de condition anormale, et donc de poursuivre le traitement sans s'apercevoir qu'un service n'a pu être rendu, en violation du principe :

> *Il n'est qu'une chose pire qu'un programme qui se «plante» : un programme qui donne des résultats faux, mais vraisemblables.*

En cas d'appels imbriqués, un module ne peut rendre un service que si les fonctionnalités qu'il appelle se sont elles-mêmes terminées correctement : le composant *diagnostiquera* un problème, parce qu'un sous-programme appelé lui aura *signalé* une exception. La notion de diagnostic pour une couche correspond donc à la notion de signalement des couches plus profondes.

### c) Traitement

Le traitement consiste à réagir de façon appropriée lorsqu'une condition exceptionnelle a été signalée. Le traitement est *interne* s'il s'effectue à l'intérieur du sous-programme appelé, *externe* s'il est à la charge de l'appelant. Dans certains cas, il peut y avoir à la fois un traitement interne systématique et un traitement externe à la charge de l'appelant.

Que faire lorsqu'une condition exceptionnelle est diagnostiquée ? Il faut d'abord déterminer s'il s'agit d'une erreur ou non. Si ce n'est *pas* une erreur, c'est donc une condition qui fait partie de l'algorithme ; elle aura été prévue au niveau de la conception du cas général. Le cas le plus évident est celui de la lecture de données qui s'arrête en cas de fin de fichier. Cela peut être également le cas

d'exceptions plus «violentes» : il existe par exemple des algorithmes numériques rapides, mais qui risquent de déborder dans certains cas limites, et d'autres plus lents, mais sûrs. Une solution consiste alors à lancer l'algorithme rapide, et s'il échoue, l'algorithme lent :

```
begin
   Algorithme_Rapide;
exception
   when Constraint_Error =>
       Algorithme_Lent;
end;
```

Malgré son aspect «bestial», cette technique peut s'avérer statistiquement efficace.

Lorsque l'exception constitue une erreur, le logiciel doit s'efforcer d'atteindre un état bien défini. Il existe quelques grandes «classes» de réactions, qui ne sont d'ailleurs pas exclusives :

*Corriger*. La cause de l'erreur est connue et le programme sait prendre les mesures appropriées.

*Entrer en mode dégradé*. La cause de l'erreur est connue mais la correction n'est pas possible. On effectue une correction partielle, qui réduit les fonctionnalités du logiciel, mais permet de poursuivre... ou de se terminer «proprement».

*Informer*. On envoie un message à l'utilisateur, et on abandonne totalement l'opération demandée. C'est souvent ce que l'on fait dans les boucles les plus externes des programmes interactifs. Des variantes sont possibles, comme de demander à l'utilisateur une intervention externe (déprotéger la disquette...), lui proposer de réessayer, etc.

*Consigner*. On enregistre dans un fichier, un listing ou tout autre support l'apparition de l'erreur pour analyse ultérieure.

*Propager*. L'erreur empêche de poursuivre normalement : elle constitue donc un diagnostic, que nous devons signaler aux couches appelantes. La propagation est la réaction normale lorsqu'un incident dépasse les capacités de réaction de la couche qui la reçoit (je ne sais plus quoi faire, donc je repasse le problème au client).

### d) Politiques

Une politique de gestion d'erreurs consiste à définir précisément ce qui constitue une erreur, et comment celle-ci doit être gérée (diagnostiquée, signalée, traitée) par le logiciel. Comme il existe plusieurs formes d'erreurs (de l'erreur utilisateur «normale» à l'erreur fatale empêchant la poursuite de l'exécution du programme), ainsi éventuellement que plusieurs modes de traitement selon les caractéristiques du module qui l'a détectée, il faut définir une politique de gestion d'erreur pour chaque forme caractéristique. Nous avons ainsi vu que la seule possibilité dans le cas d'un composant logiciel réutilisable est de lever une exception[1]. Dans les autres cas, il est souvent nécessaire d'adopter un comportement plus évolué. C'est l'ensemble de ces politiques particulières qui définit la politique d'erreur générale du projet.

## *20.4 .2 Politique de correction locale*

Cette stratégie consiste à essayer de remédier au problème à la source : le service qui diagnostique un problème tente d'y remédier tout seul en interne : il n'y a donc pas signalement. Cette stratégie est fréquemment employée dans les langages sans exceptions : une fonction recherchant une sous-chaîne dans une chaîne renverra la valeur 0 si la sous-chaîne n'est pas trouvée, une fonction mathématique imprimera un message d'erreur au terminal en cas d'argument incorrect, etc. Comme il n'y a pas signalement, il ne peut y avoir traitement : le client n'a aucun moyen

[1] Ceci ne contredit pas ce que nous avons dit plus tôt sur la nécessité de définir une politique d'erreur : dans ce cas particulier, nous définissons une politique qui se *réduit* à la seule levée d'une exception.

d'influer sur la réponse à un incident, ni même de savoir qu'un incident s'est produit. Diagnostic, signalement et traitement sont donc totalement figés à l'intérieur de la procédure appelée.

### *20.4 .3 Politique du code de retour*

La première technique est celle du *code de retour* : on renvoie une valeur particulière qui indique si le traitement a pu être effectué. Le diagnostic est donc interne. Plusieurs formes sont possibles : utilisation d'un paramètre **`out`** :

```
procedure Traitement (Résultat : out Status);
```

valeur renvoyée par une fonction :

```
function Traitement return Status;
```

ou utilisation d'une variable globale :

```
Résultat : Status;
procedure Traitement;
```

La première forme, qui paraît plus logique, est peu utilisée en pratique, car elle oblige l'utilisateur à déclarer une variable supplémentaire,[1] et ne peut être utilisée (en Ada) avec des fonctions.

> Les fonctions Ada ne peuvent avoir que des paramètres **`in`**. La raison de cette contrainte supplémentaire est purement méthodologique.

La deuxième forme est utilisée quasiment systématiquement en C, où l'absence de typage rend cette forme parfois commode. Ainsi, une fonction peut renvoyer un pointeur, ou la valeur 0 (qui est également la valeur logique fausse !) si l'élément n'a pas été trouvé. L'inconvénient de cette façon de faire est qu'elle impose l'utilisation d'une fonction (par opposition à une procédure).

La troisième technique permet de garder les fonctionnalités sous forme de procédures ou de fonctions au choix de l'utilisateur, en n'imposant pas de déclaration de variable supplémentaire ; en revanche elle est difficilement utilisable en présence de parallélisme, car entre le moment où une tâche appelle la procédure `Traitement` et le moment où elle va lire la variable `Résultat`, une autre tâche a pu appeler la procédure, et rien ne garantit que le résultat lu corresponde à la bonne exécution ! Et pourtant, cette technique est largement utilisée pour les interfaces, en particulier avec POSIX qui renvoie toujours les résultats dans la variable `ERRNO`[2].

L'avantage du signalement par code de retour est qu'il est synchrone : l'utilisateur est maître de l'endroit où il effectue le test de bon achèvement du composant utilisé, et ce test apparaît explicitement dans le texte du programme. Le problème est que rien n'oblige l'utilisateur à effectuer le test, qu'il est très facile de l'oublier, et qu'alors le programme continuera en croyant qu'une certaine action a été effectuée, alors qu'il n'en est rien. Bien sûr, la forme «fonction» oblige l'utilisateur à effectuer le test, mais conduit souvent à une imbrication de **`if`** très lourde et pénible, non seulement à écrire, mais aussi à maintenir. Remarquer qu'en C, il est possible d'appeler une fonction comme si c'était une procédure, et que le résultat est alors perdu : l'utilisateur aura tendance à le faire pour des appels dont il est sûr qu'ils ne peuvent pas échouer... ce qui conduit à des catastrophes si le service échoue quand même pour des circonstances qui avaient échappé au programmeur.

---

[1] Et alors ? pensera le lecteur (avec raison). Il se trouve que l'état d'esprit de ceux qui utilisent cette technique les conduit en général à minimiser le nombre de lignes écrites.

[2] Ce qui a amené la norme IEEE 1003.1c, définissant le parallélisme, à définir `ERRNO` comme une fonction, contredisant ainsi la norme 1003.1a qui ne définit que les aspects séquentiels...

## *20.4 .4 Politique du déroutement*

La politique du *déroutement* consiste à appeler une procédure, fournie par l'utilisateur, lors de la survenue d'événements exceptionnels. On peut la voir comme une politique de traitement local, mais où le traitement est fourni par l'utilisateur au lieu d'être imposé d'office. Certains langages offrent cette politique en standard : en PL/I par exemple, l'utilisateur peut annoncer son intention de traiter les cas de division par zéro en déclarant :

```
ON ZERO_DIVIDE CALL MA_PROCEDURE;
```

En cas d'anomalie, le compilateur appelle la procédure déclarée, qui est censée remédier au problème, à la suite de quoi on reprend le traitement normal. Il est facile d'implémenter cette politique en Ada. Il suffit de déclarer :

```
package Gestion_Erreur is
   type Traitement_Erreur is access procedure;
   procedure Activer (Traitement : Traitement_Erreur);
   procedure Désactiver;
   function Traitement_Courant return Traitement_Erreur;

   procedure Signaler;
end Gestion_Erreur;
```

On active le traitement en l'enregistrant par la procédure `Activer` (équivalent de la clause `ON` de PL/I) et on le désactive par `Désactiver` ; la fonction `Traitement_Courant` renvoie un pointeur sur la procédure de traitement courant, ce qui permet de la conserver et de la restaurer plus tard. En cas de condition anormale, on appelle la procédure `Signaler`, qui appellera la procédure de traitement active. On peut prévoir une procédure par défaut (ou lever `Program_Error`) si aucune procédure de traitement n'a été prévue. On peut perfectionner ce paquetage en fournissant un paramètre à `Signaler` et aux procédures de traitement pour indiquer la cause de l'erreur : ce peut être une chaîne de caractères, un identificateur d'exception (`Exception_ID`, déclaré dans le paquetage `Ada.Exceptions`), etc.

Avec cette politique, le diagnostic est *interne*, le signalement est *asynchrone* : celui qui fournit la procédure ne sait pas *a priori* quand celle-ci sera appelée, et le traitement est *interne* : il s'effectue depuis le sous-programme appelé. Noter que la procédure de traitement peut décider de lever une exception, et que le composant doit y être préparé (et la laisser filer). Une difficulté est qu'il faut impérativement prévoir de «débrancher» la procédure dès que l'on sort de la zone où celle-ci s'applique, sous peine de voir un traitement prévu pour travailler dans un certain contexte appelé dans un contexte totalement différent. Enfin, l'activation étant exécutable, il n'est pas possible en lisant un morceau de programme de savoir quel est le traitement actif : il faut reconstituer tout l'historique de l'exécution du code.

Une variante de cette politique a été proposée par Kruchten [Kru89, Kru90], sous le nom de *politique sans exception*. Elle consiste à transformer les composants en génériques auxquels on passe une procédure utilisateur, appelée par le composant en cas de problème :

```
generic
   with procedure Traiter_Erreur;
package Le_Composant is...
```

Le signalement s'effectue comme dans le cas précédent par appel de la procédure fournie. Cette variante présente l'avantage que l'attachement d'un traitement est spécifique du composant, statique, et permanent dans la portée de l'instanciation. Il n'y a donc plus le problème de l'oubli du «débranchement» du traitement. Une solution intermédiaire consiste à fournir à la procédure appelée un pointeur sur une procédure à appeler en cas de problème :

```
type Traitement_Erreur is access procedure;
procedure Service (Autres paramètres...;
                   Erreur : Traitement_Erreur);
```

## *20.4 .5 Politique du contrat*

Cette technique, qui a été développée dans [Mey90], vise à empêcher les erreurs de se produire, en définissant un *contrat* entre l'appelé et l'appelant permettant de garantir la bonne fin du service rendu. L'idée de base est que le diagnostic doit toujours être *externe*. Chaque module déclare les préconditions qui doivent être respectées pour garantir son bon fonctionnement ; moyennant le respect des préconditions, celui qui fait le composant garantit la vérification de la postassertion. Pour une pile par exemple[1], la précondition de l'opération `Dépiler` est que la pile est non vide. L'opération `Dépiler` peut garantir la fourniture de la valeur du sommet de pile (postassertion) à la condition que la précondition soit vérifiée. La vérification du bon respect des préconditions est à la charge de l'appelant, en conséquence de quoi aucune erreur ne peut se produire. On doit donc écrire par exemple :

```
if not Pile_Vide then
   Dépiler (Valeur);
else
   Signaler_erreur;
end if;
```

Avec cette politique, le diagnostic est *synchrone*, puisque programmé par l'appelant, et confondu avec le signalement ; diagnostic, signalement et traitement sont entièrement à la charge de l'appelant. Ce que ne précise pas le modèle du contrat, c'est le comportement de la procédure `Dépiler` si on l'appelle malgré tout avec une pile vide : le résultat peut alors «légalement» être n'importe quoi : si l'appelant ne respecte pas sa part du contrat, alors l'appelé n'est plus tenu à rien. Cette méthode est donc basée sur un principe de confiance mutuelle : à l'intérieur du composant, on suppose que les pré-assertions tiennent, et on ne se préoccupe pas de ce qui pourrait se passer dans le cas contraire. C'est à l'appelant de vérifier avant l'appel le bon respect des préconditions, et s'il ne respecte pas son contrat, tant pis pour lui...

Hélas, ce système est totalement opposé au principe de programmation fiable. Si suite à une erreur dans un paramètre d'appel, votre composant a pour effet de «planter» totalement le programme, avec pour conséquence que l'avion dont c'était le système de contrôle va s'écraser avec ses 300 passagers sur les Champs-Elysées un jour de 14 juillet, votre responsabilité est dégagée : il ne fallait pas vous appeler de cette façon. Oui, certes, mais n'aurait-on pas pu spécifier *aussi* le comportement en cas d'appel incorrect, de façon à permettre de signaler une erreur, lancer une procédure de secours pour passer le système en mode dégradé, et éviter une catastrophe ?

Aussi cette politique ne dispense-t-elle pas d'une programmation défensive : les seules garanties que l'on a lors de l'appel d'un composant sont celles fournies par le langage. Ceci conduit généralement à une redondance de tests, puisque le composant vérifie que l'appelant a respecté sa part de contrat. En revanche, les signalements sont simplifiés, puisque tout diagnostic correspond à une faute de programmation de l'appelant : on se contentera en général de lever l'exception `Program_Error`. Cette exception ne sera en principe pas traitée par l'appelant, sauf pour sortir correctement du programme, en fermant tous les fichiers par exemple... et en demandant à l'utilisateur d'envoyer un rapport d'anomalie !

Les assertions jouent un rôle important dans cette politique, ce qui a amené Eiffel a introduire cette notion dans le langage lui-même. Cela n'a pas été jugé nécessaire en Ada, car il est extrêmement aisé d'écrire un paquetage fournissant cette fonctionnalité, au moins dans les cas simples. Les cas plus sophistiqués sont plutôt du niveau d'outils d'environnement, comme Anna [Luc90], qui permettent de rajouter des contrôles d'assertions à l'aide de commentaires spéciaux traités par un préprocesseur.

---

[1] Exemple dont on a tellement abusé que l'on est en droit de se demander s'il est possible de travailler formellement sur autre chose que des piles.

## 20.4 .6 Politique par exceptions simples

Cette politique, entièrement fondée sur le système d'exceptions du langage, consiste à signaler l'erreur par levée d'exception. L'utilisateur traite les erreurs dans des traite-exceptions. Le diagnostic est *interne*, le signalement est *asynchrone*, et le traitement est *externe* à la procédure qui effectue le diagnostic. Elle possède de nombreux avantages : sécurité, impossibilité d'ignorer la présence de l'anomalie, possibilité de sortir de nombreux niveaux imbriqués directement... C'est la seule possible pour des composants logiciels de base.

Elle présente cependant certains inconvénients lorsqu'elle est utilisée systématiquement à travers de nombreuses couches [Kru89, Kru90]. Le principe d'encapsulation veut que l'implémentation d'une fonctionnalité soit invisible à l'appelant ; or si celle-ci appelle des couches de plus bas niveau susceptibles de lever des exceptions, il faut empêcher la propagation de celles-ci, puisqu'elles proviennent d'un niveau invisible à l'utilisateur. Ceci conduit à une multiplication de traite-exceptions, en particulier dans les paquetages «relais» qui n'ont d'autre fonction que d'encapsuler, suivant le schéma ci-dessous :

```
package Haut_Niveau is
   procedure Service;
   Erreur : exception;
end Haut_Niveau;

with Bas_Niveau;
package body Haut_Niveau is
   procedure Service is
   begin
      Bas_Niveau.Service;
   exception
      when Bas_Niveau.Erreur => raise Haut_Niveau.Erreur;
   end Service;
end Haut_Niveau;
```

S'il y a consignation des erreurs, le problème est pire : en effet dans ce cas on exige que tout traite-exception consigne l'anomalie. Lorsqu'une exception traverse ainsi de nombreuses couches, on se trouve envahi par une avalanche de consignations qui ne signalent en fait que le passage de l'exception.

## 20.4 .7 Politique Oméga

Cette politique a été également proposée par Kruchten [Kru89, Kru90], et s'applique principalement aux types de données abstraits. Elle consiste à introduire un état supplémentaire décrivant si la donnée est valide ou invalide. L'état invalide est appelé «Oméga», par référence à la valeur `OM` du langage SETL [Sch86]. Toute opération portant sur des données «Oméga» fournit un résultat «Oméga». Une fonction permet de savoir si une valeur est dans l'état «Oméga» ; une variante non exclusive consiste à lever une exception dans ce cas. Un type entier ainsi protégé s'exprime comme :

```
package Entier_Protégé is
   type Entier is private;

   function Vers_Entier  (X : Integer) return Entier;
   function Vers_Integer (X : Entier)  return Integer;

   function "+" (Gauche, Droite : Entier) return Entier;
   function "-" (Gauche, Droite : Entier) return Entier;
   -- etc..

   function Est_Oméga (Valeur : Entier) return Boolean;
   Erreur_Oméga : exception;
   procedure Erreur_si_Oméga (Valeur : Entier);
```

```
private
   type Entier is
      record
         Est_Oméga : Boolean := False;
         Valeur    : Integer; -- par exemple
      end record;
end Entier_Protégé;
```

L'intérêt de cette méthode est qu'il n'est plus nécessaire de prévoir des traite-exceptions sur toutes les phases de calcul intermédiaires ; le signalement est *synchrone* (par appel de la procédure `Erreur_si_Oméga`, ou de la fonction `Est_Oméga` si on ne veut pas d'exception), mais le traitement est *externe*. La fonction `Vers_Integer` lève l'exception `Erreur_Oméga` si on l'appelle sur la valeur «Oméga», ce qui garantit qu'il est impossible de faire quoi que ce soit d'une valeur incorrecte ; on évite donc ainsi l'insécurité généralement liée au signalement synchrone (si on oublie d'effectuer le test).

## *20.4 .8 Politique de gestion centralisée*

Une autre possibilité pour limiter les phénomènes de cascade d'erreurs est la gestion centralisée des erreurs : lors de toute anomalie, au lieu de lever une exception, on appelle une procédure chargée de signaler l'erreur. Ceci peut s'exprimer sous la forme :

```
package Gestion_Erreurs is
   procedure Erreur (Message  : String := "");
   function Message_Erreur return String;
   Erreur_Traitée: exception;
end Gestion_Erreurs;
```

La procédure `Erreur` stocke le message, consigne l'anomalie si c'est demandé par le projet, puis lève l'exception `Erreur_Traitée`. Les traite-exceptions de la chaîne d'appel traitent les autres exceptions par appel de la procédure `Erreur`, mais laissent filer l'exception `Erreur_Traitée`, ou interrogent la fonction `Message_Erreur` pour connaître la cause de l'erreur d'origine. L'erreur n'est consignée qu'une fois, au moment de l'appel, et il n'est pas nécessaire de transformer une erreur provenant de couches profondes, car tous les modules ont la visibilité du paquetage `Gestion_Erreurs`.

Cette politique a les mêmes caractéristiques que celle par exceptions simples : diagnostic *interne* et signalement *asynchrone*. Elle évite cependant les effets de cascade en divisant le problème en deux : le traitement systématique est *interne* et préalable au signalement, ce qui laisse la possibilité d'un traitement complémentaire *externe*, dépendant du contexte de l'appelant.

## *20.4 .9 Récapitulatif des différentes politiques*

Le tableau de la figure 35 récapitule et compare les principales caractéristiques des politiques que nous avons mentionnées. Il rappelle la répartition des responsabilités entre le client (l'appelant) et le fournisseur de service (l'appelé). Toutes les combinaisons ne sont bien entendu pas possibles : par exemple, un traitement interne implique nécessairement un diagnostic interne.

| | **Diagnostic** | **Signalement** | **Traitement** |
|---|---|---|---|
| Correction locale | Interne | Asynchrone | Interne |
| Code de retour | Interne | Synchrone | Externe |
| Déroutement | Interne | Asynchrone | Interne |
| Contrat | Externe | Synchrone | Externe |
| Exception simple | Interne | Asynchrone | Externe |
| Oméga | Interne | Synchrone | Externe |

| | | | |
|---|---|---|---|
| Gestion centralisée | Interne | Asynchrone | Interne/Externe |

**Figure 35 :** Les différentes politiques d'erreurs

## 20.5 Exercices

1. Imaginer un cas (justifié) où le langage principal ne serait pas Ada, mais où Ada serait utilisé comme langage secondaire.
2. Justifier les choix de présentation vérifiés automatiquement par le compilateur GNAT (voir la documentation fournie avec le compilateur).
3. Estimer le coût relatif des différentes politiques de gestion d'erreur du point de vue des performances. Ecrire *ensuite* des programmes de test et comparer les mesures aux prévisions.

# 21

# Les choix de la phase de réalisation

Les choix que nous avons vus jusqu'à présent sont typiquement effectués au démarrage du projet. Mais tout au long de la conception et du codage, on est encore amené à prendre des décisions ; là encore, il importe que celles-ci résultent d'une analyse délibérée et non de ce qui passait par la tête du programmeur à ce moment.

Dans le chapitre précédent, nous avions situé les problèmes généraux et montré comment Ada permettait d'y répondre ; en revanche, nombre de choix exposés dans ce chapitre ne s'offrent qu'avec Ada. Est-ce qu'Ada pose plus de problèmes ? Non, c'est seulement qu'Ada est un langage plus riche, et que le programmeur dispose de nombreuses possibilités là où d'autres langages lui imposent une seule façon de faire. La «simplicité» dont se vantent certains langages n'est en fait qu'un moyen de cacher la pauvreté des moyens offerts.

## 21.1 Identificateurs et règles de nommage

Choisir des bons identificateurs est un point important qui est loin d'être évident. Ada offre de nombreuses possibilités, encore faut-il les utiliser à bon escient. Remarquons que tous les caractères d'un identificateur Ada sont significatifs et qu'il n'existe pas de limitation à leur longueur[1].

Nous allons commencer par un exemple et, exceptionnellement, nous utiliserons des identificateurs anglais. Nous verrons plus loin que l'utilisation d'identificateurs en français complique encore le problème !

Supposons que nous ayons quelque chose qui s'appelle un «contexte» (*context*), que nous voulions empiler (*push*) à l'entrée d'une procédure, pour pouvoir le restaurer à la sortie. Une expression informelle du problème serait :

*Empiler l'ancien contexte (push the old context)*

Si nous voulons que l'expression Ada ressemble à cette phrase, nous pouvons le faire de différentes façons :

```
Context.Push (Old);               -- (1)
Push (Old_Context);               -- (2)
Push_Context (Old);               -- (3)
Push (Context => Old);            -- (4)
Push (Context'(Old));             -- (5)
Push_Old_Context (X);             -- (6)
```

Ainsi que les combinaisons des précédents :

```
Context.Push (Old_Context);                    -- (7)
Context.Push_Context(Context => Old_Context);  -- (8)
-- etc.
```

[1] ... ou presque. Si le compilateur a le droit de mettre une limite, celle-ci ne peut être inférieure à 200 caractères, ce qui est plus que suffisant !

En (1), nous avons un paquetage appelé `Context` contenant une procédure `Push`, et une variable `Old`. Dans les autres exemples, nous ne nommons pas le paquetage ; nous supposons que la clause **`use`** a été utilisée. En (2), la procédure s'appelle `Push` et la variable `Old_Context`, alors qu'en (3) la procédure s'appelle `Push_Context` et la variable `Old`. En (4), la procédure s'appelle également `Push`, mais la variable s'appelle `Old` et nous profitons d'une association nommée pour exprimer que nous empilons un contexte. En (5), la procédure s'appelle `Push` et la variable `Old`, mais nous utilisons la qualification par `Context`, dont nous supposons que c'est le type de la variable `Old`. En (6), la procédure s'appelle `Push_Old_Context` et ce nom porte toute l'information ; le nom de la variable (`X`) est sans importance.

Quel est le meilleur choix ? Si vous n'êtes pas convaincu que ce n'est pas évident, faites un sondage parmi vos collègues ; il est vraisemblable que chaque solution aura la préférence de quelqu'un...

### *21.1 .1 Considérations générales*

Une règle fréquente est que le nom d'une entité[1] doit être «parlant» et «suffisamment long». Un indice clair de nom mal choisi est quand le programmeur ressent le besoin de mettre un commentaire avec la déclaration d'une entité pour décrire ce à quoi elle sert. Les noms doivent se comprendre d'eux-mêmes ! Bien qu'il y ait une forte part de subjectivité dans l'appréciation de la lisibilité d'un nom, rappelons quelques règles générales :

> Lorsqu'un nom est formé de plusieurs «mots», utiliser le trait bas (caractère '_') pour séparer les mots. L'usage de majuscules dans ce but peut conduire à des ambiguïtés, comme ce fichier de nombres entiers dont le nom était FichierDentiers[2]...
>
> > En Ada, le trait bas est toujours autorisé dans les identificateurs. Dans beaucoup d'autres langages, il peut l'être ou non à la discrétion des compilateurs, ce qui interdit son usage dans les composants portables. Les utilisateurs de langages à syntaxe «C» (notamment Java) tendent à éviter l'utilisation du trait bas, alors même qu'il est autorisé par le langage.

- Les procédures expriment des actions, et doivent être désignées par des verbes ; les fonctions retournent des valeurs et doivent utiliser des noms caractéristiques de la valeur retournée, ou des formes verbales de la forme «`Est_`...» pour les fonctions à résultat booléen. De même, les variables doivent avoir des noms caractéristiques de la valeur contenue.
- Et bien sûr, il faut éviter les abréviations.

Il existe même des outils vérifiant que tous les noms d'un programme ont au moins cinq lettres et appartiennent à un dictionnaire. Ceci fonctionne correctement en général, mais n'est pas une règle suffisante. Une formulation plus subtile serait :

> *Un nom doit porter toute l'information nécessaire au lecteur pour comprendre sa signification et pas plus.*

En rajouter plus qu'il n'est nécessaire a un effet néfaste sur la lisibilité. Par exemple, `Line_Length` est un identificateur excellent, mais `The_Length_Of_The_Line_That_I_Am_Considering` est absolument désastreux. Une autre règle «évidente» est :

> *Un nom doit exprimer la nature de l'entité désignée.*

Ceci est moins simple qu'il n'y paraît. Par exemple, [Boo84] utilise un paquetage `Complex` qui déclare le type `Number` ; ceci fait de `Complex.Number` un nom très parlant. Mais le paquetage

---

[1] Nous utiliserons ce terme pour dénoter tout ce qui peut avoir un nom en Ada, plutôt que le terme «objet» qui a trop de significations.

[2] Cet exemple n'a *pas* été inventé pour le «gag» : il figure effectivement dans la norme Pascal.

lui-même *n'est pas* un complexe ; le vrai complexe, c'est le type `Number`, ce qui est totalement trompeur. L'erreur provient ici de ce que le choix suppose *a priori* une forme d'utilisation du nom. Dans un composant logiciel, le concepteur ignore comment le nom sera utilisé. Ce qui nous amène à la règle suivante :

> *Bien qu'une des formes d'utilisation du nom puisse être préférable, toutes les autres doivent être acceptables.*

Avec l'exemple précédent, nous pouvons appeler le paquetage `Complex_Handler` et le type `Complex_Number`. On aurait alors les déclarations suivantes :

```
X : Complex_Number;                    -- préférentiel
Y : Complex_Handler.Complex_Number;    -- acceptable
```

Noter que nous favorisons ici l'utilisateur de la clause **`use`**.

## *21.1 .2 Utilisation de la surcharge*

Doit-on choisir un nom spécifique pour chaque opération, ou doit-on essayer de réutiliser des noms existants en profitant du mécanisme de surcharge ? Dans notre exemple initial, est-il préférable de nommer la procédure `Push` (qui pourrait être surchargée pour des piles d'autres choses que des contextes), ou `Push_Context`, qui serait unique dans le système ? Le but de la surcharge est l'abstraction : *se concentrer sur les similitudes en ignorant pour un temps les différences*. La notion d'empilement est la même, qu'il s'agisse de contextes ou d'entiers ; le nom `Push` est donc préférable.

Prenons un autre exemple. Supposons que nous ayons une procédure qui efface (*clear*) l'écran et une autre qui réinitialise (*clear* également) une variable. Les spécifications suivantes sont autorisées par le langage :

```
procedure Clear;                                    -- L'écran
procedure Clear (V : out Some_type);   -- La variable
```

Ici, les deux `Clear` ne désignent *pas* la même notion. Dans ce cas, nous recommandons d'appeler la première `Clear_Screen`. On pourrait garder le nom `Clear` pour la seconde, car l'argument obligatoire montrerait clairement ce que la procédure réinitialise. Nous pouvons en déduire la règle :

> *Les opérations spécifiques doivent porter des noms spécifiques ; les opérations générales doivent utiliser des noms généraux.*

## *21.1 .3 La question de la langue*

Faut-il donner aux entités un nom anglais ou utiliser le français (ou l'allemand, ou ...) ? La réponse dépend du contexte général du projet. En faveur du français, on trouvera essentiellement le fait qu'il est plus compréhensible... pour les Français, qui parlent souvent assez mal les langues étrangères. Bien que ce soit l'inclinaison naturelle de beaucoup de programmeurs, cela a cependant un certain nombre d'inconvénients. Lorsque l'on développe un composant logiciel réutilisable, l'anglais est quasiment obligatoire : sinon, le composant serait inexportable en dehors du marché français, sauf à maintenir deux versions de chaque composant. La maintenance peut être simplifiée avec un outil qui traduit automatiquement les identificateurs au moyen d'un dictionnaire[1], mais l'effort reste important. Une solution plus pratique est de ne dédoubler que la spécification (la seule qui importe à l'utilisateur). Le corps n'utilise que les noms français, avec quelques **`renames`** pour assurer la compatibilité avec la spécification. Les seules modifications au corps en cas de

[1] Un tel outil (Adasubst) est librement disponible depuis le site Adalog (http://www.adalog.fr)

changement de langue de la spécification seraient dues à la concordance des corps de sous-programmes avec leurs spécifications.

Cette question est moins cruciale pour les développements spécifiques, puisque le problème de l'exportation des *sources* ne se pose pas. Mais il existe d'autres raisons ; l'utilisation du français *diminue la lisibilité*, tout au moins pour les personnes maîtrisant suffisamment l'anglais, du fait qu'Ada utilise l'anglais pour les mots réservés. L'instruction suivante est presque du langage naturel en anglais :

```
if Device_is_not_ready then Call_operator ...
```

alors que sa traduction rappelle la nature «informatique» de la formulation :

```
if Periph_non_prêt then Appeler_opérateur ...
```

Enfin, il faut bien reconnaître que la structure de l'anglais se prête mieux aux formulations concises et que le fait de mettre les adjectifs avant les noms correspond mieux à la notion de qualification. Si `Complex.Number` «sonne» bien en anglais, on ne saurait en dire autant de `Complexe.Nombre`...

## *21.1 .4 Discussion de l'exemple*

Après avoir vu ces différentes règles, critiquons les différentes possibilités de notre premier exemple.

- (1) se présente bien et est facile à lire. Mais il implique que le paquetage s'appelle `Context`, or le paquetage *n'est pas lui-même le contexte*, en violation de notre première règle. Le paquetage serait plutôt un *gestionnaire de contexte* (`Context_Handler`) ; mais écrire `Context_Handler.Push(Old)` détruirait l'expression «naturelle» du problème. De plus, si l'utilisateur n'emploie pas une notation complète, `Push(Old)` n'exprimerait pas toute l'information.
- (2) et (3) ont bonne allure. Noter que si l'on utilise une notation complète, cela donnera `Context_Handler.Push(Old_Context)`, ce qui est encore acceptable. L'argument en faveur de la surcharge jouera en faveur de (2).
- (3) et (5) souffrent du même problème que (1) : si l'utilisateur n'emploie pas la notation prévue par le concepteur, ils se réduisent à `Push(Old)`.
- (4) fait une hypothèse inadmissible sur l'utilisation de la procédure ; rien ne dit que l'on souhaite empiler un «vieux» contexte. Si l'utilisateur veut empiler un contexte temporaire, la formulation deviendrait `Push_Old_Context(Temporary)`, ce qui serait contradictoire.

Nous ne discuterons pas de toute la combinatoire des solutions précédentes, mais notons tout de même que (8) ne saurait être qualifié de lisible !

## *21.1 .5 Démarche*

Compte tenu des remarques précédentes, on voit que le choix d'un nom, surtout pour les entités exportées, ne doit pas être fait à la légère. Nous suggérons que le concepteur considère les points suivants pour faire son choix :

*Choix du langage*. L'anglais de préférence pour les composants commerciaux. Pour les composants d'entreprise, suivre la règle générale de l'entreprise.
*Identifier la nature de l'entité* que l'on nomme pour choisir un nom pertinent pour cette entité, *indépendamment du contexte.*

*Identifier si l'opération est générale ou spécifique* pour décider d'un nom unique ou tenter de réutiliser d'autres noms existants exprimant la même fonctionnalité sur des types différents.

- *Faire des essais de nommage* avec les différentes possibilités (avec ou sans utilisation du nom complet, avec ou sans utilisation des paramètres nommés) pour vérifier qu'aucune combinaison n'est inacceptable.

## *21.1 .6 Utilisation de la clause «use»*

L'utilisation ou l'interdiction de la clause **use** est un vaste sujet de débat dans la communauté Ada [Bry87] [Boo86] [Ros87]. Nous allons tenter de le présenter succinctement et d'en tirer quelques conclusions.

> Les entités définies dans la partie visible d'un paquetage ne peuvent être utilisées qu'en les préfixant avec le nom du paquetage. La clause **use** ouvre la visibilité et permet de les nommer directement. Ada 95 a ajouté la clause **use type** qui rend les *opérateurs* d'un type (tels que "+" et "-") visibles, sans ouvrir la visibilité aux autres éléments. Une clause **use** (ou **use type**) peut figurer dans n'importe quelle partie déclarative et n'a d'effet que pour la durée de vie de cette portée.

Notons tout d'abord que le débat n'est pas tant celui de la clause **use** que celui de savoir si l'on doit utiliser des *noms complets* (préfixés par le nom du paquetage) ou des *noms simples*. En effet, rien n'empêche d'utiliser les noms complets même en présence d'une clause **use**, tout comme il est possible (par surnommage) d'utiliser des noms simples sans clause **use**. Disons simplement que si un projet choisit d'imposer des noms complets partout, alors il doit interdire les clauses **use** car cela permet de faire vérifier par le compilateur la bonne application de la règle.

### a) Les raisons d'être de la clause «use»

On aurait pu imaginer que le simple fait de mettre un **with** donne la visibilité directe[1]. Mais alors, on courrait le risque d'avoir trop d'identificateurs visibles simultanément et un souci constant dans la conception d'Ada a été de limiter et de contrôler l'espace des noms. C'est ce qui a donné naissance à ce mécanisme à deux niveaux : la clause **with** exprime les dépendances entre modules pris globalement : elle ne donne donc qu'une indication grossière de l'utilisation. La clause **use**, en n'ouvrant la visibilité directe que là où c'est nécessaire, permet de dire précisément où tel ou tel paquetage est effectivement utilisé. Imaginons par exemple un paquetage «type de donnée abstrait», fournissant un type et des opérations, dont des entrées-sorties :

```
package Gestion_Couleurs is
   type Couleurs is (Rouge, Bleu, Jaune, Blanc, Noir);

   function "+" (Left, Right : Couleurs) return Couleurs;
   procedure Put (Item : Couleurs);
end Gestion_Couleurs;
```

Pour implémenter la procédure `Put`, il faut importer `Text_IO` au moyen d'une clause **with** sur le corps du paquetage. Cependant, aucune autre partie de ce corps n'en a besoin ; ceci s'exprime naturellement en écrivant :

```
with Text_IO;
package body Gestion_Couleurs is
   function "+" (Left, Right: Couleurs) return Couleurs is
      ...
   end "+";
```

[1] C'est ce qui se produit lorsque l'on importe une «unit» en Turbo-Pascal par exemple.

```
   procedure PUT (Item : Couleurs) is
      use Text_IO;  -- il n'y a qu'ici qu'on l'utilise
   begin
      Put (Couleurs'Image (Item));
   end PUT;
end Gestion_Couleurs;
```

De cette façon, on documente pour le lecteur les endroits précis où l'on a besoin de `Text_IO` et l'on limite les visibilités inutiles. Comme d'habitude, le compilateur peut également mettre à profit ce supplément d'information pour effectuer des contrôles supplémentaires ; par exemple, si à la suite d'un couper/coller malheureux une instruction `Put` se retrouve en dehors de la procédure, cela produira une erreur de compilation, puisque la procédure n'est visible directement que dans la portée du **`use`**.

### b) Quelques bonnes raisons d'utiliser les noms complets

Fort malencontreusement, dans beaucoup d'ouvrages (y compris le manuel de référence) les exemples de **`use`** portent globalement sur toute une unité de compilation, ce qui a conduit de nombreux programmeurs à assortir systématiquement les **`with`** du **`use`** correspondant. L'effet obtenu est exactement inverse de celui recherché par les concepteurs du langage : tout devient visible. D'autre part, selon la structure du logiciel et les outils disponibles, il peut être plus ou moins facile de retrouver à quel paquetage appartient une entité. Un des avantages de l'approche objet est justement que tous les aspects liés à une certaine entité appartiennent à un même module ; avec une bonne conception et des identificateurs bien choisis, le simple nom de l'entité doit permettre de retrouver où elle a été déclarée. Il existe également des environnements évolués où le simple fait de «cliquer» sur le nom d'une entité ouvre une fenêtre sur le paquetage où elle a été déclarée. Si l'on ne dispose pas de tels outils, ou si la structure du logiciel rend plus difficile l'identification des entités, l'utilisation de noms complets peut faciliter l'identification de l'origine. N'oublions pas non plus le cas des logiciels devant faire l'objet d'une certification, comme ceux de type «aviation». Le certificateur[1] veut être absolument certain de connaître avec précision les éléments appelés. La notation nommée lui garantit l'absence de toute ambiguïté due, par exemple, à des surcharges.

Enfin un bon principe de programmation est qu'une opération s'appliquant à un objet précis doit mentionner le nom de cet objet. Si l'on appelle un sous-programme qui est une opération d'un type de donnée abstrait, pas de problème : l'objet auquel il s'applique figure dans les paramètres. Mais s'il s'agit d'une opération d'une machine abstraite, l'objet auquel elle s'applique est représenté en fait par le paquetage dans lequel il est déclaré : il est tout à fait logique dans ce cas de préfixer l'opération par le nom du paquetage.

### c) Quelques bonnes raisons d'utiliser les noms simples

Un nom simple est naturellement plus lisible qu'un nom complet ; comparez par exemple les deux formulations suivantes :

```
Float_Text_IO.Put (Fonctions_Mathématiques.Sin (X));
Put (Sin (X));
```

La longueur n'est pas seule en cause : on a remarqué depuis longtemps que dans le processus de lecture, l'attention se focalise d'abord sur le début du mot. C'est pourquoi dans la plupart des langues, les marques (accessoires) de déclinaison, conjugaison, etc. sont situées à la *fin* des mots. Malheureusement, la notation pointée fait exactement le contraire : la partie la plus intéressante de la notation (la fonctionnalité appelée) se trouve à la fin. Il paraît donc logique de nommer les entités par leur nom simple. D'autre part, les noms simples, avec le mécanisme de surcharge, renforcent le principe d'abstraction en focalisant l'attention sur la fonctionnalité logique et non sur les différences de réalisation. Si par exemple nous écrivons :

[1] A qui incombe la lourde responsabilité d'autoriser le logiciel à contrôler un avion avec 300 personnes à bord...

```
Integer_Text_IO.Get (I);
Float_Text_IO.Get (F);
```

nous attirons l'attention du lecteur sur la différence entre ces deux procédures, alors qu'elles réalisent la même chose : l'impression d'un nombre. Si nous avions *vraiment* voulu montrer cette différence, il suffisait de ne pas utiliser la surcharge et d'appeler les sous-programmes `Integer_Get` et `Float_Get`. La situation est encore pire pour les instanciations de certains génériques :

```
type Longueur is new Float;
package Fonctions_sur_Longueur is new
   Ada.Numerics.Generic_Elementary_Functions (Longueur);

type Temps is new Float;
package Fonctions_sur_Temps is new
   Ada.Numerics.Generic_Elementary_Functions (Temps);
```

Cela n'aurait aucun sens de rappeler explicitement au lecteur que la fonction `Sin` portant sur `Longueur` se trouve dans un paquetage différent de la fonction `Sin` portant sur `Temps`.

> Le paquetage `Ada.Numerics.Generic_Elementary_Functions` fournit les fonctions élémentaires (`Sin`, `Log`, etc.) sur n'importe quel type flottant et doit être instancié sur chaque type particulier, selon le même mécanisme que les entrées-sorties. Ce paquetage, qui était défini par une norme séparée en Ada 83, a été intégré dans la norme Ada 95 et est donc fourni par toutes les implémentations.

Enfin, la présence de bibliothèques hiérarchiques rend l'utilisation de la clause **`use`** encore plus nécessaire : les noms des préfixes en son absence tendent à dépasser nettement ce qui est acceptable.

### d) Critères de choix

Sans vouloir relancer tout le débat, rappelons ici quelques éléments à prendre en compte pour décider d'une politique d'utilisation de noms simples ou de noms complets.

Si la découpe est fonctionnelle et conduit à des difficultés de localisation des éléments utilisés, on peut imposer des noms complets, sauf si l'on dispose d'outils d'environnement permettant de localiser aisément les identificateurs.

- Si l'on a de fortes contraintes de certification, on peut exiger que tout nom soit préfixé par le nom du paquetage dans lequel il apparaît. On peut quand même utiliser la clause **`use`** pour ne mentionner que le nom du dernier enfant dans le cas d'unités hiérarchiques. Si on n'a pas ou peu fait usage d'unités hiérarchiques, il est possible de faire vérifier le bon respect de la règle par le compilateur en interdisant la clause **`use`**.
  Les opérations définies dans des paquetages de type «machine abstraite» seront préfixées par le nom du paquetage.
- Dans les autres cas, on préférera l'utilisation de noms simples. On limitera cependant la portée des clauses **`use`** à la plus petite zone de visibilité où l'on utilise effectivement le paquetage.

Il est possible d'imposer localement des contraintes supplémentaires. Ainsi, dans l'exemple de classification de la deuxième partie, nous avions les paquetages `Employé`, `Salarié`, `Commissionné`, etc., qui déclaraient chacun un type `Instance` et un type `Classe`. L'intention était que le programmeur les utilise toujours sous la forme `Employé.Classe` ou `Salarié.Instance`. Ceci est vérifié par le compilateur : même en présence de clauses **`use`**, ces types (volontairement) identiques vont se cacher les uns les autres et le programmeur sera bien obligé d'utiliser les noms complets[1].

[1] Sauf dans le cas particulier où un module n'utiliserait qu'une seule classe... mais l'intérêt de faire de la classification avec une seule classe est plutôt douteux ! Et de toute façon, il n'y aurait plus aucune ambiguïté possible.

## 21.2 Choix des types

La définition des types est une des étapes les plus importantes de l'écriture d'une application en Ada. C'est la rigueur du typage qui permet d'assurer le maximum de contrôle par le compilateur ; mais inversement, le programmeur (nouvellement) converti à Ada tend à définir *trop* de types, provoquant ainsi une multiplication des conversions nuisant en définitive à la lisibilité.

Prenons un exemple typique : soit une procédure `Aller_En`, destinée à positionner le curseur en un point de l'écran. Cette procédure admet à l'évidence deux paramètres, correspondant aux coordonnées. Une première solution consiste à définir des types différents pour les abscisses et les ordonnées :

```
type Abscisse is range 0..799;
type Ordonnée is range 0..599;

procedure Aller_en (X : Abscisse; Y : Ordonnée);
```

Cette solution présente l'avantage de faire vérifier par le compilateur que l'on ne mélange pas les coordonnées. Ainsi si l'on écrit :

```
   X_courant : Abscisse := 1;
   Y_courant : Ordonnée := 1;
begin
   Aller_en (Y_courant, X_courant);
```

le compilateur diagnostiquera que les paramètres ont été inversés dès la première compilation.

L'inconvénient de cette solution est que l'on a souvent à effectuer des calculs mixtes portant à la fois sur des lignes et des colonnes. Par exemple, pour calculer le périmètre d'un rectangle, nous devons évaluer l'expression `2*((Y2-Y1)+(X2-X1))`. Quel est le type de ce calcul ? Certainement ni une abscisse, ni une ordonnée. Ce ne serait pas homogène du point de vue physique et de plus le résultat peut prendre des valeurs en dehors de l'intervalle défini pour nos coordonnées. Le programmeur se rabat souvent dans de tels cas sur un type prédéfini, comme `Integer` et la formule devient `2*(Integer(Y2-Y1)+Integer(X2-X1))`. Mais rien n'assure que le type `Integer` convienne et en particulier qu'il comporte suffisamment de valeurs.

La seconde solution consiste à définir un seul type «coordonnées», d'intervalle suffisamment grand pour tenir non seulement les valeurs d'abscisse et d'ordonnée, mais encore les résultats des calculs intermédiaires entre abscisses et ordonnées. Pour améliorer les vérifications, nous définirons Abscisse et Ordonnée comme des *sous-types* de Coordonnées :

```
X_max : constant := 799;
Y_max : constant := 599;
type Coordonnées is
   range -(X_max+Y_max) .. (X_max+Y_max);
subtype Abscisse is Coordonnées range 0 .. X_max;
subtype Ordonnée is Coordonnées range 0 .. Y_max;

procedure Aller_en (X : Abscisse; Y : Ordonnée);
```

Ici, nous perdons toute vérification à la compilation d'éventuelles inversions entre abscisses et ordonnées, puisque précisément cette solution présente l'avantage de permettre de les mélanger dans les calculs. Il reste cependant certains garde-fous. D'une part, l'utilisation des sous-types nous garantit que l'exception `Constraint_Error` sera levée si jamais une valeur trop grande (ou négative) était passée à l'une des coordonnées. D'autre part, l'utilisation de la notation nommée rend beaucoup plus improbable un mélange de coordonnées. On voit mal un programmeur écrire (à moins de le faire vraiment exprès) :

```
Aller_en (X => Y_milieu, Y => X_milieu);
```

Notons une fois de plus l'importance du choix de bons identificateurs ! Cette sécurité disparaîtrait si nous appelions nos paramètres formels `A` et `B` au lieu de `X` et `Y`.

La règle la plus simple à appliquer est qu'il faut utiliser des types différents pour représenter des entités *réellement différentes*. Une prolifération de conversions doit amener à se demander s'il ne serait pas judicieux de fusionner certains types.

### *21.2 .1 Types discrets*

Le terme «type discret» recouvre les types énumératifs et les types entiers, c'est-à-dire les types que l'on utilise chaque fois que l'on souhaite représenter un élément du monde réel caractérisé par un nombre fini de valeurs distinctes.

Si l'on souhaite simplement représenter différentes valeurs, les types énumératifs s'imposent : ils sont plus lisibles (leur nom indique à quoi ils correspondent) et plus sûrs, notamment grâce au fait que les instructions **`case`** vérifient toujours que toutes les valeurs possibles ont été mentionnées : en cas de rajout d'une valeur au type suite à une modification du logiciel, le compilateur vous préviendra gentiment de tous les endroits où vous avez oublié d'ajuster les **`case`** correspondants. Enfin, le fait de devoir énumérer toutes les valeurs oblige à réfléchir dès le départ à l'ensemble des valeurs possibles.

> Ada dispose d'attributs permettant de transformer facilement des énumératifs en chaînes de caractères, ainsi que de sous-programmes d'entrées-sorties qui leur sont applicables. Ceci les rend beaucoup plus utilisables qu'en Pascal.

On réservera les types entiers au cas où l'on souhaite vraiment une sémantique de *nombre*, c'est-à-dire lorsque la variable est utilisée pour *compter* ou lorsque l'on a besoin d'effectuer des opérations arithmétiques. Et même alors, on évitera **ABSOLUMENT** l'utilisation du type `Integer`, sauf dans quelques cas que nous mentionnerons par la suite. Pourquoi l'avoir écrit si gros ? Parce qu'à l'usage, il s'avère que c'est une des habitudes les plus difficiles à faire perdre aux nouveaux programmeurs Ada.

> Attention : le type `Integer` peut parfois réapparaître de façon insidieuse. Si par exemple l'on écrit :
>
> ```
> for I in 1..10 loop ...
> ```
>
> le compilateur ne dispose d'aucune information sur le type de `I` et choisit donc `Integer`, faute de mieux. Il est toujours possible (et préférable) de donner explicitement le type :
>
> ```
> for I in Age range 1..10 loop ...
> ```
>
> Ceci s'applique également aux indices des types tableau. Pour une fois qu'Ada avait tenté d'économiser quelques touches à taper au programmeur, l'inconvénient se révèle supérieur au gain...

Le choix d'un type de données s'effectue en étudiant le domaine de problème ; c'est vrai des types entiers comme des autres. En particulier, on doit toujours penser qu'un type entier est limité par la machine et que l'utilisation d'un type prédéfini ne dispense pas d'étudier le problème des bornes ; ceci s'applique à *tous les langages*, mais seul Ada fournit le moyen de choisir les types de façon indépendante de la machine. On distingue plusieurs formes caractéristiques de déclarations de types en fonction des contraintes :

> Le modèle impose des contraintes absolues. Celles-ci sont directement reproduites dans la déclaration :
>
> ```
> type Jour_de_l_Année is range 1..366;
> ```
>
> Le modèle n'impose que des contraintes minimales ; le programmeur souhaite une implémentation matérielle efficace couvrant au moins les limites données :
>
> ```
> type Longueur_minimale is range 0..200;
> subtype Longueur_Utile is Longueur_minimale'Base;
> ```
>
> > Le type de base (obtenu par l'attribut `'Base`) est le type machine sous-jacent choisi par le compilateur. Il est donc plus vaste que le type d'origine. Cette utilisation de l'attribut `'Base` est nouvelle en Ada 95.
>
> Le programmeur souhaite imposer des limites liées au matériel, par exemple pour des interfaçages. On définira le type en conséquence et on l'assortira d'une clause de représentation :
>
> ```
> type INT_16 is range -2**15..2**15-1;
> for INT_16'Size use 16; -- Représenter le type sur 16 bits
> ```

En Ada 95, le paquetage `Interfaces` contient en standard la définition de tous les types machines supportés par le compilateur, sous la forme `Integer_8`, `Unsigned_16`, etc. Le programmeur n'a donc plus à les récrire.

Le programmeur a besoin d'un type entier, non lié à une sémantique particulière, sans contrainte d'efficacité... En fait, si c'était possible, il lui faudrait l'ensemble (infini) des entiers relatifs, au sens mathématique. Il se contentera du plus grand type possible :

```
type Grand_Entier is range System.Min_Int..System.Max_Int;
```

Le paquetage `System` est présent sur toutes les implémentations et contient des déclarations de constantes permettant de connaître les caractéristiques de la machine sur laquelle on s'exécute. Par exemple, `Min_Int` et `Max_Int` sont la plus petite et la plus grande valeur entière supportées par l'implémentation.

De plus, on a le choix entre les types signés, comme dans les déclarations ci-dessus et les types modulaires. On préférera les premiers en général, car ils offrent plus de sécurité (notamment en cas de débordement). On utilisera les types modulaires lorsque l'on a effectivement besoin d'une arithmétique circulaire, ou pour un type de très bas niveau, car ils permettent des manipulations supplémentaires au niveau du bit. Enfin il existe quelques cas où l'on *doit* utiliser `Integer` :

- Pour indexer le type `String`. Ce type a, par définition, `Integer` (ou plutôt son sous-type `Positive`) comme type d'index. Toutes les variables servant à manipuler les chaînes seront donc de type `Integer`.
- Lorsque le besoin est celui d'un type de taille raisonnable, adapté aux possibilités de la machine ; en particulier comme indice de tableau ou comme limite lorsque l'on n'a rien de mieux sous la main. `Integer` est censé être le type entier «naturel» de la machine et il doit permettre une indexation efficace. On préférera l'utiliser sous forme de type dérivé, pour éviter les mélanges :

```
type Index is new Integer;
type Liste is array (Index range <>) of Angle;
```

## *21.2 .2 Types réels*

De même que l'on n'utilise pas directement le type `Integer` en Ada, on ne saurait utiliser `Float` pour tous les calculs «réels». Ada offre toute une palette de types numériques et il faut choisir le plus approprié au problème à résoudre. Voici quelques grandes lignes pour orienter ce choix :

Décider d'abord si les grandeurs se représentent mieux sur une échelle linéaire (types point fixe) ou logarithmique (types point flottant). Noter que pour un même nombre de bits, la précision est meilleure avec les points flottants au voisinage de 0, mais qu'au contraire elle est meilleure pour les grandes valeurs avec les points fixes . En particulier, toute représentation du temps (pour lequel la notion de zéro est arbitraire) se définit normalement avec des points fixes[1].

Dans le cas d'une échelle linéaire, décider si l'on a un modèle de calculs approchés, ou si le besoin est celui d'une arithmétique exacte (comme avec des entiers), mais avec un pas différent de 1. Dans le premier cas, on utilisera des points fixes normaux :

```
type Volts is delta 0.01 range 0.0 .. 380.0;
```

Dans le second, si le pas souhaité est une puissance de 10, on utilisera des décimaux :

```
type Francs is delta 0.01 digits 10;
```

Autrement, on mettra une clause de représentation pour garantir le bon pas :

[1] C'est pour n'avoir pas suivi ce conseil que le programme (écrit en C !) qui contrôlait le départ des missiles Patriot pendant la guerre du Golfe a raté un Skud qui causa la mort de plus d'une dizaine de personnes.

```
type Position is delta 1.0/50.0 range 0.0 .. 1.0;
for Position'Small use 1.0/50.0;
```

- Dans le cas d'une échelle logarithmique, on peut souhaiter une précision qui soit fonction des capacités de la machine ou au contraire une précision indépendante de la machine. Par exemple, si l'on fait un programme graphique, la précision des calculs n'a pas grande importance et l'on souhaite utiliser les flottants normaux de la machine. Bien sûr, on préservera l'abstraction en n'utilisant pas directement le type `Float`, mais un type dérivé :

```
type Coordonnée is new Float;
type Coordonnée_précise is new Long_Float;
```

Noter qu'il est alors possible de connaître (de façon portable) la précision *a posteriori* des calculs grâce aux attributs tels que `Coordonnée'Digits`.

Dans d'autres cas, la précision minimale est obligatoire. Si l'on calcule par exemple l'angle de rentrée d'une navette dans l'atmosphère, la précision demandée est vitale : trop à plat, la navette rebondira sur les couches hautes, trop inclinée elle brûlera. Une analyse fine peut conduire à la conclusion que pour obtenir une précision de $10^{-3}$ sur le résultat, les données doivent être représentées avec une précision de $10^{-7}$. On déclarera donc :

```
type Donnée is digits 7;
```

Noter que selon la représentation interne de la machine, cette précision peut tenir ou non sur 32 bits ; le compilateur choisira donc une représentation interne en simple ou double précision, selon la machine. On aura peut-être plus de précision (ce qui ne sert à rien puisqu'on pilote des vérins dont la précision est limitée), mais la précision minimale est garantie.

On voit qu'Ada offre toute une palette de possibilités dans le choix des types numériques ; on évitera de se précipiter sur le type `Float` comme on le ferait dans d'autres langages.

## *21.2 .3 Chaînes de caractères*

La notion de chaîne de caractères est moins évidente qu'il n'y paraît. Là encore, une analyse fine des besoins amène à reconnaître plusieurs modes d'utilisation des chaînes de caractères, avec des contraintes différentes. C'est pourquoi Ada offre plusieurs modèles pour répondre aux différentes utilisations ; noter que les différents paquetages correspondant à ces possibilités offrent des sous-programmes de manipulation identiques (grâce à la surcharge) et que l'utilisateur n'a donc à apprendre qu'un seul mode d'emploi. Voici ces différents modèles.

- Les messages. Il s'agit de chaînes destinées à dialoguer avec l'utilisateur et qui ne sont donc pas des représentations d'entités de plus haut niveau. Le type prédéfini `String` est parfaitement approprié. Le paquetage `Ada.Strings.Fixed` fournit toutes les fonctionnalités habituelles : déplacement, recopie, transcodage, recherche, etc.
- Des types de données abstraits, que l'on choisit de *représenter* au moyen de chaînes de caractères. Si les propriétés du type abstrait conduisent à la conclusion que toutes les chaînes ont la même longueur, on utilise simplement un sous-type de `String`. Si au contraire on a besoin de chaînes de longueur variable, on instancie le paquetage `Generic_Bounded_Length` (qui est fourni dans le paquetage `Ada.Strings.Bounded`). Ce paquetage fournit un type de chaîne de caractère de longueur variable, dont la taille maximale est donnée à l'instanciation. Bien entendu, la taille maximale choisie doit résulter d'une analyse des propriétés du type de donnée abstrait. Par exemple, on peut choisir de représenter le nom d'une personne par une chaîne variable et décider que la longueur maximale sera de 15 caractères :

```
Max_Nom : constant := 15;
package Opérations_Nom is new
   Ada.Strings.Bounded.Generic_Bounded_Length(Max_Nom);
subtype Nom is Opérations_Nom.Bounded_String;
```

Nous avons fourni le sous-type `Nom` pour exporter un identificateur significatif pour l'utilisateur. Ceci ne rend pas les opérations définies dans le paquetage directement visibles ; mais ces opérations ne sont utiles qu'aux personnes effectuant des recherches, modifications, etc., sur des noms. En appelant l'instanciation `Operations_Nom`, cela permet à ces utilisateurs de marquer les endroits où ils travaillent effectivement sur des noms grâce à une clause **`use`** `Opérations_Nom`.

- Le stockage de caractères. Le besoin n'est pas à proprement parler celui d'une «chaîne», mais plutôt d'un tableau de caractères que le programme manipule. En général, la taille de ce tableau est virtuellement infinie ; un exemple typique est celui d'un programme de traitement de texte qui stocke tout le document dans un seul grand tableau en mémoire. On utilise alors le type `Unbounded_String` (ou plutôt un type dérivé), du paquetage `Ada.Strings.Unbounded` :

  ```
  type Document is new Unbounded_String;
  ```

  Attention : la limite des `Unbounded_String` (il en faut bien une) est donnée par `Natural'Last`, c'est-à-dire 2_147_483_647 pour une implémentation de `Integer` sur 32 bits (ce qui devrait suffire amplement), mais seulement 32_767 pour une implémentation 16 bits, ce qui peut s'avérer insuffisant. On risque donc d'avoir une non-portabilité à ce niveau.

Enfin, il se peut que d'autres besoins conduisent à la définition d'un paquetage «maison» de chaînes de caractères, avec des caractéristiques différentes. Au nom de l'uniformité et de la diminution de la complexité, on veillera à fournir les mêmes fonctionnalités, avec les mêmes noms, que dans les paquetages faisant partie de l'environnement standard.

## *21.2 .4 Types articles*

On utilisera des types articles chaque fois qu'une entité est composée de plusieurs autres éléments. Le principal choix (non évident) est de décider si l'on doit utiliser un type étiqueté ou un type article à discriminants lorsque l'on veut rassembler dans un même type (ou famille de types) des valeurs *polymorphes*, c'est-à-dire ne comportant pas toutes les mêmes éléments.

Dans un type à discriminant, les différentes variantes sont définies statiquement lors de la déclaration initiale du type ; l'ajout par la suite de structures non prévues au départ nécessite des changements importants dans l'ensemble du logiciel. En revanche, ce même aspect statique garantit un maximum de contrôle. Il n'est pas possible d'effectuer des modifications qui remettraient en cause la validité d'une partie de logiciel déjà validée. Enfin, il est possible (si le type l'a autorisé) de déclarer des variables *non contraintes*, qui peuvent donc contenir différentes variantes au cours du temps.

Les type étiquetés offrent un maximum de souplesse ; ils peuvent être enrichis après coup sans toucher au type d'origine. En revanche, ces modifications peuvent affecter (*via* le mécanisme de liaison dynamique) le comportement des modules déjà écrits. La liaison dynamique, de par son principe même, peut empêcher toute certification exhaustive des modules. Enfin, les objets se voient affecter un type effectif au moment de leur élaboration et ne peuvent en changer par la suite.

Conceptuellement, on préférera les types à discriminants si les différentes variantes constituent de simples paramètres, ou des options, d'un seul type. En particulier, si ces options sont susceptibles d'évoluer au cours du temps, il faudra utiliser des variables non contraintes qui ne sont possibles qu'avec les types à discriminants. On préférera les types étiquetés si l'on pense avoir affaire à des types *différents*, mais appartenant à une *même famille* dont on veut exprimer dans le langage la nature commune. Ces principes peuvent être pondérés par des considérations pragmatiques : là où la validation et la sécurité sont prédominants, les types à discriminants sont préférables. Inversement,

pour des logiciels à moindres contraintes sécuritaires, mais dont on prévoit une évolution plus importante dans le temps, les types étiquetés apportent plus de souplesse.

Enfin, ces deux options ne sont pas exclusives : il est possible d'avoir des types étiquetés à discriminants. Il sera rare de les utiliser pour des parties variables, mais il est tout à fait raisonnable de définir des types étiquetés dont certains éléments, des tables par exemple, ont une taille paramétrable au moyen de discriminants.

## *21.2 .5 Pointeurs*

Les pointeurs sont un outil indispensable dans certains cas, mais l'expérience montre qu'ils sont une cause importante de difficultés. Celles-ci peuvent être causées d'une part par le fait de travailler sur des références, plus difficiles à appréhender par le programmeur que les variables directes (le programmeur doit gérer dans sa tête le niveau d'indirection supplémentaire) ; d'autre part, le mécanisme d'allocation et surtout de désallocation dynamique est difficile à gérer : démontrer l'absence de «fuites de mémoire» (variables allouées et jamais désallouées) est techniquement si ardu que de nombreux systèmes temps réel devant fonctionner en permanence préfèrent interdire l'utilisation de toute allocation dynamique. L'expérience nous a amené à formuler la «loi» suivante :

> *La probabilité de «bugs» dans un logiciel est directement proportionnelle à la densité des pointeurs.*

On utilise moins les pointeurs en Ada que dans d'autres langages, car on dispose d'autres moyens pour réaliser certaines structures. Notons en particulier que :

- Il est possible d'avoir des structures de données, notamment des tableaux, dont la taille n'est connue qu'à l'exécution sans utiliser pour autant l'allocation dynamique.
  Le mécanisme de la programmation orientée objet n'est pas lié à la notion de pointeur.
- Les génériques offrent une alternative à l'utilisation de pointeurs pour fournir à un sous-programme ou à un paquetage des références à des éléments qui leur sont extérieurs (sous-programmes, variables) .

Inversement, un cas exige des pointeurs et de l'allocation dynamique : la construction de structures de données liées et dynamiques : arbres, listes, etc., et plus généralement lorsque l'on souhaite définir un type de donnée abstrait à sémantique de référence (rappelons cependant que ce n'est pas la seule solution possible).

On utilise aussi parfois des pointeurs pour éviter de copier des données volumineuses : imaginons une table de symboles par exemple. Il serait très pénalisant de renvoyer toute l'information associée à un symbole, surtout si l'utilisateur n'a besoin que d'une information spécifique. Il est donc préférable de renvoyer une référence sur l'entrée dans la table. Dans d'autres langages, ceci a une conséquence fâcheuse : l'utilisateur peut modifier le contenu de la table en utilisant directement la référence. C'est pourquoi Ada fournit des pointeurs sur constantes : de tels pointeurs peuvent désigner aussi bien des variables que des constantes, mais ne peuvent être utilisés pour modifier l'objet désigné. Une table des symboles pourrait ainsi s'écrire :

```
package Table_des_Symboles is
   type Information is
      record
         Info_1 : ...;
         Info_2 : ...;
         etc...
      end record;

   type P_Information is access constant Information;

   type Clé is new String;
```

```
   procedure Ajouter (Nom : Clé; Info : Information);
   function L_Information (De : Clé) return P_Information;
end Table_des_Symboles;
```

Côté utilisateur on aura :

```
Ajouter ("Rosen", ....);
   ...
if L_Information (De => "Rosen").Info_1 = ...  --OK
   ...
L_Information (De => "Rosen").Info_1 := ...; -- ERREUR !
```

On peut donc garantir l'intégrité de la table, tout en renvoyant des références sur les structures internes. Notons que certaines formes apparentées à des pointeurs (paramètres accès, autopointeurs, attribut 'Access) permettent d'établir des références et de créer des entités logiquement reliées sans nécessiter d'allocation dynamique ; les règles du langage garantissent l'impossibilité de conserver des pointeurs sur des entités ayant disparu.

On limitera donc l'utilisation des pointeurs et plus spécialement de l'allocation dynamique, aux seuls cas où elle est réellement indispensable. On fera usage si possible des autres possibilités et on prendra garde de ne pas se laisser entraîner à mettre des pointeurs partout par mimétisme avec des langages ne disposant pas de structures de données aussi puissantes que celles d'Ada.

## *21.2 .6 Utilisation de Unchecked_Conversion*

Ayant abondamment vanté les mérites du typage fort, on peut se demander pourquoi Ada autorise le détypage au moyen de Unchecked_Conversion. En fait, il s'agit d'une fonctionnalité *indispensable* lors de changements de niveaux d'abstraction, qui imposent de voir la même donnée selon deux vues différentes. Prenons un exemple.

Un nombre flottant est considéré, dans la plupart des applications, comme une entité autonome. Cependant, certains algorithmes numériques fins nécessitent d'accéder séparément à la mantisse ou à l'exposant du nombre. Il convient donc de fournir deux vues du même type : soit comme un tout, soit comme une entité composite comprenant signe, mantisse et exposant[1]. Nous pouvons fournir ce service de la façon suivante :

```
type Intervalle_Mantisse is range 0 .. 2**24;
type Intervalle_Exposant is range -64..63;

type Flottant_Composite is
   record
      Négatif  : Boolean;
      Exposant : Intervalle_Exposant;
      Mantisse : Intervalle_Mantisse;
   end record;

-- Clauses de représentation
for Flottant_Composite'ALIGNMENT use 4;
for Flottant_Composite use
   record
      Négatif  at 0 range 0..0;
      Exposant at 0 range 1..7;
      Mantisse at 1 range 0..23;
   end record;

function Vers_composite is new
   Unchecked_Conversion (Float, Flottant_composite);
```

[1] Les numériciens ont tellement besoin de ces fonctionnalités de manière portable (ce qui n'est pas le cas de notre solution) qu'Ada 95 offre de nouveaux attributs fournissant directement ces services.

```
function Exposant (Nombre : Float)
   return Intervalle_Exposant is
begin
   return Vers_composite (Nombre).Exposant;
end Exposant;
```

Remarquez le processus : on décrit la vue que l'on veut obtenir (le type article) et sa représentation interne (les clauses de représentation). Ensuite, `Unchecked_Conversion` nous permet de voir la même donnée selon l'une ou l'autre vue. A partir de là, la fonction `Exposant` n'a plus qu'à retourner la partie exposant d'un flottant considéré comme un article. Noter au passage l'aspect descriptif de cette façon de faire : c'est le compilateur qui devra se débrouiller pour gérer les masques et les décalages nécessaires à extraire l'information... ce qui n'est pas un petit service rendu au programmeur qui n'a plus à se soucier de cette «cuisine».

Notons enfin qu'il existait un cas fréquent d'utilisation de `Unchecked_Conversion` qui a disparu avec Ada 95 : celui où l'on devait voir des données de haut niveau comme des suites d'octets, pour stockage sur disque ou envoi sur un réseau par exemple. Les nouveaux attributs `'Read` et `'Write` fournissent désormais cette conversion de vue ; citons également l'attribut `'Valid` qui permet de vérifier qu'une valeur en provenance de l'extérieur (entrée-sortie, autre langage) est bien conforme aux exigences du typage Ada, ainsi que le paquetage `Ada.Storage_IO` qui effectue des pseudo-entrées-sorties vers des tableaux d'octets.

### *21.2 .7 Conclusion sur l'utilisation des types*

Choisir le bon type pour représenter une entité n'est pas chose aisée : entre le désir de modéliser de façon précise les entités du monde réel, la nécessité d'éviter une prolifération abusive des types, les contraintes d'évolutivité et d'efficacité, les solutions sont nombreuses et il faut rechercher le meilleur compromis. Il est donc important de *prendre le temps* de réfléchir soigneusement avant de définir un type. Eventuellement, l'utilisation du paquetage `ADPT` et des types associés permet de poursuivre le projet en différant la décision jusqu'à un moment où le problème et les contraintes éventuelles seront mieux connus. Il ne faudrait cependant pas en déduire qu'Ada crée de nouvelles difficultés ; bien sûr, dans les langages où le seul type entier est `Integer`, le problème du choix d'un type entier ne se pose pas ! Il n'empêche que le problème de la définition du type le plus approprié existe toujours. Simplement, ces autres langages n'offrent aucun outil pour le résoudre et il reste entièrement à la charge du programmeur.

## 21.3 Choix liés aux génériques

Faut-il ou non utiliser des génériques ? Encore un point où l'on se pose des questions avec Ada que l'on ne se posait pas avant... puisque la fonctionnalité n'existait pas. Et pourtant, les génériques sont bien utiles, puisqu'ils ont été adoptés (sous des formes plus ou moins différentes) par des langages comme Eiffel et C++. Les points à considérer sont d'ordre économique, méthodologique et de performance.

### *21.3 .1 Point de vue économique*

Toutes choses égales d'ailleurs, le développement d'un générique prend plus de temps et d'effort que son équivalent non générique... tant qu'il n'y en a qu'un. Ce surcoût est rapidement amorti dès que le générique est utilisé plusieurs fois. Inversement, il est moins coûteux d'écrire directement un générique que de transformer par la suite une unité qui n'a pas été prévue pour[1]. Voici quelques règles empiriques pour guider ce choix :

[1] Sauf dans le cas d'une première version développée sous forme non générique pour faciliter la mise au point, mais

Un module dont on n'a aucune raison de penser qu'il puisse se généraliser n'est pas mis sous forme de générique.
Un module qui a toutes les chances d'être généralisé est écrit directement sous forme de générique.
Un module dont on ne pense pas qu'il puisse être généralisé, au moins à court terme, est écrit sous forme non générique, mais on prendra des précautions pour faciliter un éventuel passage ultérieur en générique.

- Il vaut mieux adapter un module déjà développé en le passant en générique que de récrire entièrement une entité voisine, surtout si l'on est tenté de faire des duplications à partir de l'existant.

## *21.3 .2 Point de vue méthodologique*

### a) Héritage et généricité

Parler des interfaces

On a souvent besoin d'écrire des algorithmes applicables à plusieurs types de données, ce qui peut être obtenu par des génériques ou par des techniques d'héritage. Tous les ouvrages portant sur les langages à objets (notamment [Mey90]) comportent un chapitre comparant les avantages et inconvénients respectifs de ces deux solutions. Ils sous-estiment généralement les possibilités de la généricité[1] et surtout ne la considèrent que comme un succédané d'héritage dans les langages d'où celui-ci est absent. En fait, généricité et héritage relèvent chacun de sa propre démarche et il n'y a aucune raison de considérer l'un comme «supérieur» à l'autre.

Reprenons l'exemple qui nous a permis d'introduire la notion de liaison dynamique dans la deuxième partie : le déplacement d'un objet graphique. En termes de généricité, ce problème s'analyserait de la façon suivante :

> *Pour tout objet muni de procédures d'effacement, de dessin et de la notion de point caractéristique dont on peut mettre à jour la position, on peut écrire l'algorithme de déplacement.*

Cette analyse conduit à la définition de la procédure générique suivante :

```
generic
   type Objet       is limited private;
   type Coordonnées is private;
   with procedure Dessiner (Quoi : Objet);
   with procedure Effacer  (Quoi : Objet);
   with procedure Mise_A_Jour (Quoi : Objet;
                               X, Y : Coordonnées);
procedure Déplacer (Quoi : Objet; X, Y : Coordonnées);

procedure Déplacer (Quoi : Objet; X, Y : Coordonnées) is
begin
   Effacer (Quoi);
   Mise_A_Jour (Quoi, X, Y);
   Dessiner (Quoi);
end Déplacer;
```

Si la formulation algorithmique est sensiblement équivalente à celle obtenue dans le cas de l'héritage, sa philosophie est différente. Il s'agit ici d'une abstraction algorithmique *extérieure* aux

dont la structure a été pensée dès le départ pour devenir générique par la suite.

[1] En ne considérant que les possibilités de paramétrisation par les types et en omettant la possibilité d'importer les opérations associées, ce qui réduit considérablement la puissance des génériques.

objets manipulés : `Déplacer` est applicable à tout type ayant les propriétés minimales requises, sans pour autant que ceux-ci aient quoi que ce soit en commun. Par exemple, on peut imaginer une grue (jouet) télécommandée, capable de poser et de ramasser des cubes. On peut raisonnablement assimiler le fait de poser un cube à `Dessiner` (le cube «apparaît» sur le sol), ainsi que le fait de prendre un cube à `Effacer`, `Mise_A_Jour` déplaçant physiquement la grue sur le sol. Dans ce contexte, notre algorithme peut être utilisé pour déplacer un cube d'un point à un autre. Une démarche par héritage demanderait que l'objet `Grue` hérite de la classe des objets graphiques, ce qui signifierait que nous considérons qu'une grue *est* une forme d'objet graphique. Bien sûr cela fonctionnerait du point de vue informatique, mais ce serait tout de même conceptuellement gênant...

On ne peut réutiliser un algorithme que s'il existe une certaine parenté entre les utilisateurs ; mais le changement de technique conduit à une *inversion de la hiérarchie des valeurs*. Avec la généricité, on définit un type de donnée abstrait dans un paquetage et on lui rajoute des fonctionnalités par instanciation de générique : c'est donc le code réutilisé qui vient enrichir l'abstraction de l'utilisateur. En revanche, pour récupérer un algorithme appartenant à une classe par héritage, il faut en hériter, c'est-à-dire incorporer son propre type de données à la famille de la classe dont on hérite. Pour prendre une image, si on instancie un générique pour rajouter des fonctionnalités à un type, on soumet l'algorithme instancié aux besoins du type à concevoir ; si en revanche, on fait hériter le type d'une classe, on soumet le type à concevoir à la classe dont on hérite.

Si dans les deux cas il y a dépendance du réutilisateur vers le réutilisé, la dépendance est moins forte avec la généricité, car il n'y a pas de relations entre deux types qui instancient un même générique, alors qu'il y en a nécessairement une, au moins conceptuelle, entre deux types qui héritent d'une même classe. Nous considérerons donc que la généricité induit un couplage moins fort que l'héritage.

En résumé, on peut dire que la solution par héritage permet une meilleure *concentration* des propriétés, mais que la généricité fournit une meilleure *indépendance*. On utilisera la généricité pour exprimer un algorithme applicable à différents types de données vérifiant un certain nombre de propriétés minimales, sans qu'il y ait nécessairement de lien logique entre ces types ; inversement, on utilisera l'héritage si l'algorithme n'est applicable qu'à une famille de types liés par une dépendance conceptuelle.

### b) Généricité et pointeurs sur sous-programmes

Un autre cas où l'on doit se poser la question de l'utilisation ou non de génériques est celui d'algorithmes paramétrables par un sous-programme. Si nous voulons calculer l'intégrale d'une fonction, nous pouvons écrire :

```
generic
   with La_Fonction (X: Float) return Float;
procedure Intégrer (Début, Fin : Float);
....
procedure Intégrer_Sinus is new Intégrer (Sin);
...
Intégrer_Sinus(0.0, 1.0);
```

Nous pouvons aussi passer un pointeur sur fonction :

```
type Fonction_Math is access function (X: Float) return Float;
procedure Intégrer (La_Fonction : Fonction_Math;
                    Début, Fin  : Float);
...
Intégrer (Sin'Access, 0.0, 1.0);
```

La seconde solution est la seule permise par d'autres langages ne disposant pas de la généricité ; elle paraît donc plus naturelle aux nouveaux venus à Ada. Inversement, la première solution était la seule permise par Ada 83 et sera préférée par les vétérans du langage. La principale différence (en dehors de quelques considérations mineures d'efficacité) provient des règles de portée. Comme toute déclaration, une instanciation de générique est utilisable depuis sa déclaration jusqu'à la fin de

sa portée : il n'y a aucune mauvaise surprise à craindre, on peut instancier le générique avec n'importe quelle fonction et toute mauvaise utilisation sera détectée à la compilation. En revanche, la seconde solution n'autorise de passer à la procédure que des fonctions dont la durée de vie soit *supérieure* à celle de la procédure elle-même, afin d'interdire toute référence à une fonction qui n'existerait plus. Dans certains cas, ce test doit être effectué à l'exécution (avec levée éventuelle de l'exception `Program_Error`). Ceci limite quasiment les fonctions que l'on peut désigner par un pointeur à celles déclarées au niveau le plus externe[1] ; on préférera donc souvent la solution générique. Il est cependant un cas qui ne peut être traité que par pointeur sur sous-programme : celui où l'on souhaite stocker l'identité du sous-programme pour l'appeler par la suite, cas fréquent lorsque l'on veut «enregistrer» un traitement pour l'appeler par la suite lors de la survenue d'un événement (mécanisme dit du *call back,* fréquent dans les systèmes de fenêtrage) ; dans ce cas l'utilisation du pointeur sur sous-programme est impérative.

## *21.3 .3 Point de vue des performances*

On a souvent peur de la duplication de code engendrée par l'utilisation des génériques ; mais si l'on écrit explicitement deux fois la même chose, il y a *certainement* duplication, alors que certains compilateurs sont capables de partager du code entre instanciations. Autrement dit, ce n'est pas l'utilisation du générique qui augmente la taille du code, c'est le fait d'avoir deux modules voisins et l'utilisation d'un générique ne peut être pire (mais peut être plus économique) que d'écrire deux fois le module[2]. De plus, le risque de duplication peut être considérablement réduit par une découpe judicieuse de la structure. Retenons une règle générale :

> *On ne doit mettre en générique que les parties qui dépendent spécifiquement des paramètres génériques.*

En particulier, si des sous-programmes sont nécessaires pour la réalisation de l'unité générique, mais qu'ils ne dépendent pas eux-mêmes des paramètres génériques, on aura intérêt à les regrouper dans des paquetages (non génériques) externes. Le générique se trouvera être en fait le point d'entrée d'un petit sous-système. Dans certains cas, on peut aboutir à une structure où la partie générique n'est plus qu'une «peau», généralement destinée à préserver le typage Ada. Supposons par exemple que nous voulions faire une interface de boîte aux lettres. Nous voulons bien sûr respecter le typage Ada, nous fournissons donc une interface :

```
generic
   type Type_Message is private;
package Boîte_Aux_Lettres is
   procedure Emettre  (Message : in  Type_Message);
   procedure Recevoir (Message : out Type_Message);
end Boîte_Aux_Lettres;
```

En fait le traitement est totalement indépendant du type de données, puisque pour les besoins de la boîte aux lettres, l'élément à envoyer n'est qu'une suite d'octets... Nous définissons donc le paquetage (non générique) suivant, qui fera l'essentiel du travail :

```
with System.Storage_Elements;
package Messagerie_Bas_Niveau is
   use System.Storage_Elements;
   procedure Emettre    (Depuis : in  Storage_Array);
   procedure Recevoir   (Dans   : out Storage_Array);
end Messagerie_Bas_Niveau;
```

---

[1] Les autres langages ne font pas mieux ; simplement, comme les fonctions C sont *toujours* définies au niveau le plus externe (il n'y a pas d'imbrication), il n'est pas besoin de préciser cette règle...

[2] Se méfier en particulier de la notion de «macros» présente dans d'autres langages, qui conduit à des duplications de code que l'on a tendance à oublier.

Le corps générique va utiliser le paquetage `Ada.Storage_IO` pour convertir le type de haut niveau en une suite d'octets, puis appeler les fonctions de `Messagerie_Bas_Niveau` :

```
with  Messagerie_Bas_Niveau, Ada.Storage_IO;
package body Boîte_Aux_Lettres is
   package Message_IO is new Ada.Storage_IO(Type_Message);
   use  Messagerie_Bas_Niveau, Message_IO;
   Tampon : Buffer_Type;
   procedure Emettre  (Message : in  Type_Message) is
   begin
      Write (Tampon, Message);
      Emettre (Tampon);
   end Emettre;

   procedure Recevoir (Message : out Type_Message) is
   begin
      Recevoir (Tampon);
      Read (Tampon, Message);
   end Recevoir;
end Boîte_Aux_Lettres;
```

Pourquoi ne pas mettre directement `Messagerie_Bas_Niveau` à la disposition de l'utilisateur ? Précisément parce qu'à ce niveau, toute vérification de type a disparu. Si c'est nécessaire pour l'implémentation, il est toujours préférable de le cacher à l'utilisateur.

Noter qu'en Ada 83, il n'existait pas de moyen de cacher *physiquement* le paquetage `Messagerie_Bas_Niveau` et que l'on n'avait donc pas de garantie que l'utilisateur n'y avait pas accès, à moins de tout rassembler en un seul paquetage. En Ada 95, le mécanisme de bibliothèques hiérarchiques nous permet ce contrôle tout en bénéficiant de compilations séparées. Une meilleure organisation est donc la suivante :

```
package Boîtes_Aux_Lettres is
end Boîtes_Aux_Lettres;

generic
   type Type_Message is private;
package Boîtes_Aux_Lettres.Une_Boite is
   procedure Emettre  (Message : in  Type_Message);
   procedure Recevoir (Message : out Type_Message);
end Boîtes_Aux_Lettres.Une_Boite;

with System.Storage_Elements;
private package Boîtes_Aux_Lettres.Messagerie_Bas_Niveau is
   use System.Storage_Elements;
   procedure Emettre    (Depuis : in  Storage_Array);
   procedure Recevoir   (Dans   : out Storage_Array);
end Boîtes_Aux_Lettres.Messagerie_Bas_Niveau;
```

Le paquetage `Boîtes_Aux_Lettres` ne sert plus que d'enveloppe ; `Une_Boîte` est un enfant public, utilisable par tout le monde, mais `Messagerie_Bas_Niveau` est un enfant privé, appelable uniquement depuis `Une_Boîte`. Les propriétés de fermeture du sous-système sont donc garanties. Le paquetage général `Boîtes_Aux_Lettres` est quand même nécessaire, car un enfant de générique ne peut être que générique ; `Messagerie_Bas_Niveau` ne pourrait donc être un enfant (même privé) de `Une_Boîte`.

En termes de vitesse d'exécution, l'utilisation de génériques ne devrait[1] pas provoquer de pénalité, sauf dans le cas de génériques partagés ; mais c'est alors un effet de la règle habituelle que les gains en espace mémoire se paient en temps d'exécution. Certains compilateurs disposent d'ailleurs d'options permettant à l'utilisateur de choisir entre partage et duplication.

[1] Ce conditionnel est là pour rappeler au lecteur que *toute* considération d'efficacité doit faire l'objet d'une mesure.

## 21.4 Choix liés au parallélisme

Lorsque l'analyse du système conduit à la définition de processus parallèles, on retrouvera naturellement des tâches correspondantes au niveau de l'implémentation. Ceci dit, l'usage des tâches n'est pas limité à ces tâches structurelles : elles peuvent se révéler un excellent moyen de structuration. Inversement, l'enthousiasme né de la disponibilité de cet outil qui n'existe que dans peu d'autres langages peut conduire à des abus : il faut donc définir les conditions où l'usage de tâches peut être bénéfique.

### *21.4 .1 Principes généraux*

Le modèle Ada des tâches en fait fondamentalement des serveurs, fournissant par leurs entrées des services à leurs clients. En général, l'état normal d'un serveur sera l'état bloqué, en attente de demande de service ; lors de la réception d'une demande, le serveur deviendra actif, provoquant au besoin le réveil d'autres serveurs, puis au bout d'un certain temps l'ensemble tendra à retourner vers l'état bloqué. On peut donc voir l'ensemble des tâches comme une suite d'engrenages, qu'un mouvement de balancier vient mettre en mouvement, puis qui retourne à son état stable... jusqu'au prochain évènement.

De telles tâches sont généralement peu coûteuses pour le système : une tâche à l'état bloqué ne prend aucune puissance de calcul. En revanche, elle occupe un certain espace mémoire, au moins l'espace de sa pile.

> L'utilisateur peut spécifier au moyen du pragma `Storage_Size` l'espace mémoire d'une tâche. Ce pragma, comme l'attribut d'Ada 83 qu'il remplace, peut être dynamique : on peut donc ajuster à l'exécution l'espace des tâches.

Inversement, les tâches actives, c'est-à-dire celles qui «tournent» sans se bloquer, consomment de la puissance de calcul inutilement. On en a besoin par exemple pour faire des scrutations (*polling*) de périphériques qui ne disposent pas de possibilités d'interruptions : le seul moyen de savoir qu'une donnée est arrivée est alors d'aller les interroger périodiquement. On veillera cependant à insérer un **`delay`** (même faible) dans la boucle pour éviter que la tâche n'accapare l'unité centrale sur un monoprocesseur.

On pourra enfin avoir une, et *au plus une*, tâche en boucle réellement continue ; ce sera par exemple la boucle principale d'un exécutif périodique. On peut en autoriser une dans un système temps réel, car il faut bien «occuper l'UC» quand le système ne traite aucun événement ; on ne peut en autoriser qu'une, car en l'absence de suspension (ou de partage de temps) sur un monoprocesseur, une autre tâche en boucle permanente ne prendrait jamais la main.

Les tâches constituent un outil très commode et, sauf contrainte particulière, il ne faut pas chercher à les éviter délibérément. Inversement, certains ont parfois tendance à en mettre partout. Les remarques que nous avons faites sur le minimum de complexité s'appliquent à la décomposition en tâches : un petit nombre de tâches fait décroître la complexité, un trop grand nombre fait perdre ce bénéfice à cause de la multiplication des communications. Il importe donc de choisir une granularité raisonnable, entre les deux extrêmes que sont la tâche unique qui fait tout et une tâche par service. En particulier, lorsque plusieurs services sont logiquement reliés, il est préférable de les faire traiter par une seule tâche, même si cela provoque de petites sérialisations.

### *21.4 .2 Tâches comme unités de structuration*

On peut être parfois amené à utiliser des tâches comme unités de structuration, indépendamment de tout aspect «parallélisme». En fait, il est assez naturel d'utiliser des tâches (au sens Ada) pour modéliser des «tâches à accomplir» qui n'ont pas de rapports dans le temps. Imaginons par exemple que nous devions remplir deux formulaires A et B. Ceux-ci comportent des calculs et il faut parfois

reporter des calculs intermédiaires d'une feuille sur l'autre pour pouvoir poursuivre (Fig. 36). Il est bien sûr possible d'écrire un programme séquentiel effectuant les calculs ; mais il faudra déterminer soigneusement dans quel ordre s'occuper par moments des calculs de A et par moments des calculs de B. Il est de plus impossible d'obtenir une structure «objet», avec des modules distincts pour représenter chacune des feuilles.

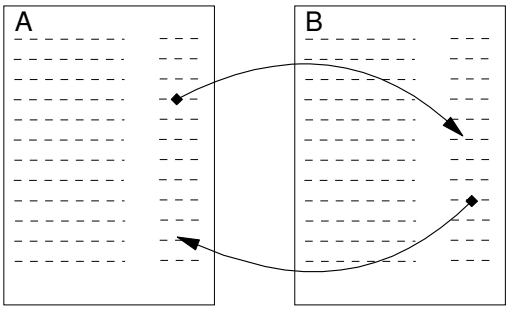


**Figure 36 :** Feuilles de calcul couplées

C'est possible en revanche si l'on représente chacune des feuilles par une tâche qui effectue les calculs nécessaires, qui envoie à l'autre les résultats intermédiaires et qui se met en attente lorsqu'elle a besoin de données en provenance de l'autre feuille. Le programmeur n'a plus à se soucier de l'ordre exact des calculs : il résultera de l'interaction entre les tâches. On voit qu'ici, l'utilisation des tâches apporte des améliorations d'ordre structurel (un objet du programme par objet du monde réel) et fonctionnel (plus besoin de se préoccuper des dépendances temporelles). Le parallélisme ne joue aucun rôle.

## *21.4 .3 Synchronisations et communications : rendez-vous ou objets protégés ?*

En Ada 83, le rendez-vous était le seul moyen de synchronisation et de communication. Ada 95 a rajouté à celui-ci les objets protégés. Il ne faudrait surtout pas croire que ceux-ci remplacent les rendez-vous : ils comblent plutôt certains besoins.

Le rendez-vous est un mécanisme de haut niveau. Facile à comprendre, il modélise bien les interactions entre objets du monde réel. C'est cependant un mécanisme riche ; de nombreux besoins réclament un mécanisme plus fruste. Par exemple, il est facile d'utiliser une tâche et des rendez-vous pour contrôler l'accès à une variable partagée par plusieurs tâches. C'est cependant un mécanisme lourd par rapport à la nature du problème et qui peut conduire à une multiplication abusive des tâches s'il y a beaucoup de variables partagées. De même, pour réaliser par exemple des objets gardés, il est nécessaire de fournir des verrous de bas niveau, de type «sémaphores». Bien qu'il soit possible de les réaliser au moyen de tâches, cela conduit à des inversions d'abstractions : implémentation d'outils de bas niveau au moyen d'outils de plus haut niveau. Les objets protégés ont donc été rajoutés pour satisfaire ce besoin : fournir un moyen de synchronisation *de bas niveau*, très efficace, mais aux possibilités restreintes. Bien que syntaxiquement rien ne l'y oblige, il est clair que le corps d'une opération protégée ne doit pas faire plus en pratique que changer l'état de quelques variables.

Si les inversions d'abstraction sont fâcheuses, il ne faudrait pas pour autant tomber dans le travers inverse : tout réaliser au moyen d'outils élémentaires, alors que l'on dispose d'outils de plus haut niveau. On réservera donc l'emploi des objets protégés aux mécanismes simples, rapides et de bas niveau : protection de variable, barrière, sémaphore... Il s'agira en général d'abstractions d'objets *informatiques*. On utilisera plutôt les rendez-vous pour les communications de haut niveau entre tâches appartenant au *domaine de problème*.

## 21.5 Choix liés aux situations exceptionnelles

Au niveau de la politique de projet, nous avons décidé d'une politique de gestion des *erreurs*. Mais toutes les situations exceptionnelles ne sont pas nécessairement des erreurs. Il convient donc de décider individuellement, pour chaque sous-programme, quoi faire dans l'éventualité d'une situation exceptionnelle ne permettant pas de fournir le service demandé. Les différents points à considérer sont les suivants [Goo88] :

Y a-t-il des cas exceptionnels ? Comme nous l'avons vu à propos des composants logiciels, il est souvent possible (mais pas toujours souhaitable) de compléter la sémantique au lieu de lever une exception. Par exemple, si nous avons une fonction de recherche d'une sous-chaîne dans une chaîne, il est souvent plus simple, *même pour l'utilisateur*, de renvoyer une valeur conventionnelle[1] lorsque la chaîne recherchée n'est pas présente, plutôt que de lever une exception.

En cas de condition exceptionnelle, faut-il lever une exception, ou utiliser le mécanisme défini par le projet pour gérer les erreurs ? Tout se joue ici sur la distinction entre «erreur» et «exception», ainsi qu'entre composant réutilisable et module spécifique du projet.

Faut-il fournir une fonction d'interrogation pour savoir si l'exécution du sous-programme est possible ? Il faut pour cela qu'une telle fonction ait une utilité et que sa complexité soit faible devant celle de la fonctionnalité testée. Il est par exemple utile de fournir une fonction testant la fin de fichier avant lecture ; mais une fonction qui fournirait un prédicat pour savoir si une matrice est inversible serait peu utile et presque aussi complexe que l'inversion de matrice elle-même.

Faut-il fournir une fonction d'interrogation permettant, en cas d'anomalie, d'obtenir plus de détails sur la cause ? Une telle fonction peut être très utile, mais ne s'applique qu'aux anomalies «sophistiquées» ; de plus, il faut prendre des précautions pour assurer la réentrance en cas d'utilisation du parallélisme.

Ada 95 offre des outils permettant d'avoir plus d'information sur la cause d'une exception, comme d'attacher un message spécifique à la levée d'une exception. De plus, les *attributs de tâches* permettent de stocker des valeurs dans le contexte de chaque tâche et donc d'écrire des fonctions d'interrogations réentrantes.

## 21.6 Conclusion

Tout au long de cette quatrième partie, nous avons insisté sur l'importance de faire des choix rationnels à tous les niveaux du développement. Bien que ceci puisse paraître contraignant, nous terminerons en rappelant que cette démarche est avant tout *rentable*. Ne rien laisser au hasard conduit à des logiciels mieux construits, plus portables, plus évolutifs, plus sûrs et en un mot de meilleure qualité. Le génie logiciel est avant tout un état d'esprit ; il nécessite des méthodes, des outils de haut niveau – en particulier un langage adapté – mais la clé permettant le développement de logiciels toujours plus complexes se situe avant tout dans la tête des développeurs

## 21.7 Exercices

1. Ecrire un algorithme d'intégration d'une fonction entre deux bornes. En faire ensuite un générique auquel on passe la fonction en formel, puis une procédure à laquelle on passe un pointeur sur la fonction. Comparer les performances.
2. Etudier la structure d'un mécanisme de fenêtrage qui serait fondé sur le parallélisme et non sur le mécanisme des *call back*.

[1] Noter que l'habitude est de renvoyer 0, mais il serait souvent plus commode pour les algorithmes de renvoyer une valeur supérieure à l'index du dernier élément.

# Cinquième partie

# Une méthode orientée objet pour les projets de taille moyenne

Nous avons vu dans la quatrième partie qu'il était possible de définir une méthode «maison» adaptée au développement d'un projet particulier.

C'est ce processus que nous allons tenter d'illustrer dans cette partie. Le but n'est donc pas tant de développer «une méthode de plus» que de montrer comment les idées que nous avons développées peuvent être mises en œuvre à l'intérieur d'un cadre méthodologique fondé sur les méthodes existantes.

# 22

# Cahier des charges de la méthode

## 22.1 Enoncé

Comme indiqué dans le titre de cette partie, nous voulons développer une méthode pour des projets de taille moyenne (soit, en première approximation, de 10 000 à 100 000 lignes de code). Nous visons donc une classe de problèmes qui requiert une véritable équipe de développement, mais de taille telle qu'il soit encore possible d'organiser des réunions rassemblant la totalité de l'équipe. Il s'agit d'une méthode de conception détaillée : on suppose que l'on dispose d'un cahier des charges (qui n'est pas forcément ni aussi complet, ni aussi précis que l'on pourrait le souhaiter) et que le but est d'obtenir un programme réalisant ce cahier des charges. En particulier, le problème des spécifications matérielles est supposé résolu, ou tout au moins suffisamment avancé pour permettre aux développeurs logiciels de ne pas s'en soucier.

La méthode doit servir pour des applications à longue durée de vie : la documentation, la compréhensibilité de la structure du projet par des personnes n'ayant pas participé au développement initial, sont donc primordiales. En revanche, on souhaite limiter les phénomènes d'avalanche documentaire observés avec des méthodes destinées aux problèmes plus importants : la documentation doit rester concise et abordable.

La méthode ne doit pas imposer de trop fortes contraintes : il faut qu'elle aide le développeur sans lui imposer un cadre trop rigide qui risquerait, en l'absence de directives «militaires», de l'en détourner. *A priori*, la méthode s'appliquera à tous les domaines de l'informatique. En particulier, elle doit pouvoir être mise en œuvre pour des applications de type «temps réel», hors peut-être celui du temps réel à très fortes contraintes de performances et/ou de sécurité pour lequel des méthodes spécifiques sont nécessaires.

Cette méthode doit permettre d'utiliser au mieux le langage Ada. En particulier, elle doit prendre en compte les nouvelles possibilités apportées par Ada 95. Le processus de développement utilisera le maquettage progressif ; compte tenu de la taille visée, il est encore possible de gérer l'avancement du projet en travaillant ainsi ; en revanche, la facilité d'intégration qu'il apporte est un grand facteur d'économie d'effort de développement. Enfin, nous souhaitons promouvoir la réutilisation et le développement de composants logiciels, ainsi que l'évolutivité, tout en maintenant la sécurité. Nous nous plaçons donc à un niveau intermédiaire entre la méthode de Booch, qui est trop imprécise pour être utilisée directement et la méthode HOOD qui vise des projets de taille supérieure et où la réutilisation n'est pas une critère fondamental.

Pourquoi avoir choisi ce niveau pour illustrer notre propos ? Parce qu'il s'agit d'une taille de problèmes qui requiert l'utilisation d'une méthode, mais où les programmeurs peuvent encore être tentés de n'en utiliser que de façon informelle. Nous pensons donc que le «marché» de la méthode est important.

## 22.2 Quelques idées directrices

### *22.2 .1 Choix de la méthode de départ*

Nous l'avons déjà dit, mais on ne le répétera jamais assez : la complexité est l'ennemi n°1 de l'informaticien. Il faut arriver à un morcellement du travail tel que chaque élément pris individuellement reste toujours *simple* : soit qu'il s'agisse d'un petit morceau de projet aux spécifications parfaitement déterminées (on ne se préoccupe pas alors du contexte dans lequel il intervient), soit au contraire qu'il s'agisse d'un élément de plus haut niveau qui se contente d'utiliser des services de niveaux inférieurs, oubliant alors délibérément comment ceux-ci fonctionnent en interne. Le cloisonnement strict des rôles jouera donc un rôle fondamental dans la méthode. Or nous avons vu que ce cloisonnement ne peut être obtenu avec les méthodes orientées objet par classification : notre base de départ sera donc une méthode orientée objet par composition. Ceci n'exclut pas d'utiliser l'héritage, mais seulement dans les couches profondes de la conception. En fait, un des buts de la méthode est de permettre d'utiliser l'héritage fourni par Ada 95 lorsqu'il est utile, en évitant la complexité inhérente aux méthodes par classification. Nous définirons donc cette méthode comme une méthode orientée objet avec priorité à la composition. Et comme il est de bon ton de donner un nom aux méthodes, nous l'appellerons C-FOOD[1] pour *Composition-First Object Oriented Design*.

### *22.2 .2 Hiérarchisation*

La méthode doit permettre de définir des couches successives, clairement identifiées et avec une profondeur supérieure à 2 ! Trop souvent les méthodes passent directement du premier niveau d'analyse à l'implémentation. Nous chercherons au contraire à définir un processus qui favorise l'organisation du logiciel en niveaux suffisamment nombreux pour que la complexité de chacun reste dans les limites de ce qui est facilement compréhensible.

### *22.2 .3 Documentation et méthodologie*

Nous avons précédemment expliqué que les documents qui servent au processus de conception ne sont pas les plus appropriés à la maintenance. Nous introduirons donc dans la méthode une phase de documentation de la solution obtenue, qui ne proviendra *pas* des documents de conception.

On cherchera à *minimiser* cette documentation : elle ne devra comprendre que ce qui est nécessaire à la compréhension de la *structure* du programme. On rendra au contraire le *code* aussi autodocumenté que possible, en ne laissant à la documentation externe que ce qui ne peut s'exprimer directement au niveau du langage[2]. Nous utiliserons même Ada (avec, en particulier, l'aide du paquetage ADPT), comme moyen d'expression de nos conceptions.

### *22.2 .4 Phases*

Toute méthode est un processus itératif ; elle permet d'identifier des sous-problèmes, que l'on résoud en leur appliquant à nouveau la méthode (cf. paragraphe 4.3). Cependant, il existe des éléments à analyser au début du projet, avant de pouvoir lancer l'analyse itérative ; et il est bon de

---

[1] Jeu de mots intraduisible : C-FOOD se prononce en anglais comme «sea food» (fruits de mer), particulièrement appréciés par l'auteur !

[2] Ceci n'est bien entendu rendu possible que par le très grand pouvoir d'expression du langage Ada ; ce n'est pas forcément applicable à des langages de plus bas niveau.

tirer un bilan après achèvement de la conception. Ces étapes doivent être formalisées au niveau de la méthode, mais sont distinctes du processus général de conception itérative. Aussi définirons-nous trois phases dans notre méthode : la phase d'initialisation où nous mettons en place la structure du projet, la phase de conception itérative proprement dite et une phase de conclusion une fois celle-ci terminée.

# 23
# Description de la méthode

## 23.1 Principes de la méthode

La méthode que nous proposons ne fait plus de différence entre conception préliminaire, conception détaillée, analyse et codage. Elle définit une suite progressive de descriptions du problème, par niveau sémantique décroissant. Chaque étape est validée par des techniques de maquettage avant de passer à l'étape suivante. La méthode est itérative et s'arrête lorsque la description obtenue est complète, c'est-à-dire ne comporte plus de partie maquettée.

Elle est essentiellement fondée sur la méthode de Booch, le critère de décomposition horizontale étant l'objet en tant qu'abstraction d'un objet du monde réel et le critère vertical le niveau d'abstraction. Les changements que nous proposons visent à systématiser dans ce cadre le maquettage progressif et l'utilisation (ou la définition) de composants logiciels réutilisables, à préciser le processus de conception lui-même et à définir une politique de documentation. Nous avons de plus incorporé certaines idées provenant d'autres méthodes apparentées, de HOOD notamment.

## 23.2 Première phase : initialisation

### *23.2 .1 Comprendre le problème*

Avant même de songer à une éventualité de *solution*, il est indispensable de s'assurer que l'on a bien compris le *problème*[1]. On dispose généralement d'une description, que nous appellerons pour simplifier le cahier des charges. Rien ne prouve *a priori* que ceux qui l'ont rédigé[2] maîtrisaient eux-mêmes la complexité de ce qu'ils ont demandé. Il faut donc commencer par faire une revue de détail du cahier des charges, s'assurer que la description du travail à effectuer est claire et faire la chasse aux imprécisions, manques, non-dits et contradictions. On supposera au départ que ces imperfections existent ; si l'on n'en trouve pas, c'est que notre compréhension du problème est insuffisante.

Toute zone d'ombre dans le cahier des charges doit être discutée en équipe par les développeurs et faire l'objet d'une demande de clarification auprès du client s'il apparaît qu'elle résulte effectivement d'un défaut du cahier des charges. Rien n'est plus dangereux en effet que de partir sur une interprétation «évidente», qui se révélera (mais trop tard) ne pas correspondre à ce qui était effectivement demandé. En particulier, penser que le client a rédigé le cahier des charges en utilisant

---

1 Ceci correspond à l'étape H1.1 de la méthode HOOD.

2 Que nous appellerons pour simplifier le «client». Il peut s'agir effectivement d'un client extérieur, aussi bien que d'une autre équipe de la même entreprise dans le cas de développement de gros projets.

son propre «fonds culturel». Des notions, des termes qui lui paraissent évidents peuvent être interprétés de façon différente par l'équipe de développement. On veillera donc en particulier à ce que tous les termes employés, *surtout ceux qui bénéficient d'un sens communément accepté*, soient définis de façon précise dans le cahier des charges.

## *23.2 .2 Faire les choix fondamentaux*

### a) Choix organisationnels

L'équipe de programmation peut être ou non définie au départ du projet. Si elle ne l'est pas, c'est le premier travail du chef de projet que de la constituer. Il convient d'affecter les *rôles* que nous avons présentés, de s'assurer que chacun a bien compris ses propres responsabilités et celles de ses collègues.

### b) Choix techniques

Il convient ensuite de faire les choix techniques qui s'imposeront à l'ensemble du projet. Un certain nombre d'entre eux sont typiquement du ressort des programmeurs, mais certains doivent être décidés dès le départ du projet. Nous rappelons ici ceux qui sont les plus critiques pour la bonne marche future du développement.

*Choix de la méthode de conception*

Il convient d'être parfaitement clair dès le départ quant à la méthode choisie. Nous avons vu les différents éléments à prendre en compte pour choisir la méthodologie la plus appropriée ; mais quitte à paraître un peu cynique, nous affirmerons que peu importe la méthode choisie, l'important est d'en avoir une, de la suivre et d'être cohérent avec elle sur toute la durée du projet. La responsabilité de ce choix est du ressort du chef de projet, assisté du responsable qualité, en accord avec les programmeurs... dans la mesure où ils sont déjà connus. Il arrive en effet que l'on choisisse (ou impose) une méthode pour un projet, puis que l'on désigne les développeurs dont les connaissances correspondent au besoin.

*Organisation de la bibliothèque de programme*

Le responsable configuration doit dès le départ étudier les possibilités de l'environnement de développement choisi pour définir les procédures de mise en configuration des modules livrés et les conditions de mise à disposition auprès des autres membres de l'équipe.

Il doit aussi décider de l'utilisation de fonctionnalités comme les bibliothèques liées ou les sous-bibliothèques par rapport à la structure de l'équipe.

*Gestion des anomalies*

Nous avons vu dans la quatrième partie diverses possibilités de gestion des anomalies. Il faut décider d'une politique globale pour tout le projet. Même si le choix final se porte sur une solution «triviale» (comme d'utiliser simplement le mécanisme des exceptions), il importe que ceci résulte d'une décision concertée et non simplement de l'absence de décision à ce sujet. Cette décision doit être prise par le chef de projet, assisté du responsable qualité, en contact direct avec les programmeurs.

*Choix des composants de base*

Il est généralement possible dès ce niveau de se douter que le projet devra utiliser certains composants logiciels : structures de données habituelles (piles, files, etc.), bases de données, interface avec un système de fenêtrage...

Le chef de projet, en liaison avec le responsable composant logiciel, devra décider du choix (et de l'achat éventuel) de tels composants. Bien sûr, s'il existe déjà une base «maison» de tels composants, le plus simple sera de l'utiliser (à moins qu'elle ne réponde pas aux besoins prévus). Sinon, il faudra étudier le marché pour décider soit d'acheter des composants tout faits (et les choisir, car ce marché est aujourd'hui concurrentiel), soit de les faire développer dans le cadre du projet. Eventuellement, le responsable financier pourra être chargé de négocier avec la hiérarchie une «rallonge» supplémentaire (en autofinancement) pour permettre le développement de ces composants d'une manière suffisamment réutilisable pour qu'il puisse servir dans des projets ultérieurs.

### c) Mise en place des documents

Toute méthode s'accompagne d'un certain nombre de documents. Il faut dès le départ être clair sur qui écrit quels documents, comment ces documents sont conservés et comment on peut y avoir accès. Selon les cas et les outils utilisés, ces documents seront sur papier, sur support informatique, multimédia... Peu importe, du moment que les choix sont, comme pour le reste, délibérés et explicites. La mise en place des documents est bien entendu de la responsabilité du responsable documentation, en accord avec le responsable communication et sous contrôle du responsable qualité.

Certaines méthodes, telles HOOD, sont très explicites quant à la documentation et les outils la gèrent directement ; il y a alors peu de décisions à prendre. D'autres sont beaucoup plus vagues, parfois totalement silencieuses, quant à la définition des documents à produire. D'autre part, les habitudes de l'entreprise en matière de documentation sont également à prendre en considération. Nous pensons cependant que quelle que soit l'organisation pratique adoptée, on doit trouver sous une forme ou sous une autre au moins trois documents : l'historique, la documentation d'utilisation et le cahier des décisions de conception.

L'*historique* récapitule par ordre chronologique les grandes étapes du projet. Il est tenu par le responsable communication et contient entre autres les comptes rendus des réunions et les étapes marquantes du développement. Il doit permettre de savoir où l'on se trouve par rapport au plan de développement prévu... ou permettre de retrouver *a posteriori* l'origine d'un glissement.

La *documentation d'utilisation* est le mode d'emploi de chacun des modules. Elle doit permettre à un utilisateur de se servir du module sans aller regarder dans l'implémentation. Elle peut rappeler la spécification syntaxique (à moins que l'on ne dispose de liens hypertexte...), mais doit surtout s'attacher à décrire ce qui n'apparaît *pas* d'après la syntaxe : pré- ou post-conditions, cas de levées d'exceptions, sémantique de haut niveau, éventuellement temps et performances d'exécution.

Le *cahier des décisions de conception* doit contenir pour chaque choix qui a été fait les motivations de ce choix, les autres possibilités envisagées et les raisons ayant conduit à préférer la solution adoptée. En particulier, lorsque des possibilités ont été étudiées, puis rejetées, il faut conserver les raisons de ce rejet. Le cahier doit être organisé par module (pour les décisions spécifiques à chacun d'eux) et doit permettre à un mainteneur chargé de reprendre le projet par la suite de trouver la réponse à toute question de la forme : «Mais pourquoi ont-ils fait cela ?», afin de lui éviter de répéter les erreurs qui ont pu être commises (puis réparées) durant la phase initiale de conception.

Il faut absolument séparer la documentation d'utilisation du cahier des décisions de conception, car leurs rôles sont fondamentalement différents. Dans une voiture, le premier correspondrait au manuel remis à l'acheteur et le second serait le manuel technique pour les garagistes. Le tableau de la figure 37 résume ces différences.

| | **Documentation d'utilisation** | **Cahier des décisions de conception** |
|---|---|---|
| Lié à | Spécification | Corps |
| Destiné à | Utilisateur du module | Mainteneur du module |

| | | |
|---|---|---|
| Décrit | Comportement abstrait | Choix d'implémentations |

**Figure 37 :** Documentation d'utilisation et documentation de conception

La première partie du cahier des décisions de conception doit récapituler les décisions prises lors de cette phase et leurs motivations.

### *23.2 .3 L'analyse de premier niveau*

La plupart des méthodes ne font pas un cas particulier de la première étape. En particulier, dans les méthodes orientées objet, il faut considérer le système à concevoir comme un objet (au sens de la méthode), semblable à ceux qui seront produits par le déroulement de l'analyse. Cela conduit généralement à des difficultés ; le cahier des charges est rarement exprimé en termes «orienté objet», mais plus souvent en termes fonctionnels. L'expression des exigences sous forme d'objets est alors rarement naturelle, conduisant à baptiser «objets» des éléments qui ne sont pas vraiment des abstractions.

Nous préférons reconnaître qu'en général, le premier niveau d'analyse s'exprime mieux sous forme fonctionnelle. Puisque le cahier des charges exprime un *comportement*, ce premier niveau d'analyse, qui se concrétisera par le programme principal, peut parfaitement décrire ce comportement. Pour sa réalisation, on utilisera les opérations d'*objets* qui sont les composants de premier niveau du système. On peut donc voir le programme principal comme une sorte d'automate, dont le fonctionnement met en œuvre, anime, des objets. Cette étape diffère donc de la conception itérative essentiellement en ce que le point de départ ne peut être considéré comme un objet. Elle conduit cependant à l'identification, puis à la définition, d'objets de la même façon que celle décrite dans la phase de conception itérative.

## 23.3 Deuxième phase : la conception itérative

Nous allons maintenant aborder la phase principale du développement. Elle est itérative et, comme pour une boucle de programme, il faut définir les invariants de boucle et les conditions de terminaison.

Par «invariant de boucle», nous entendons qu'à chaque étape nous devons partir d'une description bien définie des *objets à concevoir* pour aboutir à la définition d'objets de plus bas niveau nécessaires à leur réalisation, décrits de la même manière. Une étape de la conception itérative correspond donc à un plan d'abstraction : à partir de spécifications existantes, réalisation du *corps* de l'*objet à concevoir* et définition des *spécifications* des objets nécessaires, que nous appellerons pour les différencier les *objets utilisés*. Le plan d'abstraction est ensuite vérifié en le compilant et éventuellement en fournissant des corps-maquettes permettant d'exécuter l'ensemble. Les objets utilisés deviennent alors les objets à concevoir de l'itération suivante.

Le processus itératif s'arrête soit quand les objets utilisés peuvent être réalisés directement sans faire appel à des objets de plus bas niveau, soit quand tous les objets utilisés existent déjà, dans le projet ou dans des bibliothèques d'objets réutilisables.

Cette description formelle du processus ne doit surtout pas faire croire qu'une fois la spécification d'un objet établi, elle est immuable ; nous verrons même que notre méthode prévoit explicitement une phase de remise en cause des éléments de l'étape précédente. Il arrivera donc souvent qu'une étape ultérieure conduise à un changement d'interfaces d'objets déjà établis et provoque donc une modification d'implémentation d'un plan d'abstraction supérieur. C'est normal ; mais quand cela se produit, il faut revenir en arrière, revérifier toutes les conséquences de la nouvelle spécification (éventuellement en remettant à jour les maquettes associées) avant de donner son aval à la nouvelle spécification. Le processus de conception doit donc être vu comme *globalement descendant*, mais peut présenter des *oscillations locales*, comme illustré dans la figure

38. Autrement dit, ce qui est important, c'est qu'à chaque remise en cause on remonte un peu moins haut et que l'on descende un peu plus bas.

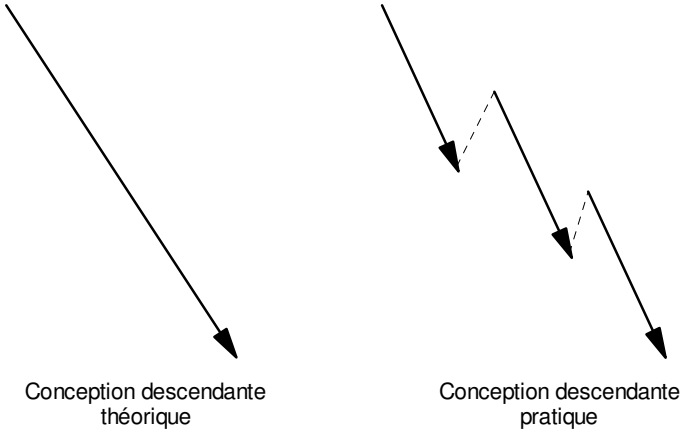


**Figure 38 :** Modèles de conception descendante

Notre hypothèse de départ en début d'itération est que nous disposons de définitions des objets à concevoir, syntaxiquement sous forme de spécifications Ada et sémantiquement sous la forme d'une documentation décrivant leurs propriétés. Ces spécifications sont supposées suffisantes pour réaliser le plan d'abstraction qui les a définies. Nous allons maintenant détailler la démarche de conception que nous proposons.

## *23.3 .1 Reprendre les spécifications*

En entrée de cette étape, nous disposons de spécifications d'un objet à concevoir. En sortie, nous devons produire des spécifications révisées.

Un objet résultant d'une phase d'analyse précédente a été simplement *défini spécifiquement*, c'est-à-dire que l'on a identifié une certaine abstraction dans le but de résoudre un problème particulier. Celui qui a identifié l'objet en a nécessairement eu une vue étroite : celle limitée aux seules fonctionnalité nécessaires à la solution de son problème. La spécification qui nous sert en entrée doit donc être considérée comme un minimum à réaliser : elle correspond aux besoins immédiats de celui qui l'a définie.

La première étape consiste à étudier le problème, c'est-à-dire analyser l'objet du monde réel dont l'*objet à concevoir* est une abstraction. Nous utiliserons toujours le terme «objet», bien qu'il puisse s'agir d'une classe, ou même parfois d'unités purement fonctionnelles. Afin de ne pas être influencé par cette vue particulière, il est bon d'ignorer *a priori* le contexte dans lequel l'objet sera utilisé. Si la taille de l'équipe le permet, il est préférable que la réalisation de l'objet soit confiée à une personne qui n'a pas participé à sa définition[1]. Le point important est de répondre à la question : cet objet à concevoir, qu'est-ce que *c'est* ? Pour des objets de haut niveau, c'est-à-dire au début du projet, la réponse est loin d'être triviale et peut conditionner largement l'architecture de la solution. Il est bon alors de discuter du problème en réunion générale de toute l'équipe. On sera surpris de la différence de vues que des personnes différentes peuvent avoir d'objets apparemment simples.

Par exemple, dans un système informatique d'entreprise, on identifiera la nécessité d'un objet «client». Il faudra se demander ce qu'*est* réellement un client. Dans un projet où nous avons été amené à intervenir, nous avons pu constater que le terme «client» recouvrait des entités tellement différentes que nous avons dû abandonner totalement cette notion au profit de plusieurs autres dont la définition était moins controversée, comme «responsable légal», «prospect», «destinataire de livraison»...

[1] Ce qui n'empêche pas le réalisateur d'aller demander des précisions à ses «clients», voire même de négocier l'interface.

Il faut conserver dans le cahier «historique» les minutes de ces réunions, avec les différentes idées qui y sont apparues. Il est fréquent qu'une direction rejetée lors de la discussion apparaisse plus tard comme une possibilité intéressante : l'archivage des discussions permet alors de repartir sur une nouvelle voie.

Cette étape produit la spécification Ada définitive de l'objet et une fiche de description de l'objet qui présente ce que l'objet *est* et ce qu'il *fait* du point de vue abstrait, c'est-à-dire sans référence à une quelconque implémentation possible. Si des ajustements ont été nécessaires, cela peut avoir une influence sur le niveau d'abstraction précédent ; en particulier, cela peut entraîner des modifications dans le corps des objets du niveau supérieur. Ceci est *normal*. Un grand danger de la conception est de figer trop tôt certaines décisions : on continue sur la même voie sous prétexte de ne pas perturber l'existant et bien souvent on s'aperçoit, mais beaucoup plus tard, que la modification est absolument indispensable et la perturbation risque d'être beaucoup plus importante. Il vaut donc mieux admettre qu'il existe des (petites) «oscillations» des spécifications à la frontière entre deux niveaux d'abstraction. L'important avec notre méthode est que ces perturbations sont limitées au niveau immédiatement supérieur et ne «percolent» pas dans toute la hiérarchie du projet.

## *23.3 .2 Définir le plan d'abstraction*

En entrée de cette étape, nous disposons des spécifications définitives d'un objet à concevoir. En sortie, nous devons produire le corps de cet objet et les spécifications des objets utilisés par ce corps. Rappelons que c'est l'ensemble constitué d'une implémentation (vue concrète du niveau *N*) et des spécifications des objets de plus bas niveau utilisés (vues abstraites du niveau *N*+1) que nous appelons un plan d'abstraction.

L'objet à concevoir étant maintenant correctement défini, l'étape va consister à identifier les sous-objets qui le composent. Ceci conduit à élaborer une stratégie informelle d'implémentation. Bien entendu, diverses solutions sont possibles ; il convient d'en envisager plusieurs afin d'être à même de faire un choix. C'est la phase de plus pure création intellectuelle, donc celle la moins susceptible d'une formalisation. L'expérience et l'exemple restent les meilleurs moyens pour mener à bien cette étape.

Nous allons présenter les différents points importants dans l'ordre où normalement on les aborde. L'ordre séquentiel de la présentation ne doit pas être pris comme une obligation ; en général, on se préoccupera plus ou moins simultanément de ces différents aspects, de même qu'il est impossible d'étudier séparément les différents objets du plan d'abstraction : l'analyse porte sur plusieurs d'entre eux simultanément.

### a) Identifier les objets

Il s'agit ici d'identifier les sous-objets nécessaires à la réalisation de l'objet à concevoir. Dans les premières phases du projet, notamment au premier niveau, il est bon d'effectuer ce travail en groupe. Les participants proposent des stratégies permettant de réaliser les différentes fonctionnalités de l'objet à concevoir. Le concepteur principal note au tableau[1] les mots employés par le groupe qu'il juge significatifs ou intéressants. Ceci donne lieu à de nouvelles discussions, où l'on va chercher à remplacer des mots au tableau par d'autres jugés plus appropriés. Ce processus n'est pas une vaine querelle de termes, mais une phase extrêmement importante qui va permettre de préciser, puis d'isoler les futurs objets. Au bout d'un certain temps, quelques notions essentielles émergent : on va alors les identifier comme des objets potentiels et essayer de leur rattacher les opérations qui ont pu apparaître dans la discussion. On ne cherche pas à ce stade à *définir complètement* les objets (on le fera lors de l'itération suivante), mais surtout à les *identifier pour la vue que l'on en a actuellement*.

[1] Il est absolument *indispensable* qu'une salle de réunion pour ce genre d'activités dispose d'un tableau blanc (ou noir !).

Les opérations à effectuer par chaque objet peuvent surgir naturellement dans la discussion. Dans d'autres cas, certaines opérations peuvent apparaître sans être *a priori* rattachées à un objet ; il faut dans ce cas déterminer à quel objet elles appartiennent.

Plusieurs difficultés ou erreurs sont susceptibles d'apparaître à ce niveau. La première difficulté est de s'assurer que les objets identifiés correspondent bien à des entités logiques. S'il s'agit d'une abstraction d'un objet du monde réel (personne, terminal, régulateur...), cela ne pose pas de problèmes. Mais on introduit parfois des objets plus informatiques, qui courent le risque de ne pas être de vrais objets, mais simplement des noms utilisés pour baptiser des éléments de décomposition fonctionnelle. On se méfiera particulièrement des objets qui portent des noms d'*action* : «Gestionnaire de...», «Contrôleur», ... Le meilleur conseil à ce niveau est d'essayer, *même pour des objets apparemment totalement informatiques*, de trouver des analogues dans le monde réel et de s'accrocher à ces images. On évitera de parler d'une «table de relation prix-produit», ou d'un «driver d'interruption», pour dire plutôt objet «tarif» ou objet «clavier».

Ensuite, il est possible d'identifier trop tôt des objets qui appartiennent en fait à des niveaux plus profonds. Le problème est grave, car il correspond souvent à des surspécifications. On a une idée précise de l'*implémentation* d'un objet et l'on confond l'objet avec son implémentation. Par exemple, si l'on veut transcoder des caractères (c'est-à-dire les faire passer d'un code, comme Latin-1, vers un autre, comme celui de l'IBM-PC), il paraît évident qu'il faut une table de transcodage. *Evident* ? Pas du tout ! Le besoin est celui d'une *fonction* de transcodage. Cette fonction peut être *implémentée* trivialement au moyen d'une table, mais ce n'est pas la seule façon de faire. La preuve : si nous traitions des `Wide_Character` (jeu de caractères 16 bits), il serait beaucoup trop coûteux d'effectuer le transcodage par une table.

Enfin, il arrive fréquemment que l'on ne rattache pas les opérations aux bons objets. Remarquons à ce sujet la *dissymétrie* fondamentale de l'approche objet. Dans un modèle entités-relations, les opérations servent à relier les données, sans leur être attachées et sont fondamentalement bidirectionnelles. On dira indifféremment qu'un client *ouvre* un compte ou qu'un compte *est ouvert* par un client. En approche objet, nous devrons décider si `Ouvrir` est une opération du client ou du compte. Une technique qui fonctionne bien pour résoudre ce problème consiste à se poser la question «qu'est-ce qui change si...». Si une modification dans la nature d'un objet nécessite une modification de l'opération, alors il s'agit d'une opération de l'objet. Par exemple, pour ouvrir le compte nous pouvons supposer qu'il est nécessaire de passer la référence du client, le montant initial et le type du compte. Si nous décidons d'augmenter le nombre de caractères du nom du client, cela n'aura aucune influence sur la procédure d'ouverture elle-même. Si en revanche nous voulons modifier la notion de compte pour introduire des comptes à intérêt, il faudra passer un paramètre supplémentaire (le taux). Nous pouvons en déduire qu'`Ouvrir` est une opération du compte et non du client.

S'il était encore relativement facile de trouver la solution correcte dans l'exemple précédent, il faut être conscient que les fautes de rattachement sont courantes. Elles se traduisent souvent en Ada par des difficultés de compilation, notamment au niveau des types ou des visibilités. En particulier, si l'on a décidé de faire un type privé et que l'on s'aperçoit que des modules extérieurs ont besoin de voir sa structure interne, il faut sérieusement envisager la possibilité d'une erreur de rattachement avant de s'empresser de rendre public le type privé, ou d'utiliser des unités enfants pour y accéder.

Retenons donc qu'une décision de conception à ce niveau ne saurait être définitive ; il importe au contraire d'être prêt à remettre en question ce qui pouvait sembler acquis cinq minutes plus tôt. Il faut penser à noter dans les minutes de la réunion les raisons de ces changements, afin de prévenir d'éventuels remords ultérieurs dus à l'oubli des raisons de la marche arrière. En fin d'étape, on décrit les objets identifiés sous forme de spécifications Ada, que l'on compile pour fournir un premier niveau de vérification.

### b) Ecrire le corps des opérations de l'objet à concevoir

Les composants étant spécifiés, il faut les assembler pour construire l'implémentation de l'objet d'origine. Concrètement, ceci signifie que nous pouvons écrire le corps en principe définitif de l'objet à concevoir avec toutes ses opérations. Ce corps sera compilé ; un problème à ce niveau signifie généralement que les objets utilisés ne sont pas correctement définis. On ne s'étonnera donc pas de trouver ici encore quelques oscillations entre la définition des objets utilisés et l'implémentation des opérations de l'objet à concevoir. Noter que les corps sont des enchaînements d'actions, effectuées par les opérations des objets utilisés. Une analyse en programmation structurée est donc tout à fait appropriée pour décrire cette phase.

On essayera d'éviter d'avoir à documenter ces implémentations ; en effet, il s'agit d'une partie purement algorithmique et nous avons déjà dit combien nous préférions nous appuyer sur un code autodocumenté. Bien que cela représente un idéal parfois impossible (et souvent difficile) à atteindre, la simple lecture du code ne doit laisser que peu de doutes au lecteur *qui aurait lu et compris la documentation de spécification des composants utilisés*, ce qui est le minimum que l'on soit en droit d'exiger de lui.

A l'issue de cette étape, on dispose d'un plan d'abstraction complètement compilé.

### c) Contester

A ce stade, on a souvent une idée préconçue de l'implémentation des objets qui viennent d'être identifiés, souvent guidée par des expériences précédentes. Nous allons donc les soumettre «à la torture», afin de nous assurer qu'ils réagissent correctement.

*Abstraction*

Première épreuve : imaginer des implémentations loufoques. Nous avons besoin d'une simple petite table de 10 éléments ? Cela ne fait rien, imaginons si une implémentation sous forme de mémoire virtuelle avec pagination sur disque aurait une influence *sur les spécifications*. Le but n'est pas tant de permettre de réaliser vraiment ces implémentations «irréalistes» que de s'assurer de la bonne indépendance des spécifications par rapport à *toute* technique d'implémentation, c'est-à-dire que la spécification est réellement abstraite. Si ce n'est pas le cas, peut-être peut-on concevoir un objet de plus haut niveau, réellement abstrait, qui utiliserait l'objet en cours de revue pour son implémentation ; il faut alors garder l'objet courant pour plus tard et définir ces objets de plus haut niveau que nous avions en quelque sorte court-circuités.

*Réutilisabilité*

Deuxième épreuve : imaginer comment réagit la conception à des changements des exigences. Essayer d'imaginer de nouveaux services que pourrait demander le «client» (c'est-à-dire celui qui nous impose les spécifications de l'objet à définir). Les objets définis nous permettraient-ils de répondre à la demande ?

> Idéalement, il suffirait d'assembler les mêmes composants différemment, preuve que notre conception est à la fois générale et puissante. Eventuellement, il faudrait rajouter de nouveaux composants, qui ne perturberaient pas ceux que nous avons définis : on peut alors dire que le système est peut-être incomplet, mais extensible.
> Moins bon : il faut généraliser les composants existants en leur rajoutant des fonctionnalités nouvelles ; ne pourrait-on le faire tout de suite ? Il est souvent plus simple de prévoir largement les services fournis au moment de la conception initiale que d'avoir à les augmenter ensuite.
> Mauvais : il faut modifier le composant ; une évolution risque alors de perturber non seulement le composant, mais également ses utilisateurs. On a alors un phénomène de

diffusion de l'action d'évolution dans des parties du projet qui n'auraient pas dû être concernées. Il est impératif de remettre en cause la solution choisie.

*Elémentarité*

Troisième épreuve : l'objet est-il élémentaire ? Chaque objet doit être une abstraction complète et insécable : il prend en charge TOUT un aspect d'UNE SEULE entité. Il faut donc vérifier qu'il n'est pas possible de le séparer en deux objets distincts. En bonne logique, il faudrait aussi vérifier que deux modules séparés ne peuvent pas être réunis en un seul objet ; en pratique, ce test est moins nécessaire, car lorsque ce problème surgit, il aboutit souvent à des impossibilités de compilation, diagnostiquées lors de l'étape précédente.

On doit également se poser les questions suivantes : l'objet est-il simple ? Correspond-il naturellement à un objet du monde réel ? A la limite, on devrait pouvoir prendre n'importe quel non-informaticien qui passe dans le couloir et lui expliquer ce qu'*est* l'objet.

### d) Caractérisation

Nous allons maintenant nous livrer à une phase de *caractérisation* des objets utilisés, c'est-à-dire que nous allons essayer de voir comment ils se positionnent par rapport aux diverses classifications de la troisième partie de ce livre. Le but de cette étape est double : vérifier la vraisemblance du composant (un composant inclassable a toutes les chances de résulter d'une erreur de conception) et préciser la sémantique des objets.

En particulier, nous devons vérifier si l'objet est *clairement* une machine abstraite ou un type de donnée abstrait. Dans ce dernier cas, nous devons préciser s'il est à sémantique de référence ou à sémantique de valeur. Il faut se demander si la notion qu'il recouvre est susceptible de donner naissance à une classe de plusieurs objets. Enfin un objet utilisé par plusieurs niveaux d'abstraction doit nous faire penser à un bus logiciel, susceptible de produire un composant réutilisable. Rappelons que parfois un composant échappera à la classification[1], mais que cela doit attirer l'attention du contrôle qualité qui devra s'assurer que cela ne dissimule pas une erreur de conception.

### e) Chercher à réutiliser

Maintenant que nous connaissons l'entité à réaliser, nous allons chercher dans l'existant, types du projet et bibliothèques de composants logiciels, s'il n'existe pas un composant que nous pourrions réutiliser au lieu d'avoir à en développer un nouveau.

Bien entendu, il sera extrêmement rare de trouver *exactement* le composant adapté à nos besoins ; s'il existe quelque chose de suffisamment proche, il faudra envisager d'adapter la stratégie de l'objet à concevoir pour permettre de réutiliser les composants existants. Si nous avons l'impression que l'abstraction nécessaire existe parmi nos composants logiciels, mais ne fournit pas les services nécessaires, il est possible que nous disposions du bon composant réutilisable, mais incomplet. On préférera alors enrichir le composant existant plutôt que d'en développer un nouveau, en prenant garde de conserver la nature «généraliste» du composant existant, même si ce n'est pas l'optimum pour notre problème particulier. Parfois, nous pourrons trouver que la similitude de l'objet désiré avec un objet existant provient de ce qu'ils appartiennent logiquement à une même famille ; nous pourrons alors envisager de créer une classe plus générale dont nous ferions hériter l'ancien et le nouvel objet. Si enfin nous ne trouvons rien de satisfaisant, il faudra développer un nouveau composant. Dans ce cas, on prendra soin de définir un composant *franchement différent* de ceux qui existent déjà. Au besoin, on fera légèrement évoluer les spécifications pour les *éloigner* des objets existants. La raison de cette règle est simple : ou bien le nouveau composant correspond à la même abstraction que des composants existants et il faut faire converger les vues plutôt que de définir à nouveau ; ou bien il s'agit d'une abstraction différente et il faut se méfier des effets de mimétisme qui font qu'un nouveau composant est influencé par les composants connus. Ce qu'il faut

[1] Une pensée émue pour les naturalistes qui découvrirent l'ornithorynque...

éviter absolument, c'est de développer plusieurs abstractions légèrement différentes du même objet. Si l'on commence dans cette voie, on aboutit à une prolifération incontrôlable de versions. Noter que la classification tend à officialiser cette prolifération en l'organisant (chacun développe sa propre version d'un composant, en ne recodant que la différence), alors que la composition tend à l'empêcher.

## *23.3 .3 Tester le plan d'abstraction*

En entrée de cette étape, on dispose de la spécification du plan d'abstraction. Il faut produire les corps maquettes des objets utilisés pour tester le plan d'abstraction et servir de référence ultérieure pour la conception.

La compilation du plan a permis de terminer l'implémentation de l'objet à concevoir et de la vérifier sur le plan syntaxique, ce qui donne déjà quelque espoir d'avoir évité les incohérences. Si le problème est suffisamment simple, on peut en rester là. Mais il faut souvent vérifier, au moins partiellement, le comportement *dynamique* du plan d'abstraction. Ceci s'obtient en écrivant des corps «maquettes» des objets utilisés grâce au paquetage `ADPT`. Une utilisation judicieuse des durées simulées sert à ajuster le «budget temps» des opérations. Cette maquette est conservée comme documentation dynamique de la façon dont le concepteur voyait le comportement de ses objets utilisés. Elle peut aussi servir à faire tourner l'ensemble du programme, notamment en cas de développement en équipe.

Le projet global peut être pris comme programme de test : on intègre tout le plan d'abstraction, y compris ses sous-implémentations maquettes, et l'on regarde si cela fonctionne. Si tout va bien, c'est parfait. Mais si l'on découvre des erreurs, il peut être difficile de les piéger dans le contexte de l'ensemble du projet. Aussi est-il préférable d'écrire des programmes de tests unitaires pour chaque plan d'abstraction. Ces programmes seront conservés comme documentation et pour effectuer des tests de non-régression par la suite. Il est possible (et même préférable) d'écrire le test unitaire *avant* l'implémentation du module testé. On peut vérifier le test au moyen de l'implémentation maquette de l'unité que l'on est sur le point de réaliser effectivement. Ceci garantit l'aspect «boîte noire» du test et permet de se prémunir contre des déviations dues à l'implémentation (ce qui n'empêche pas d'enrichir le test par la suite, suite à la découverte d'erreurs qu'il n'avait pas initialement diagnostiquées). Bien sûr, le test peut faire découvrir des erreurs ; les spécifications et les divers documents précédents peuvent donc encore évoluer.

## *23.3 .4 Documenter le plan d'abstraction*

En entrée de cette étape, nous disposons d'une description opérationnelle du plan d'abstraction. Nous devons produire la documentation associée.

### a) Cahier des décisions de conception

Ce cahier est lié à la notion d'implémentation. On note les éléments importants de la réalisation de l'*objet à concevoir* : stratégie informelle, raisons qui ont amené à définir les objets utilisés et choix stratégiques qui ont été faits. C'est là qu'il faut noter les fausses pistes éventuellement abandonnées, ainsi que les conseils pour l'évolution future, en particulier si on a prévu des possibilités d'évolution non réalisées dans cette version... et toute autre information utile pour le mainteneur futur, notamment les erreurs ou pièges à éviter. On peut inclure une représentation graphique du plan d'abstraction, comprenant au moins les dépendances logiques et éventuellement certains flots (de données, d'appels, d'exceptions) importants. Des diagrammes de HOOD (ou inspirés de HOOD) sont tout à fait appropriés pour cela.

### b) Documentation d'utilisation

Si l'analyse critique de l'objet à concevoir a conduit à une modification des spécifications, il faut remettre à jour la documentation d'utilisation correspondante. On ajoute ensuite un chapitre pour chacun des *objets utilisés*. Rappelons que le but des descriptions est de fournir la représentation abstraite nécessaire à l'utilisateur du module qui ne connaît pas l'implémentation. L'information doit être concise et pertinente : ce qu'il faut savoir pour *utiliser* l'objet en question, ni plus, ni moins. Penser à spécifier le comportement en cas de paramètres anormaux : à qui incombe la charge des vérifications ? Des exceptions peuvent-elles être levées ? Il est inutile en revanche de détailler ce qui se déduit de la spécification Ada : si un paramètre a le sous-type `Natural`, la syntaxe exprime déjà que tout appel avec une valeur négative lèvera l'exception `Constraint_Error`.

## *23.3 .5 Itération*

A ce point, les objets sont prêts à entrer à leur tour dans le cycle de réalisation. Rien n'oblige à les confier à ceux qui les ont définis. En fait, on augmentera les chances d'indépendance des modules en *évitant* que ceux qui définissent les objets les implémentent.

On gère dans le projet une liste d'objets à implémenter : à l'issue d'une itération, les développeurs rajoutent les nouveaux objets à la fin de la liste et prennent comme nouveau travail la tête de liste. Une telle façon de faire constituera un excellent test de la compréhensibilité des spécifications !

## 23.4 Troisième phase : récapitulation et analyse

Une particularité de notre méthode est qu'il n'y a pas du tout de phase d'intégration, ou plus exactement que l'intégration a lieu en continu tout au long du projet : depuis le début, le système est complet (vu de l'extérieur), même s'il est uniquement constitué de maquettes grossières. Chaque étape va faire descendre le niveau des maquettes jusqu'à ce qu'elles aient totalement disparu : à ce point la réalisation est terminée. Mais à tout moment, le projet est dans un état *stable*, *complet* et *vérifié*.

Une fois le projet terminé, il est bon de se livrer à une récapitulation (*debriefing*) du projet. On en analyse les qualités et les faiblesses. Certains choix du début ont pu se révéler à l'usage non optimaux ; le gain apporté par une autre solution n'était pas suffisant pour justifier un retour en arrière, mais il est bon pour l'avenir d'identifier les points sous-estimés lors du choix initial qui ont fait passer à côté d'une meilleure solution. On peut identifier des possibilités d'évolution pour une version ultérieure : des fonctionnalités qui n'étaient pas demandées, mais pourraient se révéler utiles plus tard ; des modifications de structure qui n'ont pu être effectuées par manque de temps, mais que l'on pourrait entreprendre à l'occasion d'une évolution du logiciel, etc. Enfin, on se livre à une analyse de réutilisabilité. On cherche des modules développés de façon spécifique, mais qui seraient candidats à une généralisation ultérieure pour les transformer en composants réutilisables, selon le processus d'*analyse a posteriori* présenté dans la troisième partie de ce livre.

## 23.5 Résumé de la méthode

Les phases de la méthode sont résumées dans le schéma de la figure 39.

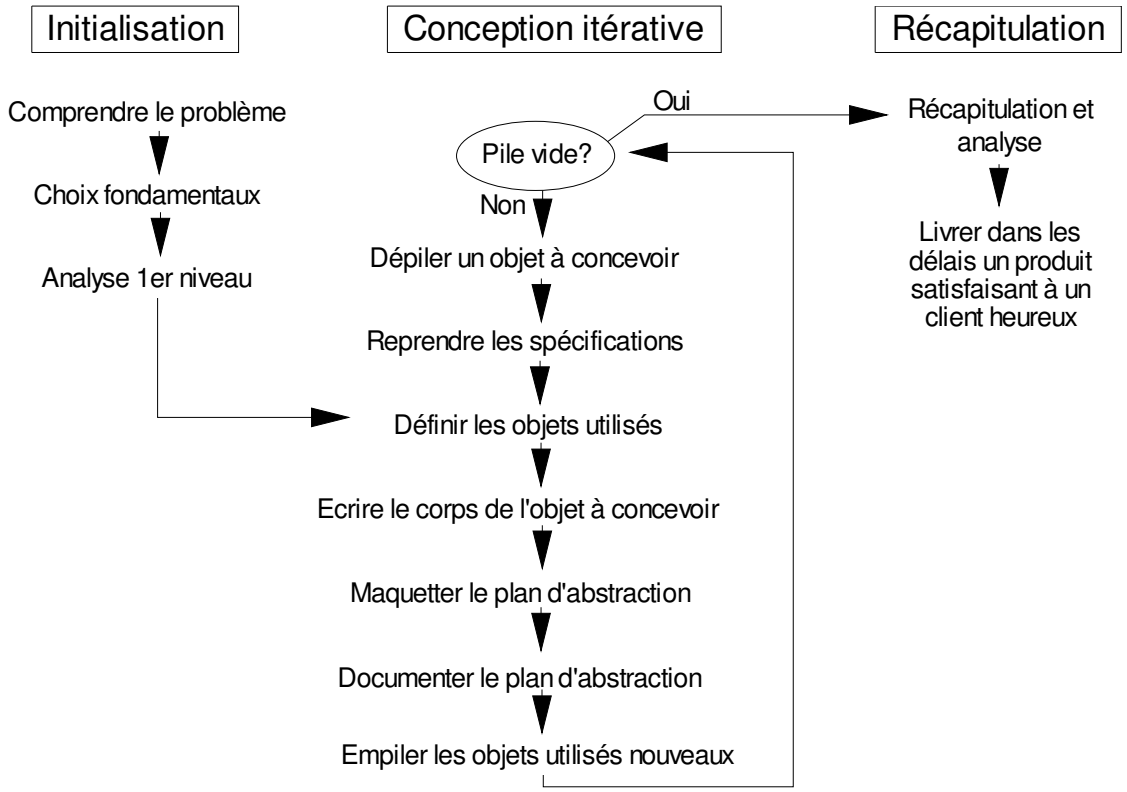


**Figure 39 :** Schéma des phases de la méthode

## 23.6 Critique de la méthode

Il n'existe pas de méthode universelle, mais seulement différentes formes de compromis, qui peuvent se révéler plus ou moins adaptés aux besoins. Que peut-on dire de la méthode que nous venons de présenter ?

C'est une méthode orientée objet par composition, qui autorise des hiérarchies de types au moment du choix des représentations, mais où l'héritage n'a qu'une faible importance. Elle s'applique bien aux projets manipulant de nombreuses entités du monde réel ; elle peut être moins adaptée à ceux essentiellement procéduraux, comme des compilateurs ou des modélisations mathématiques, pour lesquels des techniques de programmation structurée sont plus adaptées. Elle accorde la plus grande attention à la découpe en couches soigneusement isolées, grâce à la notion de plan d'abstraction. Ceci autorise une technique d'implémentation par démaquettage progressif, qui fait disparaître la phase d'intégration en tant que telle.

Le langage Ada est utilisé comme moyen d'expression des décisions de conception à toutes les phases du projet. Ceci peut surprendre en donnant l'impression de «coder» tout de suite. Le but est d'avoir un langage d'expression unique depuis la conception jusqu'à la réalisation ; cela n'est possible que grâce au très haut niveau d'abstraction fourni par Ada. L'utilisation de la méthode avec un langage de plus bas niveau serait difficile, voire dangereuse, car il ne serait pas possible d'exprimer des décisions de conception sans faire *en même temps* des décisions d'implémentation – ce que l'on veut éviter.

La méthode essaie de *limiter* la quantité de documentation, notamment en fournissant autant que possible du code autodocumenté. Ceci n'est possible que parce que la méthode ne vise que les projets de taille moyenne ; cette politique peut ne pas être applicable à des projets de taille plus importante. Enfin, la méthode ne formalise pas les contraintes temporelles au-delà de la possibilité de gérer un «budget temps». Elle n'est donc pas adaptée au développement de projets temps réel critiques.

En résumé, son domaine d'efficacité maximum est celui de la réalisation d'applications informatiques courantes, sans contraintes de certifications ni de performances trop exigeantes et de taille moyenne... ce qui représente tout de même une bonne part des développements logiciels.

## 23.7 Exercices

1. Comparer la méthode que nous venons d'exposer à HOOD et OMT en tenant compte des différences de domaines visés.
2. Rechercher les points de la méthode qui sont spécifiques d'Ada et envisager comment l'utiliser avec d'autres langages.

# 24

# Exemple d'utilisation de la méthode

Ce chapitre décrit un exemple d'utilisation de la méthode que nous venons de définir. Généralement, les exemples publiés dans les «bons ouvrages» proposent directement la solution parfaite, avec beaucoup d'explications montrant comment on l'a obtenue *naturellement*. En fait, la conception de logiciel, comme toute activité créatrice, comporte des tâtonnements et des erreurs. Nous avons essayé dans cet exemple de suivre la conception d'une application en n'occultant *pas* les erreurs initiales, mais au contraire en les montrant et en expliquant comment elles ont été identifiées et corrigées pour aboutir à la solution. De même, nous avons montré les hésitations que nous avons éprouvées lors de choix entre plusieurs solutions et essayé d'expliciter le processus qui nous a amené à nos décisions de conception. Il importe donc de lire le chapitre en entier avant de décider que l'auteur raconte n'importe quoi... En revanche, nous invitons le lecteur à réfléchir par lui-même aux solutions possibles à chaque étape avant de lire celles que nous proposons. Peut-être en trouvera-t-il qui ne figurent pas ici ; qu'il n'en déduise pas que les siennes sont mauvaises, mais simplement que cela montre la richesse des choix possibles et la nécessité de réfléchir avant de se précipiter sur la solution «évidente».

D'autre part, les contraintes propres à un livre nous empêchent de fournir tous les documents et de suivre précisément toutes les étapes qui sont nécessaires à un développement industriel. Que le développeur industriel n'en profite pas pour croire qu'ils sont moins nécessaires que le reste !

## 24.1 Phase initiale

### *24.1 .1 Comprendre le problème*

La célèbre société «L'Automatique de Distribution Alimentaire» (A.D.A.) nous a chargé de réaliser le logiciel de contrôle d'un distributeur automatique de boissons. L'appareil distribue du café (avec ou sans lait, avec ou sans sucre), du thé, des boissons froides... Il accepte toutes les pièces et rend la monnaie. Les différentes boissons n'ont pas forcément le même prix. Le logiciel doit permettre de changer aisément le tarif des boissons et être conçu de façon réutilisable pour éviter de trop gros changements en cas d'évolution des éléments matériels sous-jacents.

Nous notons qu'un tel cahier des charges est «ouvert» : tout le monde connaît ce genre d'appareil, aussi ne nous a-t-on fourni que peu de détails sur les contraintes. Bien que ceci nous laisse relativement libre, c'est par nature dangereux : la quantité de «non-dit culturel» est maximale et notre vue risque de ne pas correspondre à celle du client. Dans un projet réel, il faudrait prévoir une réunion avec le client à la fin de la phase de conception initiale, afin de vérifier que la structure adoptée correspond bien à celle attendue.

## *24.1 .2 Choix fondamentaux*

Bien entendu, un exemple livresque ne saurait avoir les mêmes contraintes qu'un projet industriel. Disons que le contexte de développement correspondrait aux choix fondamentaux suivants :

> Utilisation de la méthode de développement C-FOOD.
> Pas de composants logiciels préexistants et volonté du client d'obtenir la totalité des sources : nous devrons donc développer nous-même tout l'ensemble et non rechercher d'éventuels composants commerciaux.
> Le logiciel doit être robuste (aucun opérateur humain), sans cependant être critique du point de vue de la sécurité. En particulier, une politique d'erreurs par exceptions simples est suffisante.

## *24.1 .3 Analyse de premier niveau*

Une première discussion du problème aboutit à la formulation suivante :

> *Le logiciel doit piloter un **distributeur**[1] de **boissons** en fonction du **montant** introduit par le **consommateur** dans le **monnayeur**. Le consommateur choisit sa boisson en appuyant sur l'un des **boutons** de l'appareil. Chaque boisson a un **prix** et rien n'indique a priori que toutes les boissons doivent avoir le même prix.*

## *24.1 .4 Définir le plan d'abstraction*

### a) Identifier les objets utilisés

A partir de la définition informelle précédente, nous allons essayer d'identifier les principaux objets utilisés.

Le problème comporte à l'évidence une notion centrale qui est la *boisson*. Il convient cependant d'éviter de se précipiter vers une définition trop hâtive ; comme cet élément est très important, de nombreuses vues sont possibles et nous ne maîtrisons pas encore assez bien le problème pour la définir. Retenons simplement pour l'instant qu'il existe «quelque chose» appelé boisson. Nous devrons gérer de l'*argent,* qui est manifestement une grandeur d'un type numérique approprié. Le *consommateur* est évidemment un objet important du monde réel. Toutefois, il est totalement extérieur au système. Le consommateur n'est pas *géré* par le logiciel et il n'interagit pas non plus *directement* avec le logiciel. Ce n'est donc pas un objet intéressant *pour notre vue actuelle.*Le *monnayeur* se chargera de tout ce qui est lié à la gestion de l'argent : décompte du montant introduit et rendu de monnaie.

Nous avions remarqué le mot *bouton* dans la description informelle. Il n'apparaît pas nécessaire cependant d'en faire un objet pour l'instant, puisque les objets précédemment définis semblent suffisants. Le bouton est un objet de première importance *pour la vue du consommateur* (ce qui nous l'a fait mettre dans la définition informelle). Il ne s'agit que d'un détail de bas niveau *pour la vue du logiciel.* Pour s'en convaincre, il suffit de remarquer que l'on pourrait remplacer les boutons par un contacteur rotatif... ou des messages provenant d'un réseau local sans que cela change quoi que ce soit au logiciel *à ce niveau*. Pourtant, si ce terme figure dans l'énoncé, cela signifie qu'il cache une notion plus importante... Réfléchissons à la *nature* du problème. Le bouton est le moyen utilisé par le consommateur pour faire son *choix*. Le programme doit être informé du choix de boisson du consommateur, du prix correspondant et peut-être d'autres éléments non encore

[1] Les mots en caractères gras sont ceux qui peuvent être candidats au statut d'objets.

identifiés. Nous pouvons formaliser ceci en introduisant un objet *menu* qui sera chargé de gérer les choix du consommateur.

Nous allons maintenant revenir sur chacune des notions précédentes, en essayant de préciser un peu mieux leurs définitions, d'identifier leurs opérations et de tenter une première approximation Ada de leurs spécifications.

*Argent*

L'argent est un type numérique comportant des francs et des centimes. Les seules opérations nécessaires *a priori* sont l'addition et la soustraction. Nous en faisons cependant un paquetage autonome, car cela correspond à une notion indépendante du reste du problème. Un type décimal est particulièrement adapté et comporte les opérations nécessaires. Le type `Argent` est donc un objet (ou plutôt un type) terminal du projet. Quelles limites devons-nous utiliser pour ce type ? Actuellement, la plus grosse pièce courante en France est la pièce de 20F, mais il ne paraît pas impossible de voir un jour apparaître des pièces de 50 ou même 100F. La prudence nous recommande donc de prévoir cinq chiffres significatifs (avec les centimes).

```
package Définition_Argent is
   type Argent is delta 0.01 digits 5;
end Définition_Argent;
```

*Boisson*

Tout d'abord, nous nous apercevons qu'à ce point de l'analyse, rien n'est spécifique des produits liquides ; nous pourrions aussi bien distribuer des cacahuètes... Nous allons donc remplacer le terme boisson par *produit*.

Qu'est-ce qui caractérise un produit ? Il lui est associé un *prix* et chaque produit a une façon bien particulière d'être fourni. Ce peut être élémentaire (cas du paquet de cacahuètes : il suffit d'ouvrir un relais pour laisser tomber le paquet), mais d'autres doivent être réellement fabriqués ; pour un café par exemple, il faut laisser tomber de la poudre de café (et éventuellement de la poudre de lait), du sucre, une petite cuillère, une quantité plus ou moins grande d'eau... Deux solutions sont possibles à ce niveau :

- Nous considérons que la notion de produit est une classe générale, munie d'un attribut «prix» et d'une méthode de fabrication propre à chaque élément de la classe. Comme aucun produit réel ne peut être un «produit» sans être quelque chose de plus précis, cette classe doit être abstraite. C'est une solution typiquement «classification» : nous créerons les produits réels grâce au mécanisme d'héritage à partir de la classe `Produit`. Nous exprimerions cette solution de la façon suivante :

  ```
  with Définition_Argent; use Définition_Argent;
  package Classe_Produit is
     type Produit is abstract tagged record
       Prix : Argent;
     end record;

     procedure Fabriquer (Le_Produit: Produit) is abstract;
  end Classe_Produit;
  ```

- Nous considérons que le produit n'est qu'un identifiant auquel sont associés de façon extérieure un *tarif* (qui fournit le prix de chaque produit) et une *recette* (qui définit comment fabriquer le produit). Une telle approche ne fait pas appel aux notions de la classification ni à l'héritage.

Quelle est la meilleure solution ? Ce n'est pas évident. Remarquons que chaque solution conduit à une factorisation, mais pas à la même : dans la première, tous les aspects liés à une boisson (prix, fabrication) sont regroupés, alors que dans la seconde on rassemble d'une part tout ce qui est tarif, d'autre part tout ce qui est fabrication.

Il ne semble pas que la gestion du prix soit un facteur déterminant : l'implémentation sera triviale dans tous les cas de figure ; nous allons donc réfléchir au problème de la fabrication. La première solution nous permet d'avoir simultanément des produits dont la méthode de fabrication est radicalement différente : elle n'implique aucune ressemblance entre les différentes méthodes `Fabriquer`, alors que la seconde implique que la fabrication de tous les produits s'exprime au moyen du même genre de «recette». Un avantage de la première solution est donc qu'une évolution du logiciel conduisant à distribuer un produit radicalement différent ne perturbera pas les produits existants. Cependant, un changement des produits distribués nécessiterait une reprogrammation : si nous nous apercevons que personne n'achète le jus de citron et que nous décidons de vendre de la soupe à la place, il faudra faire une nouvelle version du logiciel. Si au contraire nous adoptons la seconde solution, il suffit de changer quelques tables, en modifiant une PROM par exemple, ou même simplement en branchant un terminal pour entrer la nouvelle recette. De même, un changement de tarif est plus facile si tous les prix sont rassemblés, plutôt que dispersés dans chacun des produits. D'autre part, il paraît peu probable que la souplesse d'évolution offerte par la première solution soit nécessaire : un changement radical de méthode de fabrication signifierait de nouveaux dispositifs matériels, donc quasiment la construction d'un appareil entièrement nouveau. Ce n'est pas susceptible de se produire souvent, comparé au fait de changer les produits distribués. Enfin, quiconque a déjà vu un distributeur ouvert a certainement remarqué qu'il comporte une entité, l'unité de fabrication, qui confectionne effectivement les produits, bien séparée du monnayeur qui gère l'argent. Dans une optique orientée objet, il paraît préférable que la découpe du logiciel corresponde aux éléments matériels.

Nous choisirons donc la seconde solution, mais les raisons qui nous y ont amené devront être soigneusement consignées dans le document de maintenance. Nous notons que nous identifions alors trois nouveaux objets : le *tarif*, la *recette* et l'*unité de fabrication*. La notion de produit devient alors extrêmement ténue : elle ne sert qu'à faire la liaison entre un bouton sur lequel on appuie, un prix et une recette. Nous pouvons la représenter par n'importe quel type discret. La tendance naturelle en Ada serait d'utiliser un type énumératif :

```
type Produit is (Café, Thé, Orange, Soda);
```

Cette solution a nécessairement un aspect limitatif : les boissons doivent être choisies à la compilation et tout changement nécessite une recompilation de tout le système... ce que nous cherchons précisément à éviter. Revenons un peu sur ce qu'est un produit pour la vue que nous en avons maintenant. C'est finalement une notion qui nous permet de faire la liaison entre un bouton poussé par un utilisateur et une recette préparée par l'unité de fabrication. Le «parfum» choisi est sans importance pour nous ! Ce que nous appelons `Produit` n'est qu'un numéro de bouton sur le panneau de contrôle. A un bouton est associée une recette (et un prix) et la notion de `Produit` devient un simple identifiant neutre. Type énumératif, type entier, que vaut-il mieux choisir ? Difficile à dire à ce niveau, tout au plus pouvons-nous nous dire qu'il doit s'agir d'un type discret. Comme nous n'avons pas vraiment de critère de choix, mais que les raffinements ultérieurs pourront peut-être nous guider, il suffit d'exprimer l'état de nos réflexions en déclarant :

```
with ADPT;
package Définition_Produit is
   type Produit is new ADPT.Type_Discret;
end Définition_Produit;
```

Nous n'avons *a priori* besoin d'aucune opération sur les produits (on ne va pas, par exemple, additionner deux produits). Comme pour l'argent, nous en faisons un paquetage séparé, car même s'il ne s'agit après tout que d'une simple déclaration de type, la notion qu'elle recouvre est importante.

*Unité de fabrication*

L'unité de fabrication est une sorte de machine stupide, à qui l'on dit de confectionner un produit... et qui le fait. Elle n'a pas à se préoccuper du qui ni du quoi (comme de savoir si le

consommateur a payé). Inversement, elle doit nous permettre d'ignorer totalement (à ce niveau) *comment* on fait pour fabriquer le produit. Du point de vue des spécifications, ceci s'exprime comme :

```
with Définition_Produit; use Définition_Produit;
package Unité_Fabrication is

   procedure Confectionner (Le_produit : Produit);

end Unité_Fabrication;
```

Avec cette vue il ne peut s'agir que d'une machine abstraite. Doit-elle être active ? Il est clair que le dispositif physique peut très bien évoluer en parallèle. Mais en pratique, pendant que la machine confectionne un produit, elle reste totalement bloquée et n'accepte aucune autre opération. Dans ces conditions, mieux vaut la laisser passive.

*Monnayeur*

Le monnayeur gère tout ce qui est lié à l'argent. Il doit pouvoir nous indiquer le montant entré par le consommateur. Il doit également pouvoir rendre la monnaie sur le prix d'un produit. Notons que le monnayeur doit être actif en permanence, car le consommateur peut introduire des pièces à n'importe quel moment[1]. Il s'agira donc vraisemblablement d'une machine abstraite active.

```
with Définition_Argent;  use Définition_Argent;
with Définition_Produit; use Définition_Produit;
package Monnayeur is
   function  Montant_Introduit return Argent;
   procedure Rendre_Monnaie (Du_produit: Produit);
end Monnayeur;
```

*Menu*

Le menu nous indique le produit choisi par le consommateur. Il gère les choix de l'utilisateur et ne renvoie qu'un choix *valide*. Ceci s'exprime comme :

```
with Définition_Produit; use Définition_Produit;
package Menu is
   function Choix_Consommateur return Produit;
end Menu;
```

*Tarif*

Le tarif nous indique le prix de chaque produit :

```
with Définition_Produit; use Définition_Produit;
with Définition_Argent;  use Définition_Argent;
package Tarif is
   function Le_Prix (De : Produit) return Argent;
end Tarif;
```

Ne peut-on en faire directement une table ? Certainement, mais alors nous sommes lié au choix final du type de `Produit`. Si pour une raison ou une autre nous décidons que `Produit` n'est *pas* un type discret (ce pourrait être un type accès par exemple), il nous sera beaucoup plus facile d'évoluer si nous gardons le tarif sous la forme d'une fonction. Inversement, si nous *implémentons* la fonction au moyen d'une table, il suffira de mettre la fonction «en ligne» pour ne payer aucune pénalité d'efficacité[2]. Décider à ce niveau d'utiliser une table serait un exemple typique de surspécification – spécifier à partir d'une implémentation particulière, au lieu d'abstraire le besoin réel.

---

[1] Dans un système bien fait, le consommateur doit pouvoir introduire une pièce d'1F., puis appuyer sur «Café» et enfin introduire sa deuxième pièce. Il est très difficile d'obtenir ce comportement sans parallélisme... d'ailleurs certains distributeurs ne l'autorisent pas.

[2] Cette dernière remarque est pour le principe. Il est très peu vraisemblable que nous ayons un problème d'efficacité à ce niveau.

### b) Ecrire le corps de l'objet à concevoir

Dans la phase initiale, l'objet à concevoir est simplement le programme principal. Comme tout programme principal Ada, il s'agit d'une simple procédure sans paramètres.

```
with Définition_Produit, Définition_Argent, Menu,
     Monnayeur,          Unité_Fabrication, Tarif;
use  Définition_Produit, Définition_Argent, Menu,
     Monnayeur,          Unité_Fabrication, Tarif;
procedure Principal is
   Choisi : Produit;
begin
   loop
     Choisi := Choix_consommateur;
     while Montant_Introduit < Le_Prix (De => Choisi) loop
       delay 0.1;
     end loop;
     Rendre_Monnaie (Du_produit => Choisi);
     Confectionner (Le_Produit => Choisi);
   end loop;
end Principal;
```

### c) Contester

Maintenant que nous avons une première structure sur laquelle appuyer notre pensée, il faut regarder de plus près et inspecter si nous n'avons pas omis des détails, manqué de précision dans les définitions, ou oublié une fonctionnalité. En particulier, nous n'avons pas envisagé les cas exceptionnels : épuisement des produits, non-disponibilité de monnaie, annulation de la demande par l'utilisateur. De plus, le programme principal effectue une attente active, ce qui est toujours gênant. Nous avons eu la prudence d'introduire un **`delay`** pour éviter des blocages sur un système monoprocesseur, mais une solution sans attente active serait préférable.

*Cas du produit épuisé*

Nous avons dit que le menu doit toujours renvoyer un choix valide, c'est-à-dire qu'il ne doit pas prendre en compte un produit épuisé. Au premier niveau, il est commode de le spécifier ainsi, mais il faut quand même se poser la question de la faisabilité de cette spécification. Autrement dit : comment le menu peut-il savoir quels produits peuvent être fabriqués ? Ou bien le menu peut interroger l'unité de fabrication, ou bien il faut que «quelqu'un» prévienne le menu qu'un produit n'est plus disponible. La première solution introduit un couplage supplémentaire entre des unités qui n'ont aucune raison de se connaître, aussi vaudrait-il mieux l'éviter. La seconde implique de déterminer *qui* doit prévenir le menu. Ce ne peut être que l'unité de fabrication (mais on introduit de nouveau un couplage avec le menu) ou le programme principal. Comme celui-ci connaît de toute façon le menu et l'unité de fabrication, on n'introduirait pas de couplage supplémentaire.

Faut-il introduire une fonctionnalité supplémentaire pour demander à l'unité de fabrication si un produit peut être confectionné ? La détermination risque d'être très difficile : si l'on peut vérifier que les différents ingrédients nécessaires sont disponibles, ce n'est quand même pas une garantie que le produit peut être réalisé. Un réservoir de poudre peut être en panne, un robinet collé... Nous devons donc de toute façon nous attendre qu'un produit ne puisse être réalisé : il faut prévoir une exception dans l'unité de fabrication.

```
with Définition_Produit; use Définition_Produit;
package Unité_Fabrication is
   procedure Confectionner (Le_produit : Produit);

   Confection_Impossible : exception;
end Unité_Fabrication;
```

Du coup, cela résout le problème précédent : le programme principal tente la fabrication ; en cas d'échec, il invalide le produit. Nous ajoutons donc une fonctionnalité au menu pour invalider le produit ; par symétrie, il nous faut également une fonctionnalité pour le revalider :

```
with Définition_Produit; use Définition_Produit;
package Menu is
   function Choix_Consommateur return Produit;

   procedure Invalider (Le_produit : Produit);
   procedure Revalider (Le_produit : Produit);
end Menu;
```

*Cas de l'annulation*

A tout moment (tout au moins tant que la distribution n'a pas commencé), le consommateur peut annuler sa commande. Nous n'avons encore rien prévu pour le gérer. Remarquons que ce cas n'était pas prévu au cahier des charges ; dans une optique industrielle, il faudrait prévenir le client afin de vérifier que notre interprétation du comportement souhaité correspond bien à ses désirs.

Une première solution serait de considérer l'annulation comme un produit spécial (et de prix nul). Si l'utilisateur appuie sur le bouton d'annulation avant d'avoir fait son choix, le consommateur «recevra» le produit `Annulation`. `Rendre_Monnaie` remboursera le consommateur (puisque le prix est 0.00) et l'unité de fabrication ignorera simplement le produit. Cette solution peut paraître astucieuse,[1] mais elle ne permet pas de traiter simplement le cas de l'annulation *après* que le consommateur a fait son choix[2]. On pourrait également lever une exception au lieu de renvoyer une valeur spéciale, mais cela ne résoudrait pas le problème. Il faut prendre du recul, oublier l'informatique et revoir la question au niveau du domaine de problème.

Remarquons qu'il existe un «point de non-retour» dans les actions du consommateur : lorsqu'il a choisi une boisson et que le montant introduit est suffisant, la distribution commence et il n'est plus possible de l'annuler. L'annulation peut intervenir à n'importe quel moment *avant* ce point et doit immédiatement stopper tout traitement en cours pour ramener le distributeur à l'état initial (après avoir rendu d'éventuelles pièces déjà introduites). On pourrait être tenté de modéliser ce comportement par une interruption, mais cela nous ferait dépendre du matériel dès ce niveau. De plus, comment interrompre le reste des traitements une fois l'interruption matérielle traitée ? Certes on peut rajouter des booléens testés périodiquement, mais tout ceci commence à faire terriblement «bricolé». Heureusement, Ada nous procure une fonctionnalité correspondant de près au comportement souhaité : le transfert de contrôle asynchrone. Nous représenterons le bouton d'annulation comme une entrée et la séquence annulable comme la suite d'instructions avortables. Utiliserons-nous une entrée de tâche ou de type protégé ? Nous n'avons aucune raison d'introduire de tâche à ce niveau, nous prendrons donc plutôt un objet protégé, comme la barrière que nous avons donnée en exemple dans le préambule du livre. Cette décision pourra être remise en cause ultérieurement.

Qui doit gérer cet objet ? *A priori*, il correspond à un bouton de la face avant du distributeur, donc il devrait faire partie du menu. D'autre part, sur les distributeurs réels, le bouton d'annulation se situe plutôt du côté du monnayeur. Enfin nous pouvons toujours en faire une entité indépendante... Mettons-le pour l'instant dans le menu, mais ne soyons pas étonné si la suite de l'analyse vient à le déplacer. Cela nous donne :

```
with Définition_Produit; use Définition_Produit;
package Menu is
   function Choix_Consommateur return Produit;

   procedure Invalider (Le_produit : Produit);
   procedure Revalider (Le_produit : Produit);
```

[1] Ce qui n'est pas forcément un gage de qualité.

[2] Notons une décision implicite : lorsque le client a fait son choix, il ne peut plus changer d'idée en appuyant sur un autre bouton (sauf annulation). Donc, le menu est bloqué après avoir reçu un choix valide. Donc il faut prévoir un moyen de le débloquer... Nous sommes en train d'étudier un autre problème, mais il faut noter celui-ci pour ne pas l'oublier par la suite.

```
   protected Annulation is
      entry Signalement;
      procedure Demande;
   private
      Ouverte : Boolean := False;
   end  Annulation;
end Menu;
```

Cette notion de point de non-retour est également importante pour le monnayeur : si l'utilisateur continue d'introduire des pièces, elles doivent être ignorées. On va donc avoir un changement d'état dans le monnayeur : dans un premier temps, on attend d'avoir atteint au moins le montant du produit. Une fois ce montant atteint, on fait passer le monnayeur dans un état «bloqué» où il rejette systématiquement toute pièce introduite. Ce comportement peut être obtenu par logiciel, mais au fond il vaut mieux le faire par matériel : il suffit de piloter un clapet qui laisse retomber directement les pièces introduites dans le gobelet de rendu de la monnaie. Ce clapet est au repos dans l'état «bloqué», de façon à ne pas voler l'utilisateur en cas de panne de courant. Il faut évidemment introduire un moyen de faire repasser le monnayeur dans l'état «actif». De plus, il faut pouvoir demander au monnayeur de rendre toutes les sommes introduites. Notre spécification va donc devenir :

```
with Définition_Argent;  use Définition_Argent;
with Définition_Produit; use Définition_Produit;
package Monnayeur is
   procedure Activer;
   procedure Bloquer;

   function  Montant_Introduit return Argent;
   procedure Rendre_Monnaie (Du_produit : Produit);
   procedure Rembourser;
end Monnayeur;
```

*Cas de la monnaie épuisée*

Qu'appelle-t-on monnaie épuisée ? Les distributeurs qui rendent la monnaie disposent tous du voyant correspondant, mais sa signification n'est pas du tout évidente. En effet, le monnayeur peut ne pas être capable de garantir le rendu de monnaie dans *tous* les cas, tout en étant capable de le faire dans certains cas... Il faut séparer le problème du voyant de celui du rendu effectif. En dessous d'un certain nombre de pièces, le monnayeur allume automatiquement le voyant. Ceci est interne au monnayeur et ne nous concerne pas pour l'instant. Si maintenant le monnayeur est dans l'incapacité de rendre la monnaie qui lui a été demandée, il doit lever une exception (appelons-la `Rendu_Impossible`). Cette exception peut être rattrapée par le logiciel appelant qui prendra les mesures nécessaires, comme d'annuler la demande et de demander de rendre la monnaie avec un montant égal à la somme entrée... ce qui est évidemment toujours possible. La spécification du monnayeur devient :

```
with Définition_Argent;  use Définition_Argent;
with Définition_Produit; use Définition_Produit;
package Monnayeur is
   procedure Activer;
   procedure Bloquer;

   function  Montant_Introduit return Argent;
   procedure Rendre_Monnaie (Du_produit : Produit);
   procedure Rembourser;

   Rendu_Impossible : exception;
end Monnayeur;
```

Par analogie, nous allons rajouter une procédure `Activer` au `Menu` pour autoriser les choix de l'utilisateur, comme nous l'avions noté précédemment :

```
with Définition_Produit; use Définition_Produit;
package Menu is
   procedure Activer;
   function Choix_Consommateur return Produit;

   procedure Invalider (Le_produit : Produit);
   procedure Revalider (Le_produit : Produit);

   protected Annulation is
      entry Signalement;
      procedure Demande;
   private
      Ouverte : Boolean := False;
   end  Annulation;
end Menu;
```

*Autres cas exceptionnels*

Nous ne voyons pas *a priori* d'autres cas d'exception ; mais si un problème imprévu se produisait, il faut en tout état de cause rendre son argent à l'utilisateur. Il convient donc de prévoir un traitement d'exception «rattrape-tout».

*Le problème de l'attente active*

On appelle «attentes actives» les boucles qui ne font qu'attendre qu'un événement se produise, comme dans le cas de la boucle d'attente sur le montant introduit. Une telle boucle est toujours gênante, car elle accapare l'unité centrale sans rien faire d'utile. Il faut toujours mettre un **`delay`** dans ce cas, autrement cette boucle pourrait empêcher d'autres parties du programme de s'exécuter... y compris celles chargées de changer la condition de boucle. Mais la meilleure solution consiste à les éviter. Pour cela, il faut se poser la question suivante : quel est le besoin de plus haut niveau qui nous a conduit à cette boucle ? En l'occurrence, c'est d'attendre qu'un certain montant soit introduit. *Attendre le prix d'un produit* pourrait parfaitement être un service rendu par le monnayeur, faisant ainsi disparaître l'attente active, au moins à ce niveau d'abstraction. L'implémentation la fera peut-être réapparaître, à moins que nous ne trouvions une autre solution. Au moins, en reportant ce problème vers les couches plus profondes, ouvrons-nous la possibilité d'autres solutions. La spécification finale du monnayeur devient donc :

```
with Définition_Argent;  use Définition_Argent;
with Définition_Produit; use Définition_Produit;
package Monnayeur is
   procedure Activer;
   procedure Bloquer;

   procedure Attendre (Pour_Produit : Produit);

   function  Montant_Introduit return Argent;
   procedure Rendre_Monnaie (Du_produit: Produit);
   procedure Rembourser;

   Rendu_Impossible : exception;
end Monnayeur;
```

### d) Nouvelle version du programme principal

Nous pouvons maintenant récrire le programme principal en fonction des modifications que nous avons apportées :

```
with Définition_Produit, Définition_Argent, Menu,
     Monnayeur,          Unité_Fabrication, Tarif;
use  Définition_Produit, Définition_Argent, Menu,
     Monnayeur,          Unité_Fabrication, Tarif;
procedure Principal is
   Choisi : Produit;
   Annulé : Boolean;
begin
   loop
     Menu.Activer;
     Monnayeur.Activer;

     select
       Menu.Annulation.Signalement;
       Annulé := True;
     then abort
       Choisi := Choix_consommateur;
       Attendre (Pour_Produit => Choisi);
       Annulé := False;
     end select;

     Monnayeur.Bloquer;
     if Annulé then
       Rembourser;
     else
       begin
         Rendre_Monnaie (Du_Produit => Choisi);
         Confectionner (Le_Produit => Choisi);
       exception
         when Rendu_Impossible =>
           Rembourser;
         when Confection_Impossible =>
           Invalider (Le_Produit => Choisi);
           Rembourser;
       end;
     end if;
   end loop;

exception
   when others =>
     Monnayeur.Bloquer;
     Rembourser;
end Principal;
```

## *24.1 .5 Tester le plan d'abstraction*

Nous avons maintenant un schéma général qui paraît satisfaisant. Les spécifications que nous venons d'établir peuvent être compilées pour vérifier qu'elles sont cohérentes. Nous pouvons vérifier dynamiquement le plan d'abstraction en fournissant des corps maquettes tels que :

```
with ADPT;
package body Unité_Fabrication is
   procedure Confectionner (Le_produit : Produit) is
     use ADPT;
   begin
     ADPT_Action ("Confection du produit " &
                  Produit'IMAGE (Le_produit));
   end Confectionner;
end Unité_Fabrication;
```

On trouvera en annexe les corps maquettes des autres paquetages. On obtient ainsi une première version exécutable modélisant le comportement du programme à haut niveau, sans avoir encore décidé des modalités de réalisation des couches inférieures.

### *24.1 .6 Documenter le plan d'abstraction*

Sans fournir une documentation aussi complète que l'exigerait un produit industriel, nous allons résumer les caractéristiques de la solution obtenue.

Le distributeur est constitué de quatre objets principaux : l'*unité de fabrication* qui s'occupe de la fabrication effective des *produits*, le *monnayeur* qui gère tout ce qui est lié à la notion d'*argent*, le *menu* qui s'occupe de tout ce qui est dialogue avec l'utilisateur et le *tarif* qui relie les *produits* à leur *prix*. La figure 40 schématise cette architecture. Remarquer que nous avons représenté les produits et l'argent sous forme de bus logiciels : la structure réelle est ainsi beaucoup plus apparente. On remarque également qu'il n'existe aucune dépendance entre les quatre modules principaux ; ceci correspond bien au fait qu'ils constituent des sous-systèmes indépendants, dont il serait aisé de confier le développement à des équipes séparées. Cette indépendance provient évidemment du fait que ces objets correspondent directement à des sous-systèmes physiques de l'appareil, qui sont eux-mêmes indépendants.

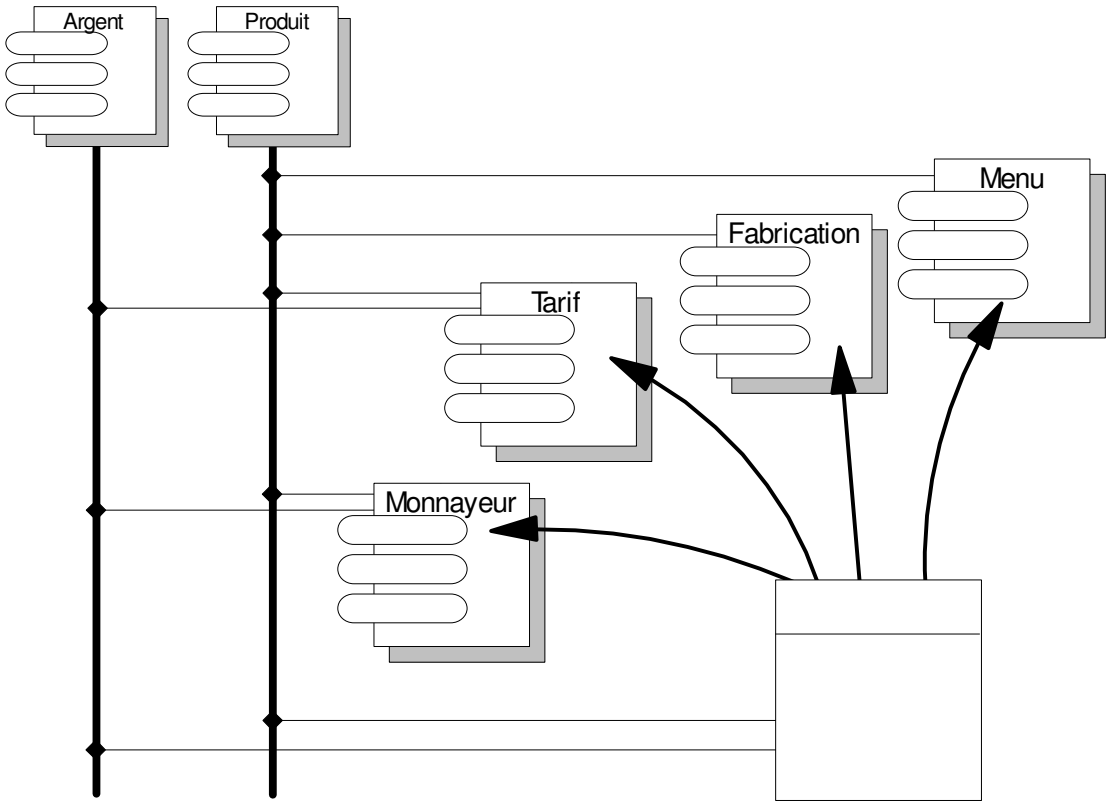


**Figure 40 :** Architecture du distributeur de boissons

## 24.2 Deuxième itération

Arrivé à ce stade, nous avons une définition satisfaisante, vérifiée, de tous les objets au premier niveau d'abstraction ; il faut maintenant les reprendre un à un et les implémenter effectivement, c'est-à-dire examiner de façon détaillée comment on peut les réaliser. Les objets que nous avons empilés (c'est-à-dire ceux pour lesquels nous ne disposons pas encore de l'implémentation définitive) sont :

Le produit
Le monnayeur
Le menu
L'unité de fabrication

- Le tarif

Notre but n'étant pas d'écrire ici l'intégralité du logiciel, nous n'allons pas détailler l'implémentation de tous ces objets jusqu'au niveau final, d'autant que certains sont liés au matériel. Nous allons juste en reprendre quelques-uns, pour montrer des exemples (terminaux et non terminaux) du processus d'itération de la méthode.

## *24.2 .1 Le produit*

Il est temps maintenant de décider de la représentation effective du `Produit`. Nous avons déjà vu que ce devait être un type discret. Le premier niveau de maquettage n'a fait apparaître aucun besoin nouveau : il s'agit d'un simple identifiant totalement neutre ; nous avons vu qu'à la limite, on pouvait assimiler le produit à un numéro de bouton. Un type énumératif serait parfaitement acceptable, puisque nous n'avons besoin d'aucune opération arithmétique, mais présente cependant un inconvénient : il est plus difficile à faire évoluer, par exemple si l'on crée un nouvel appareil disposant d'un nombre supérieur de boutons. Nous choisirons donc de le représenter par un type entier. Les bornes peuvent être données par une constante du paquetage, ou dans un paquetage global définissant la configuration matérielle, en particulier le nombre de boutons (et donc le nombre de produits pouvant être distribués).

```
package Définition_Produit is
   Max_produit : constant := 10;
   type Produit is range 1..Max_produit;
end Définition_Produit;
```

## *24.2 .2 Le monnayeur*

### a) Reprendre les spécifications

Nous sommes maintenant chargé d'étudier le problème du monnayeur. A ce stade, nous *oublions* tout ce que nous avons pu dire du distributeur en général pour étudier ce qu'*est* un monnayeur. Il apparaît immédiatement que celui que nous avons défini est trop spécifique. `Rendre_Monnaie` a comme paramètre un `Produit` ; ceci induit un couplage supplémentaire totalement inutile. De plus, rien ne dit que le monnayeur fasse partie d'un appareil où la notion de produit ait un sens ; ce pourrait être par exemple une machine à sous (type «bandit manchot»), auquel cas il peut arriver que l'on soit amené[1] à «rendre» un montant supérieur à celui introduit ! Il est beaucoup plus logique que `Rendre_Monnaie` et `Attendre` aient un argument de type `Argent` : le montant à rendre ou à attendre. Le monnayeur n'a plus à savoir pour quelles raisons on attend ou on rend cet argent. Cette modification permet de faire disparaître la procédure `Rembourser`, qui introduisait d'ailleurs un couplage temporel, le montant à rendre dépendant du montant introduit. D'autres question de sémantique délicate auraient pu se poser, comme celle de savoir si l'on avait le droit de rembourser lorsque le monnayeur n'était pas dans l'état bloqué... Notre spécification devient :

```
with Définition_Argent; use Définition_Argent;
package Monnayeur is
   procedure Activer;
   procedure Bloquer;

   procedure Attendre (Montant : Argent);

   function  Montant_Introduit return Argent;
   procedure Rendre_Monnaie (Montant : Argent);

   Rendu_Impossible : exception;
end Monnayeur;
```

[1] Certes rarement !

Bien sûr, il faut immédiatement prévenir le niveau supérieur que nous souhaitons modifier les spécifications. Le programme principal va devenir :

```
with Définition_Produit, Définition_Argent, Menu,
     Monnayeur,          Unité_Fabrication, Tarif;
use  Définition_Produit, Définition_Argent, Menu,
     Monnayeur,          Unité_Fabrication, Tarif;
procedure Principal is
   Choisi : Produit;
   Annulé : Boolean;

begin
   loop
     Menu.Activer;
     Monnayeur.Activer;

     select
       Menu.Annulation.Signalement;
       Annulé := True;
     then abort
       Choisi := Choix_consommateur;
       Attendre (Montant => Le_Prix (De => Choisi));
       Annulé := False;
     end select;

     Monnayeur.Bloquer;
     if Annulé then
       Rendre_Monnaie (Montant => Montant_Introduit);
     else
       begin
         Rendre_Monnaie (Montant =>
              Montant_Introduit - Le_Prix (De => Choisi));
         Confectionner (Le_Produit => Choisi);
       exception
         when Rendu_Impossible =>
           Rendre_Monnaie (Montant => Montant_Introduit);
         when Confection_Impossible =>
           Invalider (Le_Produit => Choisi);
           Rendre_Monnaie (Montant => Montant_Introduit);
       end;
     end if;
   end loop;

exception
   when others =>
     Monnayeur.Bloquer;
     Rendre_Monnaie (Montant => Montant_Introduit);
end Principal;
```

Un logiciel à caractère temps réel comme celui-ci doit fonctionner dans *tous les cas*, même ceux qui sont hautement improbables. Nous devons vérifier qu'aucune condition de course ne peut se produire suite à notre modification : en effet l'instruction

```
Rendre_Monnaie (Montant_Introduit-Le_Prix (De => Choisi));
```

n'est pas atomique ; si l'utilisateur introduit une pièce entre le moment où la valeur de la monnaie à rendre est calculée et le moment où la procédure est appelée, il y a un risque de rendre un montant incorrect. C'est pourquoi le `Montant_Introduit` est calculé *après* avoir bloqué le monnayeur ; si l'utilisateur continue d'introduire des pièces, elles retomberont directement dans la sébile sans être encaissées et ne gêneront donc pas le programme. Pensons également qu'une pièce peut être introduite après que nous sommes revenu de la procédure `Attendre` ; ce n'est pas gênant, car elle sera comptabilisée normalement et nous en tiendrons donc compte au niveau de `Rendre_Monnaie`.

Une autre condition de course risque de se produire au niveau de l'instruction **`select`** : que se passe-t-il si l'utilisateur appuie sur le bouton d'annulation juste après que nous avons mis la variable `Annulé` à `False` ? Si nous sommes sorti de l'instruction, nous ne nous en apercevrons même pas : l'utilisateur a pressé le bouton trop tard. Si nous ne sommes pas encore sorti du **`select`**, la partie

avant `then abort` sera exécutée, y compris la remise à `True` de la variable `Annulé`. La fourniture du produit sera ou bien totalement annulée, ou pas du tout, mais aucun cas intermédiaire ne peut se produire.

Bien entendu, nous devons immédiatement ajuster le corps maquette de `Monnayeur` pour refléter ces modifications (on le trouvera en annexe), puis recompiler et vérifier la nouvelle maquette. De cette façon, nous nous assurons qu'à tout moment le programme reste stable et vérifié.

### b) Définir le plan d'abstraction

A ce niveau d'abstraction, le monnayeur est chargé de faire la liaison entre la vue de haut niveau dont ont besoin les autres modules et les dispositifs physiques. Les objets que nous trouverons à ce niveau correspondront aux relais, commandes, etc., des dispositifs physiques. Nous ne les verrons que sous forme abstraite : leur implémentation dépendra directement du matériel.

*Identifier les objets utilisés*

Nous voyons tout de suite que le monnayeur doit comporter (au moins) deux parties principales : une tâche cachée, active en permanence, chargée de comptabiliser les pièces qui tombent et de gérer l'afficheur du montant introduit et un ensemble de services exportés.

Les services rendus à l'extérieur nécessitent la présence d'un *compteur* pour comptabiliser l'argent introduit ; nous avons déjà vu que nous devions à ce niveau piloter un *clapet* gouvernant la chute des pièces vers le compteur ou vers la sébile ; enfin les pièces doivent tomber dans des *réservoirs de pièces* que nous devrons commander pour le rendu de monnaie.

Le *compteur* doit être un objet actif, car il doit «voir passer» des pièces à tout moment. Il se décompose donc en un compteur proprement dit et une tâche de surveillance. Nous avons alors deux possibilités : avoir le compteur à l'intérieur de la tâche (la tâche *est* en quelque sorte le compteur lui-même), ou voir le compteur comme un objet externe à la tâche, celle-ci n'étant chargée que de la surveillance des pièces et de la mise à jour du compteur.

En Ada 83, nous aurions certainement adopté la première solution : en effet, le compteur doit être protégé contre des accès concurrents ; on peut interroger son montant à tout moment, y compris quand la tâche est en train de le modifier. Comme le seul moyen de protéger une variable était de la mettre dans une tâche, il était tout indiqué d'utiliser la tâche que nous avons sous la main. Sinon, il aurait fallu rajouter une tâche spéciale ; c'était faisable, mais non nécessaire (puisque nous envisagions déjà, pour des raisons structurelles, d'inclure le compteur dans la tâche de toute façon). La multiplication abusive des tâches est un travers dans lequel tombent des programmeurs Ada, enthousiasmés au-delà du raisonnable par ces nouvelles possibilités. Les tâches sont une bonne chose, mais il ne faut pas abuser des bonnes choses !

Ada 95 offre de nouvelles possibilités, grâce aux types protégés (ce qui ne signifie pas non plus qu'il faille se ruer dessus). Nous n'avons plus de raison *technique d'implémentation* pour faire ce choix : c'est la logique qui doit nous guider. En l'occurrence, nous appliquerons le principe suivant lequel chaque module doit faire une chose et une seule ; nous préférerons donc séparer le compteur de la tâche de surveillance, au nom du découplage. Noter également que nous limitons ainsi la profondeur d'imbrication des concepts. Enfin, la notion d'entrée de type protégé, avec sa garde conditionnelle, est vraisemblablement un bon moyen d'implémenter `Attendre` sans boucle active. Nous prenons donc la décision de conception d'utiliser un type protégé, mais nous nous rappelons (et nous notons dans le cahier des décisions de conception) qu'un autre choix était possible à ce stade.

Nous disposons donc d'un objet que nous appellerons le *compteur*, protégé contre des accès simultanés. Les opérations nécessaires sont de lui ajouter une valeur, de connaître la valeur courante, de le remettre à 0 et d'attendre qu'un certain montant soit introduit. Ceci s'exprime naturellement comme :

```
protected Compteur is
   procedure Ajouter (Montant : Argent);
   function  Valeur_Courante return Argent;
   procedure Remise_A_Zéro;
   entry Attendre (Montant : Argent);
end Compteur;
```

Les procédures `Activer` et `Bloquer` doivent piloter le *clapet*. Un clapet est (extérieurement) un objet simple : il possède un état ouvert et un état fermé et les opérations pour l'ouvrir ou le fermer. Ceci s'exprime comme :

```
package Clapet is
   procedure Ouvrir;
   procedure Fermer;
end Clapet;
```

Faut-il faire un paquetage (de type machine abstraite), alors que l'on pourrait avoir une simple procédure (avec un paramètre ouvert/fermé) ? Oui, car si la spécification est simple, l'implémentation peut être plus subtile. Par exemple, il faut garantir qu'on ne sort de la procédure `Ouvrir` (ou `Fermer`) que lorsque le clapet est effectivement ouvert ou fermé. Comme il s'agit d'un dispositif mécanique, ce n'est pas immédiat et il faut introduire un retard ou un contact annexe pour vérifier la position du relais. Cette dernière solution est un peu luxueuse pour nous (après tout, en cas de panne, le risque ne dépasse pas quelques francs), mais est indispensable pour les relais gouvernant des dispositifs de sécurité, comme ceux qui commandent les signaux des trains.

Pour pouvoir rendre la monnaie, il nous faut à l'évidence savoir combien il nous reste de pièces de chaque sorte et déclencher physiquement la chute d'une ou plusieurs pièces. Le *réservoir de pièces* doit donc fournir ces fonctionnalités. Comme nous devrons avoir un réservoir associé à chaque sorte de pièce, nous en faisons un type de donnée abstrait :

```
package Réservoirs_De_Pièces is
   Capacité_Max : constant := 100;
   type Capacité_réservoir is range 0 .. Capacité_Max;

   type Réservoir is private;

   function Nombre_De_Pièces (Dans : Réservoir)
     return Capacité_Réservoir;
   procedure Commander_Chute (Depuis : Réservoir);

   Réservoir_Vide : exception;
private
   ...
end Réservoirs_De_Pièces;
```

Remarquons au passage que nous avons dû définir un type pour la valeur retournée par `Nombre_De_Pièces`. Il aurait été tentant d'utiliser `Integer`... Mais il existe un problème bien réel : un magasin de pièces a nécessairement une capacité limitée. Nous préférons refléter cette propriété de l'objet physique dans les types du programme. Enfin, on ne peut exclure qu'une chute de pièce soit commandée alors qu'il ne reste rien dans le réservoir : nous devons donc prévoir une exception `Réservoir_Vide` pour ce cas.

Nous devons ensuite voir si l'objet ainsi défini nous permet de résoudre notre problème : rendre la monnaie. L'algorithme le plus simple consiste à laisser tomber des pièces de la plus grande valeur inférieure au montant à rendre, jusqu'à ce que le montant restant soit inférieur à la valeur de la pièce, ou que le réservoir soit épuisé ; puis à passer au réservoir de pièces de montant inférieur. Malheureusement, avec cet algorithme, on ne s'aperçoit que le rendu est impossible qu'après avoir commencé à laisser tomber des pièces... Il faut donc travailler en deux temps : d'abord calculer le nombre de pièces de chaque réservoir à rendre, puis, si le rendu est possible, commander les réservoirs. Le problème se complique : il va falloir définir quelque part un ensemble de pièces... Mais au fait, nous parlons de *pièces* et nous n'avons aucun objet correspondant dans notre analyse !

Il est temps de réfléchir à ce que nous allons appeler pièce, pour *notre vue* et à *ce niveau* d'abstraction.

Nous parlons ici des vraies pièces de monnaie, notion qui varie selon le pays, les fluctuations de la politique monétaire... Il serait bon de concentrer dans un seul paquetage tout ce qui dépend de la définition des pièces. Ceci comprend l'éventail des pièces disponibles, leurs valeurs, les réservoirs associés et la façon d'obtenir un montant donné à partir des pièces disponibles. Nous en faisons un paquetage distinct, pour pouvoir le modifier sans perturber le reste de l'application et pour lui garder une bonne autonomie ; mais comme il fait logiquement partie du monnayeur et n'a aucune raison d'être accessible de l'extérieur, nous en ferons un enfant privé. Ceci nous donne :

```
with Réservoirs_De_Pièces; use Réservoirs_De_Pièces;
private package Monnayeur.Définition_Pièces is
   type Pièces is (Cts_50,
                   Franc_1, Franc_2, Franc_5, Franc_10);
   Valeurs : constant array (Pièces) of Argent
           := (0.50, 1.00, 2.00, 5.00, 10.00);
   Magasin : array (Pièces) of Réservoir;
   type Répartition_Pièces is array (Pièces)
     of Capacité_Réservoir;

   function Répartir (Montant : Argent)
     return Répartition_Pièces;
end Monnayeur.Définition_Pièces;
```

Notons que la fonction `Répartir` est essentiellement un algorithme (non évident au demeurant) qu'il sera approprié d'analyser suivant les techniques de programmation structurée.

*Ecrire le corps de l'objet à concevoir*

Nous pouvons maintenant écrire le corps de l'objet monnayeur. Le compteur et la tâche sont des objets séparés, mais inclus dans le monnayeur. Nous séparerons donc leurs corps pour pouvoir compiler (et donc vérifier) l'utilisation que nous en faisons avant de songer à leur implémentation :

```
with Clapet, Monnayeur.Définition_Pièces;
package body Monnayeur is
   protected Compteur is
     procedure Ajouter (Montant : Argent);
     function  Valeur_Courante return Argent;
     procedure Remise_A_Zéro;
     entry Attendre (Montant : Argent);
   end Compteur;

   protected body Compteur is separate;

   task Surveillance;
   task body Surveillance is separate;

   procedure Activer is
   begin
     Compteur.Remise_A_Zéro;
     Clapet.Ouvrir;
   end Activer;

   procedure Bloquer is
   begin
     Clapet.Fermer;
   end Bloquer;

   procedure Attendre (Montant : Argent) is
   begin
     Compteur.Attendre(Montant);
   end Attendre;

   function  Montant_Introduit return Argent is
   begin
     return Compteur.Valeur_Courante;
   end Montant_Introduit;
```

```
   procedure Rendre_Monnaie (Montant : Argent) is
      use Définition_Pièces;
      Répartition : constant Répartition_Pièces
                  := Répartir (Montant);
   begin
      for A_Rendre in Pièces loop
         for Nombre in 1 .. Répartition (A_Rendre) loop
            Commander_Chute (Magasin (A_Rendre));
         end loop;
      end loop;
   end Rendre_Monnaie;
end Monnayeur;
```

*Contester*

La spécification du compteur telle qu'elle est répond à notre besoin ; mais est-elle suffisante dans le cas général ? Il reste un point important non spécifié : que se passe-t-il si plusieurs tâches appellent `Attendre` avant que le montant désiré ait été atteint ? En particulier, que se passe-t-il si plusieurs appels à `Attendre` ne spécifient pas le même montant attendu ? Bien sûr, on peut dire que cela n'a pas d'importance, vu qu'il n'y aura jamais qu'une seule tâche qui appelle `Attendre` dans notre système. Cependant, ceci est particulier à notre utilisation. Dans une conception orientée objet, nous devons réfléchir aux propriétés de l'objet indépendamment de toute utilisation particulière. Ce surcroît de travail peut paraître inutile, mais se justifiera dès la première réutilisation et l'expérience montre que ceci arrive souvent un peu plus tard dans le même projet.

Que faire donc si plusieurs tâches appellent `Attendre` avec des montants que nous n'avons aucune raison de supposer identiques ? Une première possibilité est de maintenir une liste de demandes triée par montant attendu croissant ; nous libérons les clients au fur et à mesure que le compteur atteint les différents seuils. Une autre possibilité est de ne traiter qu'une demande à la fois : si une nouvelle demande arrive, elle est bloquée jusqu'à ce que le compteur soit remis à zéro ; une nouvelle demande est alors acceptée et ainsi de suite. Enfin, on peut interdire plusieurs attentes simultanées : une deuxième demande recevra alors une exception.

La première solution correspond à des tâches qui voudraient être libérées progressivement au cours de la même «session» de comptage, la seconde au cas où chaque tâche dispose d'une session différente. Enfin la dernière est plus restrictive, correspondant à notre seul besoin, mais complètement spécifiée par rapport à notre première formulation. On peut trouver des exemples nécessitant chacune de ces solutions. Or dans notre cas, les trois sont également acceptables ! Nous manquons donc d'un critère pour faire notre choix. Dans ce cas, il est bon de jeter un coup d'œil sur les possibilités d'implémentation. S'il est nécessaire de toujours penser «réutilisation», il ne faudrait pas que cela conduise à des solutions inutilement compliquées par rapport au besoin ; nous pouvons utiliser la faisabilité comme critère de choix.

Etudions la réalisation de chacune de ces possibilités. Tout d'abord, il existe une difficulté technique commune aux trois : il faut attendre d'atteindre un montant qui figure en paramètre de l'appel à `Attendre`. Or la garde d'une entrée ne peut dépendre des paramètres de l'appelant. Autrement dit, pour connaître le montant à attendre, il faut accepter l'appel d'entrée. Ce problème est connu depuis longtemps [Wel83] et a été résolu en Ada 95 par l'instruction **`requeue`**. La «base commune» aux trois solutions peut s'écrire ainsi[1] :

```
protected Compteur is
   entry Attendre (Montant : Argent);
private
   entry Faire_La_Queue;
   Accumulateur    : Argent := 0.0;
   Montant_Attendu : Argent := 0.0;
end Compteur;
```

[1] Dans cette partie de la discussion, nous ne montrons pour simplifier que la partie du type protégé `Compteur` liée à l'entrée `Attendre`. Il faut bien entendu fournir aussi les autres fonctionnalités.

```
protected body Compteur is
   entry Faire_La_Queue
     when Accumulateur >= Montant_Attendu is
   begin
     null;
   end Faire_La_Queue;

   entry Attendre (Montant : Argent) when True is
   begin
     Montant_Attendu := Montant;
     requeue Faire_La_Queue with abort;
   end Attendre;
end Compteur;
```

La clause **when** `True` de l'entrée `Attendre` peut sembler bizarre, mais il est obligatoire de fournir une garde pour une entrée de type protégé. Ne peut-on en faire une procédure ? Non, car seule une entrée a le droit de faire un **requeue**. La mention **with abort** de l'instruction **requeue** autorise une éventuelle annulation de la demande par avortement ou fin d'appel temporisé.

Au fait, que se passe-t-il ici en cas d'appels multiples ? Le montant attendu est celui du dernier qui appelle ! Si le deuxième appel correspond à un montant inférieur au premier, la première demande sera débloquée alors que le montant attendu n'a pas été atteint. Un tel comportement est inacceptable.

La solution la plus simple consiste à lever une exception. L'entrée `Attendre` ne peut être appelée qu'une seule fois après une remise à zéro. Mais que faire si quelqu'un se trouve en attente lors de la remise à zéro ? Comme nous n'autorisons qu'une seule tâche à utiliser le compteur, le mieux est de lever également une exception. Laquelle ? Puisqu'il s'agit d'un comportement qui résulte d'une violation des règles d'utilisation de l'objet par le programmeur, le mieux est de lever `Program_Error`. Notre compteur devient :

```
protected Compteur is
   entry Attendre (Montant : Argent);
   entry Remise_A_Zéro;
private
   entry Faire_La_Queue;
   Accumulateur    : Argent  := 0.0;
   Montant_Attendu : Argent  := 0.0;
   Déjà_Utilisé    : Boolean := False;
end Compteur;

protected body Compteur is
   entry Faire_La_Queue
     when Accumulateur >= Montant_Attendu is
   begin
     null;
   end Faire_La_Queue;

   entry Attendre (Montant : Argent) when True is
   begin
     if Déjà_Utilisé then
       raise Program_Error;
     end if;
     Déjà_Utilisé    := True;
     Montant_Attendu := Montant;
     requeue Faire_La_Queue with abort;
   end Attendre;

   procedure Remise_A_Zéro is
   begin
     if Faire_La_Queue'Count > 0 then
       raise Program_Error;
     end if;
     Déjà_Utilisé := False;
     Accumulateur := 0.0;
   end Remise_A_Zéro;
end Compteur;
```

Une variante consiste à interdire l'appel d'`Attendre` si quelqu'un est déjà en attente, mais à autoriser plusieurs attentes même s'il n'y a pas de remise à zéro intermédiaire :

```
protected Compteur is
   entry Attendre (Montant : Argent);
private
   entry Faire_La_Queue;
   Accumulateur    : Argent := 0.0;
   Montant_Attendu : Argent := 0.0;
end Compteur;

protected body Compteur is
   entry Faire_La_Queue
     when Accumulateur >= Montant_Attendu is
   begin
     null;
   end Faire_La_Queue;

   entry Attendre (Montant : Argent) when True is
   begin
     if Faire_La_Queue'Count > 0 then
        raise Program_Error;
     end if;
     Montant_Attendu := Montant;
     requeue Faire_La_Queue with abort;
   end Attendre;
end Compteur;
```

Nous n'avons pas examiné les solutions permettant de libérer plusieurs tâches au fur et à mesure de l'incrément du compteur ; elles sont nettement plus compliquées, aussi proposons-nous de nous en tenir à l'une de celles que nous avons étudiées. La plus simple est finalement la dernière : pas d'exception à traiter, expression naturelle du comportement par les gardes du type protégé. Nous la choisirons donc, mais nous retiendrons que d'autres étaient acceptables.

Un œil critique remarquera aisément que le compteur protégé que nous venons de définir semble très utile ; tellement utile même qu'il est vraisemblable que de nombreux projets pourraient avoir besoin de fonctionnalités semblables. Ne pourrait-on en faire un composant réutilisable ? Demandons-nous donc ce qu'*est* un compteur. C'est quelque chose qui contient une valeur, qui nous permet de l'augmenter et de connaître sa valeur courante. Nous pourrions nous limiter à compter des nombres entiers, mais *a priori* rien n'empêche de compter d'autres choses ; la seule nécessité est que la notion d'*addition* ait un sens pour les valeurs à compter, ainsi bien entendu qu'une valeur nulle pour la remise à zéro. Enfin, comme nous voulons pouvoir attendre d'atteindre une valeur, il est nécessaire de disposer d'une fonction de comparaison. Ceci s'exprime comme suit :

```
generic
   type A_Compter is private;
   Valeur_Nulle : in A_Compter;
   with function "+" (Left, Right: A_Compter)
     return A_Compter is <>;
   with function "<" (Left, Right: A_Compter)
     return Boolean is <>;
package Compteur_Protégé_Générique is

   procedure Attendre (Valeur : A_Compter);

   procedure Incrémenter (De : A_Compter);
   function  Valeur_Courante return A_Compter;
   procedure Remise_A_Zéro;

end Compteur_Protégé_Générique;
```

On trouvera le corps du compteur générique en annexe. Le corps final et définitif (ce n'est plus une maquette) du monnayeur devient :

```
with Compteur_Protégé_Générique,  Réservoirs_De_Pièces,
     Monnayeur.Définition_Pièces, Clapet;
package body Monnayeur is
   package Le_Compteur is
     new Compteur_Protégé_Générique (
       A_Compter    => Définition_Argent.Argent,
       Valeur_Nulle => 0.0);

   task Surveillance;
   task body Surveillance is separate;

   procedure Activer is
   begin
     Le_Compteur.Remise_A_Zéro;
     Clapet.Ouvrir;
   end Activer;

   procedure Bloquer is
   begin
     Clapet.Fermer;
   end Bloquer;

   procedure Attendre (Montant : Argent) is
   begin
     Le_Compteur.Attendre(Montant);
   end Attendre;

   function  Montant_Introduit return Argent is
   begin
     return Le_Compteur.Valeur_Courante;
   end Montant_Introduit;

   procedure Rendre_Monnaie (Montant : Argent) is
     use Monnayeur.Définition_Pièces, Réservoirs_De_Pièces;
     Répartition : constant Répartition_Pièces
                 := Répartir (Montant);
   begin
     for A_Rendre in Pièces loop
       for Nombre in 1 .. Répartition (A_Rendre) loop
         Commander_Chute (Magasin (A_Rendre));
       end loop;
     end loop;
   end Rendre_Monnaie;

end Monnayeur;
```

### c) Tester le plan d'abstraction

Pour pouvoir tester le plan d'abstraction, il nous manque le corps de la tâche `Surveillance` (qui fait logiquement partie de ce plan). Celle-ci est intimement liée au matériel, aussi allons-nous nous contenter de fournir un corps maquette. Les corps maquettes des modules `Clapet` et `Définition_Pièces` (qui n'ont rien de particulier) sont fournis en annexe.

```
with Ada.Text_IO; use Ada.Text_IO;
separate (Monnayeur) task body Surveillance is
   C : Character;
begin
   loop
     Put ("Pièce entrée :");
     Get_Immediate (C);
     case C is
     when 'C' => Le_Compteur.Incrémenter (De =>  0.50);
     when '1' => Le_Compteur.Incrémenter (De =>  1.00);
     when '2' => Le_Compteur.Incrémenter (De =>  2.00);
     when '5' => Le_Compteur.Incrémenter (De =>  5.00);
     when 'D' => Le_Compteur.Incrémenter (De => 10.00);
     when others =>
          null;    -- Ignorer les autres caractères
     end case;
     Put ("Montant total : ");
     Put (Argent'Image (Montant_Introduit));
     New_Line;
   end loop;
end Surveillance;
```

La procédure `Get_Immediate`, de `Text_IO`, permet de lire un caractère au clavier sans attendre de retour chariot.

## d) Documenter le plan d'abstraction

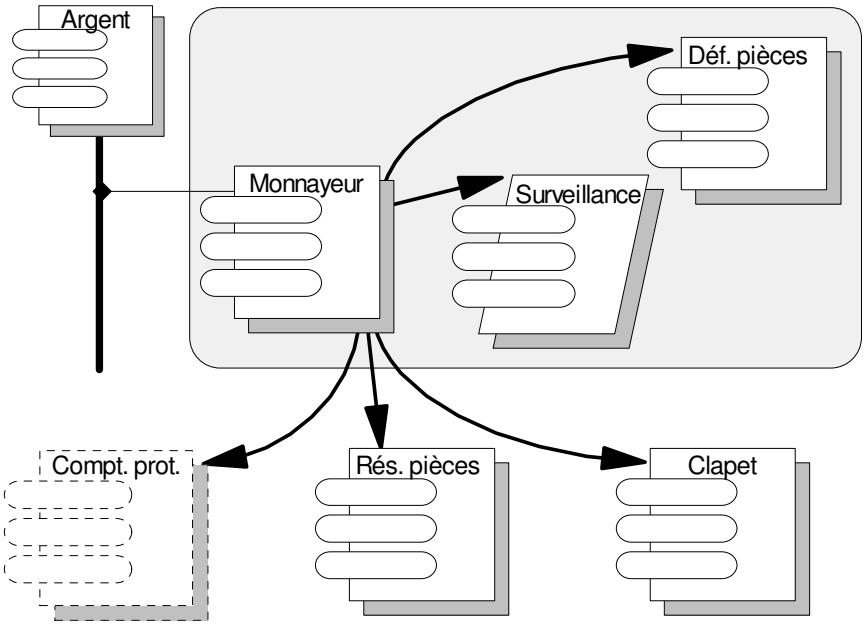


**Figure 41 :** Graphe du monnayeur

Résumons ici les points importants de la structure que nous avons adoptée. Le monnayeur est constitué d'un compteur protégé et d'une tâche chargée de gérer la chute des pièces. Il gère des réservoirs de monnaie contenant les pièces, et le clapet qui autorise ou interdit la chute des pièces. Le graphe (partiel) concernant la vue du monnayeur est indiqué en figure 41, où le rectangle grisé exprime que les éléments qui le constituent forment un sous-système.

## *24.2 .3 L'unité de fabrication*

### a) Reprendre les spécifications

L'interface définie précédemment est simple et correspond bien à un objet du monde réel (il suffit d'avoir vu un distributeur ouvert pour s'en convaincre). Il ne semble pas nécessaire d'y apporter de modifications.

### b) Définir le plan d'abstraction

L'unité de fabrication est chargée de confectionner les produits. Cette «confection» est relativement simple : il suffit de piloter à tour de rôle un certain nombre de distributeurs d'ingrédients. Il faut noter quelque part la liste des ingrédients pour un produit : appelons cela une *recette*. Le paquet de cacahuètes n'est alors qu'un cas simple de recette, ne comportant qu'un seul ingrédient : le paquet lui-même. En résumé, le distributeur doit piloter des *distributeurs d'ingrédients* en fonction de la *recette* du produit. Une recette est une liste d*'ingrédients* entrant dans la fabrication du produit. Pour établir la relation entre un produit et sa recette, il faut un *livre de recettes*.

Un *distributeur d'ingrédients* est chargé de *servir* quelque chose : une cuillère, un gobelet, une certaine quantité de sucre ou de liquide. Les distributeurs d'ingrédients peuvent être ouverts ou fermés. Le problème est de déterminer en fonction de quel critère on décidera de les refermer. Par exemple, les distributeurs de petites cuillères ont un fonctionnement élémentaire : distribuer un élément. La fermeture d'un distributeur de liquide peut être commandée au bout d'un certain temps, ou lorsqu'un débitmètre aura mesuré une certaine quantité de liquide. De même, la distribution du sucre peut être commandée de façon binaire (sucre en doses), après un temps d'écoulement, ou selon une balance... Le choix entre ces solutions va dépendre des caractéristiques du matériel sous-jacent, mais il serait dangereux de se lier à des caractéristiques de si bas niveau pour le moment. Aussi, nous allons *différer* la décision en n'introduisant pour l'instant que des ingrédients de haut niveau. On considérera ainsi que «un peu d'eau» ou «beaucoup d'eau» (pour un café court ou long) sont des ingrédients différents. Ils pourront bien entendu commander le même dispositif physique.

Si extérieurement nous voulons faire apparaître tous les ingrédients de façon uniforme, chaque implémentation sera différente. Il est certain que la notion de distributeur d'ingrédients est polymorphe. Nous avons un choix de conception à ce niveau entre types à discriminant et type étiqueté. Un type à discriminant serait certainement acceptable[1] ; toutefois, cela nécessiterait que chacune des opérations sur les distributeurs d'ingrédients comporte des instructions **`case`** selon le type de produit distribué ; nous n'aurions pas centralisé tout ce qui concerne un ingrédient particulier. Cet argument paraît faible ici, puisqu'il n'y a qu'une seule opération sur les distributeurs d'ingrédient ; mais nous raisonnons dans le cas général et rien ne permet de penser qu'une évolution ultérieure du logiciel ne nécessitera pas de rajouter des opérations. De plus, les différentes sortes d'ingrédients sont indépendantes : il n'y a aucune raison que la distribution des petites cuillères «connaisse» la distribution d'eau. Il paraît plus judicieux de séparer les implémentations de chaque ingrédient dans des procédures distinctes ; nous ferons donc d'`Ingrédient` un type étiqueté abstrait (il n'est pas possible d'avoir un ingrédient qui ne serait pas en même temps quelque chose de plus précis) et chaque ingrédient sera dérivé de la classe générale. Le type général n'a aucune propriété particulière, il ne sert que de racine. Cependant, il doit avoir une sémantique de référence (on ne peut «copier» un dispositif physique). Pour renforcer cette sémantique, la procédure `Servir` utilisera un «paramètre accès» : nous exprimons bien que nous ne pouvons manipuler que des références à des distributeurs.

> Les paramètres accès constituent un nouveau mode (en plus de **`in`**, **`out`** et **`in out`**), rajouté par Ada 95, sous la forme **`access`** <type>. Ils correspondent à n'importe quelle valeur accès dont le type *désigné* est celui

[1] C'est ce que nous aurions utilisé en Ada 83.

indiqué. Les sous-programmes utilisant ces paramètres sont des sous-programmes primitifs du type *désigné* : ils peuvent donc être utilisés pour la liaison dynamique.

Enfin, il faudra accéder par des moyens matériels au dispositif physique ; il semble donc approprié de paramétrer le type au moyen d'un discriminant caractéristique de l'appareil. Quel doit être le type de ce discriminant ? Nous supposons que nous accédons aux appareils par des interfaces sous forme de *mappings* d'adresses ; le type `Address` semble approprié. Mais d'une part, la définition de `Address` dépend de l'implémentation et d'autre part il se peut que l'adresse d'un appareil ne corresponde pas à une adresse mémoire, auquel cas un type entier serait plus «ouvert» pour fournir la notion générale d'«adresse de périphérique». Le type `Integer_Address` procure un compromis acceptable entre ces deux contraintes.

Le type `Integer_Address` du paquetage `System.Storage_Elements` est est une «vue» sous forme de nombre entier de la notion d'adresse. Le paquetage fournit des conversions entre ce type et le «vrai» type `Address`. Ceci permet d'effectuer des contrôles lors des conversions et d'éviter les problèmes connus lors de l'utilisation indisciplinée de nombres entiers comme adresses.

Le paquetage définissant la classe des distributeurs ne peut (et ne doit) être utilisé que par l'unité de fabrication ; il n'y a aucune raison de le laisser accessible de l'extérieur. Nous en ferons donc un enfant privé. Nous pouvons exprimer ces décisions de conception comme :

```
with System.Storage_Elements; use System.Storage_Elements;
private package Unité_Fabrication.Distributeur is
   type Instance (Adresse : Integer_Address) is
      abstract tagged limited private;
   subtype Classe is Instance'Class;

   procedure Servir_Dose(De : access Instance) is abstract;
private
   type Instance (Adresse : Integer_Address) is
      abstract tagged limited null record;
end Unité_Fabrication.Distributeur;
```

Nous ne pouvons exclure le cas où l'on demanderait au distributeur de confectionner un `Produit` épuisé, ou simplement celui où un ingrédient viendrait à manquer en cours de fabrication. La procédure `Servir_Dose` de chaque implémentation doit lever l'exception `Confection_Impossible` si elle échoue pour quelque raison que ce soit.

A partir de ce type de base, on dérive différents types de distributeurs d'ingrédients concrets, correspondant aux différents modèles d'appareils physiques : distributeur de liquide, de poudre, distributeur élémentaire (petite cuillère ou paquet de cacahuète). Certains produits peuvent avoir besoin de paramètres : quantité fournie, temps de réaction d'un relais, que l'on fournira grâce à des discriminants supplémentaires lors de la dérivation ; encore faut-il définir quelque part les types correspondants. Où ? Ces types sont des types globaux pour le sous-système de l'unité de fabrication ; le plus simple est de les mettre dans la spécification de `Unité_Fabrication`, ce qui les rendra visibles pour toutes les unités enfants. Mais comme il n'y a pas de raison de les rendre visible de l'extérieur du sous-système, il faut les mettre dans la partie *privée* de `Unité_Fabrication`. Comme il n'y a pas de modification de la partie *visible* du paquetage, nous savons que ceci n'aura aucune conséquence sur les couches de plus haut niveau. Le paquetage devient donc:

```
with Définition_Produit; use Définition_Produit;
package Unité_Fabrication is
   procedure Confectionner (Le_produit : Produit);

   Confection_Impossible : exception;
private
   type Temps_Ouvert is range 0 .. 10_000; -- en ms.
   ms : constant Temps_Ouvert := 1;

   type Volume_Livré is range 0 .. 50; -- en cl.
   cl : constant Volume_Livré := 1;
end Unité_Fabrication;
```

Remarquer la définition de la constante `ms` : elle autodocumente que l'unité de mesure est la milliseconde et permet d'écrire une valeur sous la forme (plus lisible) `3*ms`. En cas d'évolution du logiciel, si par exemple on décide d'utiliser le dix-millième de seconde comme unité du type, il suffira de changer cette valeur pour ne pas perturber les utilisateurs. La même remarque s'applique à `cl`.

Pour le distributeur de boissons, nous avons besoin de trois sortes de distributeurs d'ingrédients : le modèle simple qui laisse tomber un objet (petite cuillère), paramétré en fonction du temps pour s'assurer que l'objet est bien tombé, le distributeur de poudre (sucre, cacao) et le distributeur de liquide, paramétré par la quantité à distribuer. Ces paquetages sont des enfants privés du distributeur. Ceci nous donne :

```
with Unité_Fabrication.Distributeur;
with System.Storage_Elements; use System.Storage_Elements;
private package Unité_Fabrication.Distributeur_Simple is
   type Instance (Adresse   : Integer_Address;
                  Ouverture : Temps_Ouvert)
     is new Distributeur.Instance (Adresse) with private;
   procedure Servir_Dose (De : access Instance);
private
   type Instance (Adresse   : Integer_Address;
                  Ouverture : Temps_Ouvert) is
     new Distributeur.Instance (Adresse) with null record;
end Unité_Fabrication.Distributeur_Simple;

with Unité_Fabrication.Distributeur;
with System.Storage_Elements; use System.Storage_Elements;
private package Unité_Fabrication.Distributeur_Poudre is
   type Instance (Adresse   : Integer_Address)is
     new Distributeur.Instance (Adresse) with private;
   procedure Servir_Dose (De : access Instance);
private
   type Instance (Adresse   : Integer_Address)is
     new Distributeur.Instance (Adresse) with null record;
end Unité_Fabrication.Distributeur_Poudre;

with Unité_Fabrication.Distributeur;
with System.Storage_Elements; use System.Storage_Elements;
private package Unité_Fabrication.Distributeur_Liquide is
   type Instance (Adresse : Integer_Address;
                  Volume  : Volume_Livré)
     is new Distributeur.Instance (Adresse) with private;
   procedure Servir_Dose (De : access Instance);
private
   type Instance (Adresse : Integer_Address;
                  Volume  : Volume_Livré) is
     new Distributeur.Instance (Adresse) with null record;
end Unité_Fabrication.Distributeur_Liquide;
```

Nous n'avons défini pour l'instant que des *types* de distributeurs ; il nous faut maintenant définir des *instances*. On peut comparer ceci au montage de l'appareil réel : nous allons chercher dans les stocks des distributeurs d'ingrédients afin de les monter dans l'appareil. L'ensemble des distributeurs montés constitue la configuration de l'appareil. Nous pouvons décrire ceci comme :

```
with Unité_Fabrication.Distributeur_Simple;
with Unité_Fabrication.Distributeur_Poudre;
with Unité_Fabrication.Distributeur_Liquide;
private package Unité_Fabrication.Configuration is
   Cuillère  : aliased Distributeur_Simple.Instance
                 (Adresse   => 16#1F4#, Ouverture => 50*ms);
   Cacao     : aliased Distributeur_Poudre.Instance
                 (Adresse => 16#1F6#);
   Café      : aliased Distributeur_Poudre.Instance
                 (Adresse => 16#1F8#);
   Sucre     : aliased Distributeur_Poudre.Instance
                 (Adresse => 16#1FA#);
   Eau_courte: aliased Distributeur_Liquide.Instance
                 (Adresse => 16#1FC#, Volume  => 10*cl);
   Eau_longue: aliased Distributeur_Liquide.Instance
                 (Adresse => 16#1FE#, Volume  => 25*cl);
end Unité_Fabrication.Configuration;
```

Une recette est une liste d'ingrédients associée à un produit. La recette d'un produit est fournie par un livre de recettes. En bonne logique, il faut une opération pour ajouter une recette au livre ; en pratique, on peut supposer que les recettes sont construites dans le corps du paquetage. On pourra ajouter cette fonctionnalité par la suite, par exemple pour permettre de changer les recettes «sur place» au moyen d'un terminal portable. Ce paquetage est une machine abstraite qui n'est utilisée que par l'unité de fabrication ; nous en faisons donc un paquetage enfant.

```
with Unité_Fabrication.Distributeur, Définition_Produit;
use  Unité_Fabrication.Distributeur, Définition_Produit;
package Unité_Fabrication.Livre_de_recettes is
   type Index_Recette is range 0..10;
   type Elément_Recette is
     access constant Unité_Fabrication.Distributeur.Classe;
   type Recette is
     array (Index_Recette range <>) of Elément_Recette;

   function La_Recette (De : Produit) return Recette;
end Unité_Fabrication.Livre_de_recettes;
```

Nous pouvons maintenant écrire le corps de `Unité_Fabrication` :

```
with Unité_Fabrication.Distributeur,
     Unité_Fabrication.Livre_de_recettes;
package body Unité_Fabrication is
   procedure Confectionner (Le_Produit : Produit) is
     use Distributeur, Livre_de_recettes;
     R: constant Recette := La_Recette (De => Le_Produit);
   begin
     for I in R'Range loop
       Servir_Dose (R (I)); -- Liaison dynamique
     end loop;
   end Confectionner;
end Unité_Fabrication;
```

Nous avons pu supprimer les dépendances à `ADPT` : le corps de ce paquetage est définitif.

### c) Tester le plan d'abstraction

Il manque encore l'implémentation des différents distributeurs d'ingrédients, ainsi que du livre de recettes. On peut facilement fournir des corps maquettes pour les ingrédients, comme :

```
with ADPT;
package body Unité_Fabrication.Distributeur_Simple is
   procedure Servir_Dose (De : access Instance) is
     use ADPT;
   begin
     ADPT_Action
       ("Je distribue cuillère : " &
        Integer_Address'Image (De.Adresse) & " pendant " &
        Temps_Ouvert'Image (De.Ouverture)  & " ms");
   end Servir_Dose;
end Unité_Fabrication.Distributeur_Simple;
```

On trouvera en annexe les autres corps maquettes correspondant aux différentes formes de distributeurs. En ce qui concerne le livre de recettes, on peut lui fournir un premier corps comme :

```
with Unité_Fabrication.Configuration;
use  Unité_Fabrication.Configuration;
package body Unité_Fabrication.Livre_de_recettes is
   function La_Recette (De : Produit) return RECETTE is
   begin
      case De is
      when 1 =>
         return (Cuillère'Access, Café'Access,
                 Sucre'Access,    Eau_Longue'Access);
      when 2 =>
         return (Cuillère'Access, Café'Access,
                 Sucre'Access,    Eau_Courte'Access);

      when others =>
         return (Cuillère'Access, Cacao'Access,
                 Eau_Longue'Access);
      end case;
   end La_Recette;
end Unité_Fabrication.Livre_de_recettes;
```

Bien sûr, à terme il faudra remplacer ce corps en utilisant une table, mais celui-ci est suffisant pour nous permettre de tester l'ensemble du logiciel, *y compris dans ses aspects temporels*.

### d) Documenter le plan d'abstraction

Le rôle de l'unité de fabrication est essentiellement de séparer les points de vue. Chaque distributeur d'ingrédients réalise la distribution d'un ingrédient précis, l'unité de fabrication réalise l'assemblage des composants *sans avoir à connaître les éléments qu'elle met en œuvre*, grâce au mécanisme de la liaison dynamique. La *configuration* décrit les éléments matériels effectivement présents dans l'appareil.

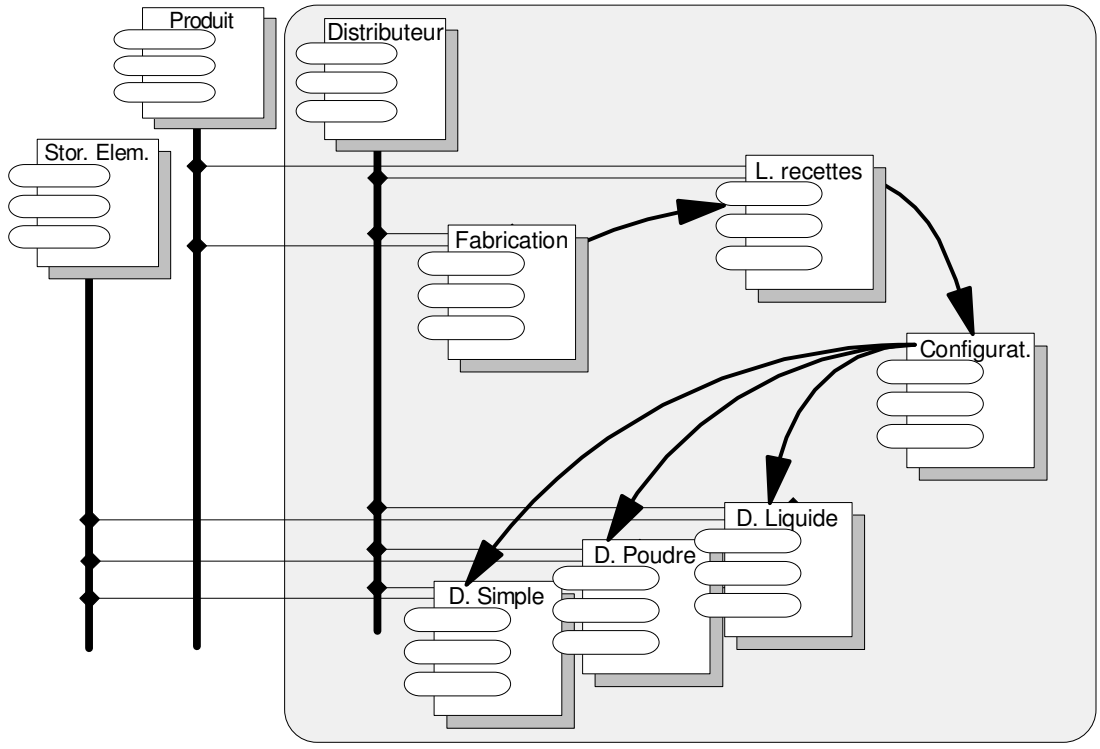


**Figure 42 :** Dépendances de l'unité de fabrication

On notera sur le schéma de la figure 42 la présence d'un *bus local* au sous-système : le `Distributeur`. C'est la notion abstraite de distributeur d'ingrédient, utilisée par tous les autres modules, qui leur permet de communiquer. En dehors du bus, la `Fabrication` ne dépend que du `Livre_De_Recettes` et seul le corps de celui-ci dépend de la `Configuration`. Ceci montre

clairement qu'un changement de configuration n'aurait que des conséquences très limitées sur le logiciel. La `Configuration` dépend des différents `Distributeurs` (concrets) qui la composent, et uniquement de ceux-ci : on retrouve qu'elle correspond à un «montage» concret.

## 24.3 Récapitulation et analyse

Les objets que nous avons identifiés sont extrêmement généraux et cette structure peut servir à réaliser n'importe quel automate simple. Le monnayeur est directement réutilisable pour tout appareil devant gérer de l'argent (parcmètre, machine à sous...). L'unité de fabrication n'est liée à aucune particularité des boissons : c'est seulement au niveau des distributeurs d'ingrédients qu'apparaît une spécialisation.

Mais le point le plus important est l'évolutivité : comment réagirait le programme en cas de changements dans les distributeurs physiques, pourrait-on le réutiliser pour d'autres appareils ? Un changement de commande des appareils électromécaniques n'impliquerait que des changements dans le corps de l'unité qui les modélise. L'ajout d'un nouvel appareil nécessiterait l'écriture d'une nouvelle classe dérivée ; on pourrait alors l'introduire dans la configuration et l'utiliser dans des recettes sans autre modification. De même, le monnayeur pourrait être remplacé par un appareil plus perfectionné : lecteur de carte de crédit par exemple. *Cela ne changerait rien au reste du programme*. Pour changer le prix des boissons ou la composition des produits, il suffit de changer une table dans le système, certainement une PROM amovible, permettant de remettre à jour le distributeur sans changer le programme.

Remarquons que cet exemple est indiscutablement *orienté objet* : tous les éléments manipulés par le programme sont des abstractions correspondant directement aux modules matériels de l'appareil. Cependant, l'héritage ne joue *pas* un rôle prépondérant : nous l'avons utilisé là où il était utile, mais, n'ayant pas raisonné par *classification* mais par *composition*, il n'apparaît pas dans l'organisation générale du projet.

Nous terminerons par une remarque plus philosophique. Le lecteur a peut-être eu par endroits le sentiment que certains états d'âme étaient abusifs ; qu'il existait une solution «bestiale» qu'il suffisait d'appliquer sans se poser tant de questions. Comme nous avons utilisé Ada comme langage d'implémentation, il pourrait en tirer la conclusion qu'Ada «complique les choses». En fait le problème n'est pas là. Spécifier complètement les comportements même dans les cas limites, étudier systématiquement plusieurs possibilités même lorsqu'une solution paraît évidente, chercher à définir des entités réutilisables doivent (devraient ?) être un souci permanent du développeur quel que soit le langage utilisé. Il se trouve simplement que le pouvoir d'expression d'Ada permet d'exprimer et de maquetter ces différentes solutions, là où d'autres langages n'offrent qu'une seule possibilité. Le programmeur Ada aura donc tendance à se poser plus de questions ; mais dans un contexte de génie logiciel, ceci doit être vu comme un avantage important du langage.

## 24.4 Exercices

1. Reprendre l'exemple avec une méthodologie par classification. Comparer l'organisation des solutions obtenues.
2. Ecrire le logiciel d'un changeur de monnaie, capable de convertir des francs en dollars. On cherchera à réutiliser au maximum les modules du distributeur.
3. Dessiner le graphe complet des dépendances du distributeur, et comparer le nombre de dépendances au nombre de modules. Combien vaut $K$ en moyenne ?

# Conclusion

Arrivé à ce point, le lecteur que nous avons pu convaincre de l'intérêt d'Ada se posera certainement la question : pourquoi n'est-il pas plus répandu ? On pensait au début qu'Ada remplacerait rapidement les langages alors principalement utilisés : FORTRAN, Pascal et, dans une moindre mesure, COBOL. Si le langage s'est bien introduit dans les domaines sensibles (aéronautique, aviation, domaine militaire, nucléaire, contrôle de processus), sa diffusion est restée modeste dans les domaines de la programmation traditionnelle : scientifique, gestion, programmation système, alors qu'on assistait à une montée en force du langage C, et plus récemment de C++.

Pourtant, toutes les mesures effectuées sur des projets réels ont montré que les bénéfices que l'on pouvait espérer d'Ada étaient obtenus ou dépassés : meilleure qualité de la conception, réutilisation, augmentation de la productivité des programmeurs, infléchissement de la courbe des coûts en fonction de la taille des logiciels, effondrement du taux d'erreurs résiduelles et des coûts d'intégration, efficacité du code produit ; et ceci, quel que soit le domaine d'application concerné : on trouvera par exemple dans [Leb94] une étude complète sur l'impact de l'utilisation d'Ada en gestion. Mais le choix d'un langage de programmation fait intervenir de nombreux éléments, qui touchent plus à la psychologie qu'à l'économie.

Tout d'abord, l'expansion de C est liée au développement du système UNIX. Il est de tradition de fournir gratuitement le compilateur C avec les outils standard d'UNIX[1]. Pourquoi alors acquérir un autre langage? D'autant plus que C a pour lui une extrême simplicité (d'aucuns disent même pauvreté) de concepts. Remarquons au passage que ces arguments ne s'appliquent plus à C++ : les compilateurs sont payants, et le langage est devenu très compliqué !

Historiquement, UNIX a été développé par un petit nombre d'individus, sans souci de politique d'ouverture commerciale. Les interfaces ont donc été pensées uniquement en fonction des particularités du C ; c'est ainsi qu'une fonction peut retourner soit un pointeur, soit la valeur `False` (c'est-à-dire en fait zéro, qui est également le pointeur nul !). De telles hérésies de typage sont incompatibles avec la plupart des langages évolués, mais fréquentes avec C. Il en résulte des difficultés à faire des interfaces abstraites, propres, pour beaucoup de fonctionnalités UNIX, ce qui accentue le retard, en particulier au niveau de la standardisation, des autres langages par rapport à C. Et puis la «sagesse populaire» affirme que l'on *doit* programmer en C sous UNIX, sans plus d'explication. Bien sûr, il existe de nombreux composants et de nombreuses interfaces adaptés à Ada. Mais les vendeurs font rarement un effort commercial conséquent pour leur promotion, et il faut les acheter en plus, alors que beaucoup de bibliothèques C sont fournies avec le système. Là encore, l'effort *initial* est plus important avec Ada, même s'il est aisé de prouver que ce surcoût au départ est largement amorti sur le long terme.

En dehors de ces considérations pratiques, l'attirance du programmeur moyen pour C, et la peur instinctive qu'il éprouve vis-à-vis d'Ada, plongent leurs racines bien plus profondément. Beaucoup de programmeurs actuellement en fonction ont été formés à une époque où l'assembleur était roi. Nous avons connu des centres de calcul où les «nobles» (ceux qui étaient capables d'écrire tous leurs programmes en langage machine) regardaient de haut les novices qui n'étaient bons qu'à programmer en FORTRAN... Avec le temps, la programmation en assembleur tend à disparaître, car les problèmes de fiabilité, de maintenabilité et de portabilité sont vraiment devenus trop importants. Que sont devenus ces programmeurs[2] ? Ils font du C. Ils ont trouvé un langage qui n'est en fait,

[1] Ceci tend à disparaître... ce qui fait une raison de plus d'utiliser Ada !

[2] N'oublions pas que les programmeurs qui étaient débutants à l'époque héroïque (début des années 60) n'ont pas

selon la définition de ses auteurs, qu'un assembleur portable. Comme nous l'avons vu, il permet de faire (presque) tout ce que l'on pouvait faire en langage machine et n'implique aucune remise en cause ni des méthodes ni de la façon de programmer. Inversement, Ada a été spécifiquement conçu pour *obliger* les programmeurs à modifier leurs habitudes ! En particulier, la programmation Ada s'accompagne d'une description plus abstraite des traitements. Outre que ceci nécessite un effort de réflexion plus important, le programmeur tend à perdre le contact direct avec la machine. Il doit faire confiance au compilateur pour la génération de code efficace, et ne doit plus se préoccuper du parcourt exact du programme. Il n'est guère étonnant que l'inertie naturelle ne rende pas Ada très attrayant *a priori* au programmeur qui n'a pas saisi (car il n'a jamais essayé) les bénéfices d'une approche de plus haut niveau. Mais bien sûr, l'attirance naturelle du programmeur ne saurait être un critère pour un choix technologique dont dépendra tout le projet ! Comme il a été remarqué :

> *C++ ressemble à un énorme gâteau à la crème, ruisselant de sucreries ; Ada ressemble plus à un poisson poché / pommes vapeur. Les questions intéressantes sont 1) quel est le meilleur pour votre santé et 2) qu'est-ce que les gens ont tendance à choisir spontanément ?*

En outre, les compilateurs Ada ont parfois mauvaise réputation. Les premiers n'étaient disponibles que sur un petit nombre de machines, et peu efficaces, quand ils n'étaient pas franchement «buggés». Les premières années, l'effort des fabricants s'est plus porté sur la conformité à la norme, vérifiée par l'incontournable suite de validation, que sur l'efficacité pour laquelle il n'y avait aucune obligation légale. Des compilateurs sont maintenant disponibles sur quasiment toutes les machines du marché, et des mesures objectives ont montré qu'ils étaient au moins aussi performants et fiables que les compilateurs C. Il n'en reste pas moins que le langage continue à traîner quelques vieilles «casseroles», d'autant qu'il est difficile de supprimer de la littérature les comptes rendus des premières expériences. C'est ainsi que [Bur85] signale qu'un rendez-vous prend 100 ms sur un Vax, ce qui rend le mécanisme inapte au temps réel serré. De nos jours, un rendez-vous prend moins de 100 µs (jusqu'à 4 µs sur certains systèmes spécialisés), mais le livre de Burns (excellent au demeurant) est toujours en vente et n'a pas été réactualisé...

Un langage ne peut se répandre que s'il est enseigné ; force est de reconnaître qu'Ada n'a pas encore réussi à séduire tous les universitaires. Le prix des compilateurs, justifié en milieu industriel par le gain de productivité apporté, est un obstacle certain. C'est ce qui a conduit le DoD à financer le GNAT, pour mettre à la disposition de tous un compilateur Ada 95 gratuit. Ensuite, chercheurs et enseignants ont généralement des contraintes différentes des industriels : les logiciels sont rarement maintenus, mais doivent souvent être développés très rapidement. Même si les cours prônent le génie logiciel et l'importance de favoriser la maintenance au détriment de la facilité de développement, les universitaires appliquent rarement ces bons principes à eux-mêmes[1]... Inversement, les langages orientés objet sont à la mode et semblent offrir des facilités supplémentaires (mais cet argument n'a plus lieu d'être avec Ada 95). Enfin, le fait que le DoD ait financé le développement d'Ada lui a créé une certaine antipathie dans les milieux universitaires, traditionnellement antimilitaristes.

Paradoxalement, les promoteurs du langage ont pu lui faire du tort par excès de purisme en exigeant de faire du 100% Ada. Le langage a tout prévu pour permettre d'écrire des modules en d'autres langages si c'était plus commode ; il autorise même l'écriture de modules en assembleur. Si ces éléments sont alors non portables, ce n'est pas trop grave dans la mesure où le mécanisme d'empaquetage garantit que ces non-portabilités sont répertoriées, aisément identifiées, et sans effet sur les utilisateurs des paquetages. Il n'y a donc pas de honte à écrire un corps de module en assembleur si, par exemple, il n'y a pas d'autre moyen d'obtenir les performances requises. En exigeant le «tout Ada» au-delà du raisonnable, on risque de se heurter à des problèmes tout à fait localisés, mais dont la publicité risque de discréditer le langage tout entier.

---

encore atteint l'âge de la retraite !

[1] Que les universitaires nous pardonnent ces quelques critiques – nous avons été des leurs pendant plus de dix ans.

Enfin, il est remarquable de constater que les plus virulents adversaires d'Ada... ne l'ont jamais pratiqué ! Peut-être ont-ils juste fait quelques essais, sans formation appropriée, en traduisant «littéralement» des bouts de code d'autres langages avec lesquels ils étaient familiers. Au bout de quelques erreurs de compilation, ils ont décidé que le langage était difficile, et n'ont pas poursuivi. Inversement, ce n'est qu'à l'usage que l'on en apprécie tous les avantages : typage fort, exceptions, parallélisme... Il est certain que pour tirer tout le bénéfice d'Ada, il ne suffit pas d'apprendre la syntaxe ; il faut assimiler l'état d'esprit, la façon de faire qui va avec. Il faut admettre qu'un langage comme Ada joue un rôle *différent* dans le processus de développement et que, comme nous l'avons démontré dans la première partie, le choix du langage de codage n'est pas neutre – précisément parce qu'il peut être beaucoup plus qu'un simple outil de codage. Comme le remarquait Booch [Boo91] :

> *Donnez une perceuse électrique à un charpentier qui n'a jamais entendu parler de l'électricité, et il l'utilisera comme marteau. Il tordra des clous et se tapera sur les doigts, car une perceuse fait un très mauvais marteau.*

Si les mauvaises nouvelles courent vite, il est plus difficile de répandre les bonnes, et Ada n'a pas toujours su «faire sa pub». Qui, en dehors des convaincus, sait qu'Ada a été utilisé avec succès dans de nombreux programmes de gestion, dont un projet de 2,5 millions de lignes de code qui a pu être développé avec la moitié du coût initialement prévu (STANFINS, [Leb94]) ? Qu'un logiciel qui n'arrivait pas à tenir ses contraintes temps réel en C a été récrit en Ada avec des performances satisfaisantes [Tar93] ? Qu'une entreprise gère plusieurs projets totalisant un million de lignes de code en n'employant qu'une seule personne à la maintenance [Pid93] ?

Soyons honnête cependant : les utilisateurs d'Ada 83 ont pu rencontrer des difficultés objectives, même si la plupart ont appris à «vivre avec». On notera en particulier le manque de support des méthodes par classification, la prolifération des tâches pour de simples synchronisations, de trop nombreuses recompilations dans les grands projets, des difficultés d'interfaçage avec d'autres langages ou des bibliothèques... et le manque de fiabilité des premiers compilateurs et des environnements de programmation. Ces problèmes ont disparu avec le temps, ou ont été résolus par Ada 95 et peuvent être maintenant considérés comme résolus.

Lorsqu'une étude objective est menée, prenant en compte les principes du génie logiciel, les coûts des développements informatiques pris sur l'ensemble du cycle de vie, et les contraintes de fiabilité et de sécurité, elle aboutit toujours à la même conclusion : *Ada est le seul langage qui offre le support nécessaire aux méthodologies modernes de développement*. Le problème est que les facteurs objectifs ne sont souvent pas suffisants pour convaincre les utilisateurs. Le bouche à oreille, la rumeur, de mauvais résultats dus à des essais sommaires sont souvent plus crus que les études formelles. Cette attitude se résume en la formule :

> *«Bien sûr, l'étude a conclu qu'il fallait utiliser Ada... mais mon voisin de palier travaille en C».*

Il reste encore du chemin à parcourir pour obtenir une approche industrielle du développement logiciel !

# Annexes

## A. Le distributeur de boissons

### *A.1. Corps maquettes de la première version*

Cette version du menu comporte une fonction `Choix_Utilisateur` qui renvoie successivement tous les choix possibles, puis déclenche une annulation de commande, enfin lève l'exception `Program_Error` de façon à terminer le programme (le programme en vraie grandeur ne se termine jamais, aussi n'y a-t-il rien de prévu à cet effet)... ce qui permet également de tester notre traite-exception de sécurité.

```
with ADPT; use ADPT;
package body Menu is
   Terminer   : Boolean := False;
   Annuler    : Boolean := False;
   A_Renvoyer : Produit := Produit'First;

   procedure Activer is
   begin
      ADPT_Action("Activation du menu");
   end Activer;

   function Choix_Consommateur return Produit is
   begin
      ADPT_Action("Choix du produit");
      if Terminer then
         raise Program_Error;
      end if;

      if Annuler then
         Terminer := True;
         Annulation.Demande;  -- Provoque l'avortement
      end if;

      if A_Renvoyer = Produit'Last then
         Annuler := True;
         return A_Renvoyer;
      end if;

      A_Renvoyer := Produit'Succ(A_Renvoyer);
      return Produit'Pred(A_Renvoyer);
   end Choix_Consommateur;

   procedure Invalider (Le_produit : Produit) is
   begin
      ADPT_Action("Invalidation du produit " &
                  Produit'Image(Le_Produit));
   end Invalider;

   procedure Revalider (Le_produit : Produit) is
   begin
      ADPT_Action("Revalidation du produit " &
                  Produit'Image(Le_Produit));
   end Revalider;
```

```
   protected body Annulation is
      entry Signalement when Ouverte is
      begin
         if Signalement'Count = 0 then
            Ouverte := False;
         end if;
      end Signalement;

      procedure Demande is
      begin
         if Signalement'Count > 0 then
            Ouverte := True;
         end if;
      end Demande;
   end  Annulation;
end Menu;

with ADPT; use ADPT;
package body Monnayeur is
   procedure Activer is
   begin
      ADPT_Action("Activation du monnayeur");
   end Activer;

   procedure Bloquer is
   begin
      ADPT_Action("Bloquer le monnayeur");
   end Bloquer;

   procedure Attendre(Pour_Produit : Produit) is
   begin
      ADPT_Action("Attente du montant suffisant");
   end Attendre;

   function  Montant_Introduit return Argent  is
   begin
      return 5.0;
   end Montant_Introduit;

   procedure Rendre_Monnaie(Du_produit: Produit) is
   begin
      ADPT_Action("Rendre monnaie sur " &
                  Produit'Image(Du_Produit));
   end Rendre_Monnaie;

   procedure Rembourser is
   begin
      ADPT_Action("Remboursement");
   end Rembourser;
end Monnayeur;

with ADPT; use ADPT;
package body Tarif is
   function Le_Prix (De: Produit) return Argent is
   begin
      return 2.0;
   end Le_Prix;
end Tarif;
```

## A.2. Corps du compteur générique

```
package body Compteur_Protégé_Générique is
   protected Compteur is
      procedure Incrémenter (De : A_Compter);
      function  Valeur_Courante return A_Compter;
      procedure Remise_A_Zéro;
      entry Attendre(Montant : A_Compter);
   private
      entry Faire_La_Queue;
      Accumulateur    : A_Compter := Valeur_Nulle;
      Montant_Attendu : A_Compter := Valeur_Nulle;
   end Compteur;

   procedure Attendre(Valeur : A_Compter) is
   begin
      Compteur.Attendre(Valeur);
   end;

   procedure Incrémenter (De : A_Compter) is
   begin
      Compteur.Incrémenter(De);
   end;

   function  Valeur_Courante return A_Compter is
   begin
      return Compteur.Valeur_Courante;
   end;

   procedure Remise_A_Zéro is
   begin
      Compteur.Remise_A_Zéro;
   end;

   protected body Compteur is
      entry Faire_La_Queue
        when not (Accumulateur < Montant_Attendu) is
      begin
        null;
      end Faire_La_Queue;

      entry Attendre(Montant : A_Compter)
        when Faire_La_Queue'Count = 0 is
      begin
        Montant_Attendu := Montant;
        requeue Faire_La_Queue with abort;
      end Attendre;

      procedure Incrémenter (De : A_Compter) is
      begin
        Accumulateur := Accumulateur + De;
      end;

      function  Valeur_Courante return A_Compter is
      begin
        return Accumulateur;
      end;

      procedure Remise_A_Zéro is
      begin
        Accumulateur := Valeur_Nulle;
      end;
   end Compteur;
end Compteur_Protégé_Générique;
```

## A.3. Corps maquettes pour le monnayeur de deuxième niveau

```
with ADPT; use ADPT;
package body Clapet is
   procedure Ouvrir is
   begin
      ADPT_Action("Ouvrir le clapet");
   end Ouvrir;

   procedure Fermer is
   begin
      ADPT_Action("Fermer le clapet");
   end Fermer;
end Clapet;

package body Monnayeur.Définition_Pièces is
   function Répartir(Montant : Argent)
      return Répartition_Pièces is
      Répartition : Répartition_Pièces := (others => 0);
      Restant     : Argent := Montant;
   begin
      for Type_Pièce in reverse Pièces loop
         while Restant >= Valeurs (Type_Pièce) and
               Nombre_De_Pièces ( Magasin(Type_Pièce)) >
               Répartition (Type_Pièce)
         loop
           Répartition(Type_Pièce) :=
              Répartition(Type_Pièce) + 1;
           Restant := Restant - Valeurs(Type_Pièce);
         end loop;
      end loop;
      if Restant > 0.0 then
         raise Rendu_Impossible;
      else
         return Répartition;
      end if;
   end Répartir;
end Monnayeur.Définition_Pièces;
```

## A.4. Corps des distributeurs d'ingrédients

```
with ADPT;
package body Unité_Fabrication.Distributeur_Poudre is
   procedure Servir_Dose(De : access Instance) is
      use ADPT;
   begin
      ADPT_ACTION("Je distribue poudre : " &
                  Integer_Address'IMAGE(De.Adresse));
   end Servir_Dose;
end Unité_Fabrication.Distributeur_Poudre;

with ADPT;
package body Unité_Fabrication.Distributeur_Liquide is
   procedure Servir_Dose(De : access Instance) is
      use ADPT;
   begin
      ADPT_ACTION("Je distribue liquide : " &
                  Integer_Address'IMAGE(De.Adresse) &
                  " volume " &
                  Volume_Livré'Image(De.Volume) &
                  " cl");
   end Servir_Dose;
end Unité_Fabrication.Distributeur_Liquide;
```

# B. Le paquetage ADPT

## *B.1. Spécification*

```
package ADPT is
   --------------------------- Type complètement indéfini
   type ADPT_Type is private;
   function ADPT_Fonction(Info  : String   := "";
                          Durée : Duration := 0.0)
     return ADPT_Type;
   ADPT_Valeur : constant ADPT_Type;

   --------------------------- Type Enumératif
   type ADPT_Type_Enuméré is (ADPT_Valeur_Enumérée);
   function ADPT_Fonction(Info  : String   := "";
                          Durée : Duration := 0.0)
     return ADPT_Type_Enuméré;

   --------------------------- Type discret
   type ADPT_Type_Discret is new ADPT_Type_Enuméré;
   function ADPT_Fonction(Info  : String   := "";
                          Durée : Duration := 0.0)
     return ADPT_Type_Discret;
   ADPT_Valeur_Discrète : constant ADPT_Type_Discret;

   --------------------------- Type entier
   type ADPT_Type_Entier is range 0..1;
   function ADPT_Fonction(Info  : String   := "";
                          Durée : Duration := 0.0)
     return ADPT_Type_Entier;
   ADPT_Valeur_Entière : constant ADPT_Type_Entier;

   --------------------------- Type flottant
   type ADPT_Type_Flottant is new FLOAT;
   function ADPT_Fonction(Info  : String   := "";
                          Durée : Duration := 0.0)
     return ADPT_Type_Flottant;
   ADPT_Valeur_Flottante : constant ADPT_Type_Flottant;

   --------------------------- Type fixe
   type ADPT_Type_Fixe is new DURATION;
   function ADPT_Fonction(Info  : String   := "";
                          Durée : Duration := 0.0)
     return ADPT_Type_Fixe;
   ADPT_Valeur_Fixe : constant ADPT_Type_Fixe;

   --------------------------- Type tableau
   type ADPT_Type_Tableau is array(ADPT_Type_Discret)
                             of ADPT_Type;
   function ADPT_Fonction(Info  : String   := "";
                          Durée : Duration := 0.0)
     return ADPT_Type_Tableau;
   ADPT_Valeur_Tableau : constant ADPT_Type_Tableau;

   --------------------------- Type article
   type ADPT_Type_Article is
     record
       ADPT_Element : ADPT_Type;
     end record;
   function ADPT_Fonction(Info  : String   := "";
                          Durée : Duration := 0.0)
     return ADPT_Type_Article;
   ADPT_Valeur_Article : constant ADPT_Type_Article;
```

```
   --------------------------- Type accès
   type ADPT_Type_Accès is access ADPT_Type;
   function ADPT_Fonction(Info  : String   := "";
                          Durée : Duration := 0.0)
     return ADPT_Type_Accès;
   ADPT_Valeur_Accès : constant ADPT_Type_Accès;

--------------------------- Type étiqueté
   type ADPT_Type_Etiqueté is tagged
     record
       ADPT_Element : ADPT_Type;
     end record;
   function ADPT_Fonction(Info  : String   := "";
                          Durée : Duration := 0.0)
     return ADPT_Type_Etiqueté;
   ADPT_Valeur_Etiquetée : constant ADPT_Type_Etiqueté;

   --------------------------- Type étiqueté abstrait
   type ADPT_Type_Etiqueté_Abstrait is
     abstract tagged null record;
   -- Les types abstraits n'ont pas de valeurs

   --------------------------- Autres déclarations
   ADPT_Condition : constant BOOLEAN := False;
   ADPT_Exception : exception;

   procedure ADPT_Procédure(Info  : String   := "";
                            Durée : Duration := 0.0);
   procedure ADPT_Actions  (Info  : String   := "";
                            Durée : Duration := 0.0);

   --------------------------- Ajustement du comportement
   type ADPT_Comportement is (Ignorer, Tracer, Piéger);
   ADPT_Comportement_Courant: ADPT_Comportement := Ignorer;

private
   type ADPT_Type is range 0..0;

   ADPT_Valeur           : constant ADPT_Type := 0;
   ADPT_Valeur_Discrète  : constant ADPT_Type_Discret
     := ADPT_Valeur_Enumérée;
   ADPT_Valeur_Entière   : constant ADPT_Type_Entier
     := 0;
   ADPT_Valeur_Flottante : constant ADPT_Type_Flottant
     := 0.0;
   ADPT_Valeur_Fixe      : constant ADPT_Type_Fixe
     := 0.0;
   ADPT_Valeur_Tableau   : constant ADPT_Type_Tableau
     := (ADPT_Valeur_Discrète => ADPT_Valeur);
   ADPT_Valeur_Article   : constant ADPT_Type_Article
     := (ADPT_Element => ADPT_Valeur);

   ADPT_Valeur_Etiquetée : constant ADPT_Type_Etiqueté
     := (ADPT_Element => ADPT_Valeur);
   ADPT_Valeur_Accès     : constant ADPT_Type_Accès
     := new ADPT_Type'(ADPT_Valeur);
end ADPT;
```

## *B.2. Corps*

```
with Ada.Text_IO;
package body ADPT is
   ADPT_Exception_Cachée : exception;
```

```
procedure Message (Nom   : String; Info  : String;
                   Durée : Duration) is
  use Ada.Text_IO;
begin
  delay Durée;
  case ADPT_Comportement_Courant is
  when Ignorer =>
    null;
  when Tracer =>
    if Info = "" then
      Put_Line ("*** Appel de " & Nom & " ***");
    else
      Put_Line ("*** Appel de " & Nom & " : " & Info);
    end if;
  when Piéger =>
    raise ADPT_Exception_Cachée;
  end case;
end Message;

function ADPT_Fonction (Info  : String   := "";
                        Durée : Duration := 0.0)
  return ADPT_Type is
begin
  Message("ADPT_Fonction (type indéfini)", Info, durée);
  return ADPT_Valeur;
end ADPT_Fonction;

function ADPT_Fonction (Info  : String   := "";
                        Durée : Duration := 0.0)
  return ADPT_Type_Enuméré is
begin
  Message("ADPT_Fonction (type énuméré)", Info, durée);
  return ADPT_Valeur_Enumérée;
end ADPT_Fonction;

function ADPT_Fonction (Info  : String   := "";
                        Durée : Duration := 0.0)
  return ADPT_Type_Discret is
begin
  Message("ADPT_Fonction (type discret)", Info, durée);
  return ADPT_Valeur_Discrète;
end ADPT_Fonction;

function ADPT_Fonction (Info  : String   := "";
                        Durée : Duration := 0.0)
  return ADPT_Type_Entier is
begin
  Message("ADPT_Fonction (type entier)", Info, durée);
  return ADPT_Valeur_Entière;
end ADPT_Fonction;

function ADPT_Fonction (Info  : String   := "";
                        Durée : Duration := 0.0)
  return ADPT_Type_Flottant is
begin
  Message("ADPT_Fonction (type flottant)", Info, durée);
  return ADPT_Valeur_Flottante;
end ADPT_Fonction;

function ADPT_Fonction (Info  : String   := "";
                        Durée : Duration := 0.0)
  return ADPT_Type_Fixe is
begin
  Message("ADPT_Fonction (type fixe)", Info, durée);
  return ADPT_Valeur_Fixe;
end ADPT_Fonction;
```

```
   function ADPT_Fonction (Info  : String   := "";
                           Durée : Duration := 0.0)
     return  ADPT_Type_Tableau is
   begin
     Message("ADPT_Fonction (type tableau)", Info, durée);
     return ADPT_Valeur_Tableau;
   end ADPT_Fonction;

   function ADPT_Fonction (Info  : String   := "";
                           Durée : Duration := 0.0)
     return ADPT_Type_Article is
   begin
     Message("ADPT_Fonction (type article)", Info, durée);
     return ADPT_Valeur_Article;
   end ADPT_Fonction;

   function ADPT_Fonction (Info  : String   := "";
                           Durée : Duration := 0.0)
     return ADPT_Type_Accès is
   begin
     Message("ADPT_Fonction (type accès)", Info, durée);
     return ADPT_Valeur_Accès;
   end ADPT_Fonction;

   function ADPT_Fonction (Info  : String   := "";
                           Durée : Duration := 0.0)
     return ADPT_Type_Etiqueté is
   begin
     Message("ADPT_Fonction (type étiqueté)", Info, durée);
     return ADPT_Valeur_Etiquetée;
   enàd ADPT_Fonction;

   procedure ADPT_Procédure (Info  : String   := "";
                             Durée : Duration := 0.0) is
   begin
     Message("procédure à définir", Info, Durée);
   end ADPT_Procédure;

   procedure ADPT_Actions (Info  : String   := "";
                           Durée : Duration := 0.0) is
   begin
     Message("action à définir", Info, Durée);
   end ADPT_Actions;
end ADPT;
```

## C. Modèle de fiche de composant

**Nom de base de l'unité Ada**

### Identification

*Cette rubrique résume les éléments communs à toutes les implémentations et à toutes les variantes. Elle doit permettre de décider rapidement si le composant est susceptible de répondre au besoin.*

Description

Brève description du rôle du composant logiciel.

Mots-clés

Mots clés référencés dans la base de donnée de composants logiciels et utilisés pour l'index.

Caractérisation

Unité : Paquetage, procédure, fonction, générique.
Genre : Machine abstraite, type de donnée abstrait...
Liaisons : Indépendant, surcouche, encapsulation, famille...

### Variantes

Liste des variantes disponibles, avec taxonomie Booch/Berard. Limites et contraintes d'implémentations pour chaque variante.

### Disponibilité

*Systèmes de compilation* : (Compilateur, hôte, cible) sous lesquels le composant est disponible (compilé, validé, testé). Matrice des versions (Variantes, systèmes).
*Accès* : Pour chaque système de compilation : Disponibilité en source ou sous forme de sous-bibliothèque. Contraintes d'accès (droits d'accès système nécessaires, composant réservé, copyright et droits d'accès, run-time éventuels, autorisations nécessaires). Procédures à suivre pour accéder au composant (information au gestionnaire de composants).

### Historique

Dates des principales releases. Nombre d'utilisations précédentes. Préciser si en cours de développement, bêta-test.

## Spécifications

### Eléments génériques et ajustement de comportement

Description des éléments apparaissant en paramètres génériques. Rôle et invariants supposés. Conditions de bon fonctionnement. Unités séparées redéfinissables par l'utilisateur.

### Eléments principaux

Spécification Ada des éléments indépendants de la variante, par groupe d'éléments logiquement reliés. Spécification abstraite du rôle de chacun des éléments. Invariants. Vérifications effectuées. Exceptions susceptibles d'être levées.

### Eléments annexes

Idem, pour les éléments supplémentaires dépendant de la variante.

## Implémentations

*Eléments de l'implémentation importants pour l'utilisateur.*

### Elaboration

Pour chaque variante : `pragma Elaborate_All` (ou `Elaborate`) nécessaires au bon fonctionnement du composant. Toutes dépendances à l'initialisation.

### Accès physique

Pour chaque système de compilation : Localisation physique du composant (machine, répertoire, nom de fichier ou d'unité si plusieurs variantes). Toutes informations nécessaires à l'utilisation pratique du composant

### Algorithme

Précisions sur l'algorithme utilisé, s'il est important pour l'utilisateur.

### Eléments sensibles utilisés

Pour chaque variante : préciser si l'implémentation utilise les points flottants (à cause de la nécessité parfois de la présence d'un co-processeur), le tasking, l'allocation dynamique.

## Performances

Pour chaque variante : complexité de l'algorithme. Conditions d'exécution du test standard. Temps d'exécution du programme de test standard en configuration nominale, par variante et système de compilation.

## Autres informations

Toutes autres informations nécessaires : utilisation d'éléments non standard du langage pouvant présenter un risque de changement en cas de modification du système de compilation. Particularités et justification d'exceptions à la sémantique générale pour certaines implémentations. Dépendances particulières à l'implémentation.

# Bibliographie

Mettre à jour (nouveauté, publications JPR, UML, Java). Faire des références.


[Afn87] AFNOR. *Langages de programmation – Ada, norme NF EN 28652*. Eyrolles, Paris, 1987.

[Bar88] G.J.P. Barnes. *Programmer en Ada*. InterEditions, Paris, 1988.

[Bar92] M. Bari. «Une méthode d'analyse et de conception orientée objet de systèmes d'information actifs», *Thèse de Doctorat de l'Université Paris VI*. Université Paris VI, Paris, 1992.

[Bau91] B. Bauer et J. Poudret. «Ada dans l'Aéronautique à SEXTANT Avionique», *Actes de la Conférence Internationale Francophone Ada-France 91*. AFCET, Paris, 1991.

[Ber87] E.V. Berard et J. Margono. «A modified Booch's taxonomy for Ada generic data-structure components and their implementation», *Ada components : libraries and tools. Proceedings of the Ada-Europe Conference 1987*. Cambridge University Press, Cambridge, 1987.

[Ber89] J.-M. Bergé, L.-O. Donzelle, V. Olive, J. Rouillard. *Ada avec le sourire*. Presses Polytechniques Romandes – Collection CNET-ENST, Lausanne, 1989.

[Boo87] G. Booch. *Software Components with Ada*. Benjamin Cummings Publishing Company, Menlo Park, 1987.

[Boo84] G. Booch, *Software Engineering with Ada*, Benjamin Cummings Publishing Company, Menlo Park, 1984.

[Boo88] G. Booch. *Ingénierie du logiciel avec Ada* InterEditions, Paris, 1988 (traduction française de J.P. Rosen).

[Boo91] G. Booch. *Object Oriented Design with Applications*. Benjamin Cummings Publishing Company, 1991

[Bry87] Doug Bryan. «Dear Ada», *Ada letters* Vol. VII, n° 1. ACM, New York, 1987.

[Buh84] R.J.A. Buhr. *System Design with Ada*. Prentice Hall, Englewood-Cliffs, N.J., 1984.

[Buh89] R.J.A. Buhr. *System Design with Machine Charts: a CAD Approach with Ada Examples*. Prentice Hall, Englewood-Cliffs, N.J., 1989.

[Bur85] A. Burns. *Concurrent Programming in Ada* . Cambridge University Press, Cambridge, 1985.

[Car91] F. Caron et X. Cusset. «Comparaison a posteriori de développements en Ada et en C», *Actes de la Conférence Internationale Francophone Ada-France 91*. AFCET, Paris, 1991.

[Chen76] P.P.S. Chen. «The Entity-Relationship Model : Towards a Unified View of Data», *ACM Transactions on Data Base Systems*, Vol 1, n°1. ACM, New York, 1976.

[Cla80] L. A. Clarke, J. C. Wilden et A. L. Wolf, «Nesting in Ada Programs is for the Birds». *SIGPlan Notices* Volume 15, N° 11, ACM, New York, Novembre 1980.

[Des37] R. Descartes. *Discours de la méthode*. Leyde, 1637.

[Dod87] DoD. *Software Design Document, DOD-STD-2167A*. NTIS, 1987.

[Fav91] J. Favaro. «What Price Reusability ? A Case Study», *Proceedings of the First Symposium on Environments and Tools for Ada.* Ada Letters Volume XI n° 3. ACM, New York, 1991.

[Gau89] M. Gauthier. *Ada : un apprentissage*. Dunod, Paris, 1989.
[Gau92] M. Gauthier. «Tout n'est pas objet», *Génie Logiciel* n°26. EC2, Nanterre, mars 1992.
[Gau94] M. Gauthier. «Contribution aux fondements de la programmation par objets», *Document d'habilitation à diriger des recherches*. Université Blaise Pascal, Clermont-Ferrand, 1994.
[Goo88] J. Goodenough. «Using Exceptions : Some Design Issues». *Ada Technology for Command & Control*. TRW Federal Systems Group, Fairfax, VA, 1988.
[Hei87] M. Heitz. «HOOD: une méthode de conception hiérarchisée orientée objets pour le développement de gros logiciels techniques et temps réel», *Actes des journées Ada-France 1987*, BIGRE n°57, décembre 1987.
[Hoa72] C. A. R. Hoare. «Notes on Data Structuring», *in* O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, 1972.
[Hof85] D. Hofstadter. *Gödel, Escher, Bach : les Brins d'une Guirlande Eternelle*. InterEditions, Paris, 1985.
[Hoo93] HOOD User Group. *Hood Reference Manual 3.1*, Masson-Prentice Hall, Paris, 1993.
[Iso87] ISO. *Ada Programming Language, ISO Standard 8652/1987*. ISO, Genève, 1987
[Iso94] ISO. *Database Programming Language -- The SQL Ada Module Description Language SAMeDL, ISO/IEC 12227:1994 (E)*. ISO, Genève, 1994
[Iso95] ISO. *Ada Programming Language, ISO Standard 8652/1995*. ISO, Genève, 1995
[Jea93] C. Jean-Pousin et S. Barbey. «Implementing Associations with Ada». *Actes du congrès «Génie Logiciel et ses applications»*. EC2, Nanterre, 1993.
[Kru86] P. Kruchten. *Une machine virtuelle pour Ada: architecture*, thèse de Docteur de l'ENST, ENST-86E010, ENST, Paris, 1986.
[Kru89] P. Kruchten. «Traitement des erreurs dans de grands systèmes utilisant la conception par objets», *Genie Logiciel et Systèmes Experts*, Vol 17. EC2, Nanterre, 1989.
[Kru90] P. Kruchten. «Error Handling in Large Object-Based Ada Systems», *Ada Letters*, Vol X, n° 7. ACM, New York, septembre 1990.
[Lai89] M. Lai. "Bilan d'utilisation de Hood et de Ada sur un projet développé au groupe RAC/GERDSM", *Actes des journées Ada-France 1989*, AFCET, Paris, 1989
[Lai91] M. Lai. *Conception Orientée Objet, pratique de la méthode HOOD*. Dunod, Paris, 1991
[Lan91] J.-A. Laneyrie. «Réflexions sur quelques aspects humains de la qualité du logiciel et des objets en programmation», *Afcet-Interfaces* n°103/104 (n° spécial «Objectif Objets !»), AFCET, Paris, 1991.
[Leb94] R. Lebib. «Incidences de la Mise en Œuvre des Concepts du Géniel Logiciel à travers l'Utilisation du Langage de Programmation Ada sur la Productivité en Informatique de Gestion», *thèse de Doctorat d'Université.* Université Paris IX Dauphine, Paris, 1994.
[Ler90] P. Leroy et P. Kruchten. «Gestion de configuration dans l'atelier Rational», *Actes de la Conférence Internationale Francophone Ada-France 90*, AFCET, Paris, 1990.
[Lin35] C. Linné. *Systema naturae*, Uppsala, 1735.
[Luc90] D. Luckham. *Programming with Specifications : An Introduction to Anna, A Language for Specifying Ada Programs*. Springer Verlag, Berlin, 1990.
[Mas89] G. Masini et alt. *Les langages à objets*. InterEditions, Paris, 1989.
[Mey88] B. Meyer. *Object Oriented Software Construction*. Prentice Hall, New York, 1988.
[Mey90] B. Meyer. *Conception et programmation par objets, pour du logiciel de qualité.* InterEditions, Paris, 1990.
[Mil56] G. A. Miller. «The Magical Number Seven, Plus or Minus Two», *The Psychological Review*, vol.63, n° 2, mars 1956.
[Moi91] T. Moineau. «ROSE-ADA : an instance of the ESF-ROSE System to reuse Ada code», *Proceedings of SEE'91 conference*, Aberystwyth, Avril 1991.
[Mor82] P. Morrison, P. Morrison, C. Eames : *Powers of Ten*. Scientific American Library, New York, 1982.

[Par71] D. L. Parnas. «On Criteria To Be Used in Decomposing Programs Into Modules», *report CMU-CS-71-101*, Computer Science Department, Carnegie-Mellon University, Pittsburgh P.A., février 1971.

[Pid93] H. Pidault. «Graphics Development in Ada», *Ada-Europe 93*, Lecture Notes in Computer Science n° 688. Springer Verlag, Berlin, 1993.

[Pit87] G. Pitette. «Adlog : Comment introduire des composants déductifs dans une application Ada», *Génie Logiciel et Systèmes Experts,* n°9. EC2, Nanterre, novembre 1987 .

[Pit88] G. Pitette. «Adlog : an Ada Components Set to Add Logic to Ada», *Ada in Industry, Proceedings of Ada-Europe International Conference*. Cambridge University Press, Cambridge, 1988.

[Pri91] R. Prieto-Diaz. «Implementing Faceted Classification for Software Reuse». *Communications of the ACM*, Vol. 34 n° 5, ACM, New York, mai 1991.

[Rei87] D. Reifer. «Ada's Impact : A Quantitative Assesment», ACM 1987, 0-89791-243-8/0012/0001.

[Rei89] D. Reifer. «Estimer l'impact d'Ada sur les coûts d'exploitation et de maintenance du logiciel», *La Lettre Ada*, n°24. EC2, Nanterre, août 1989.

[Rem94] C. Rémy. «La réutilisation : des outils, mais surtout une organisation», *01 Informatique* n°1329, 21/10/1994

[Rig87] E. Riguidel. *La méthode Mac_Adam, manuel de référence*. Thomson CSF SDC, Meudon-la-Forêt, juin 1987.

[Rig89] E. Riguidel. «La chaîne Mac_Adam: d'un modèle abstrait à une architecture concrète», *Actes des journées Ada-France 1989*, AFCET, Paris, 1989

[Ros85] J-P. Rosen. «SETL : un langage de très haut niveau pour le prototypage», Journées d'étude AFCET «Nouveaux langages pour le génie logiciel», *BIGRE + Globule* n°45, Paris, octobre 1985.

[Ros86] J-P. Rosen. *Une machine virtuelle pour Ada: le système d'exploitation*, thèse de Docteur de l'ENST, ENST-86E013, ENST, Paris, 1986.

[Ros87] J-P. Rosen. «In defense of the USE clause», *Ada Letters*, Vol. VII, No. 7, ACM, New York, novembre/décembre 1987.

[Ros89] J-P. Rosen. «Conception Orientée Objets: des méthodes pour affronter la complexité», *Le Monde Informatique* n° 376, Paris, 10 Juillet 1989.

[Ros90a] J-P. Rosen. «Pensées, proverbes et citations», *AFCET Interfaces* n° 90, AFCET, Paris, avril 1990.

[Ros90b] J-P. Rosen. «Pour une définition méthodologique de la notion d'orienté objet», AFCET Interfaces, n°96, AFCET, Paris Octobre 1990.

[Ros91] J-P. Rosen. «Introduction au monde des objets», *AFCET Interfaces*, n°103-104 spécial "Objectif Objets", AFCET, Paris mai/juin 1991

[Ros92a] J-P. Rosen. «Ada 9X : une évolution, pas une révolution», *Techniques et science informatiques*, Volume 11 n°5, HERMES, Paris,1992.

[Ros92b] J-P. Rosen. «What Orientation Should Ada Objects Take?», *Communications of the ACM*, Volume 35 n° 11, ACM, New York, 1992.

[Ros94] J-P. Rosen, N. Kettani. «Apport d'Ada 9X aux paradigmes orientés objet», *Acte des Journées Internationales sur les Nouveautés en Génie Logiciel*, C3ST, Courbevoie, 1994

[Ros95a] J-P. Rosen. «Ada 95 pour le temps réel et les applications distribuées», *Séminaire de la conférence Real-Time Systems '95*, BIRP, Paris 1995.

[Ros95b] J-P. Rosen. «A Naming Convention for Classes in Ada 9X», *Ada Letters*, Vol. XV, No. 2, ACM, New York, mars/avril 1995.

[Ros97] J-P. Rosen et le HOOD User Group. *HOOD: An Industrial Approach for Software Design*, HOOD User Group, Bruxelles 1997.

[Ros99] J-P. Rosen. «Ada pour développer sur Internet», actes de la conférence Ada-France, Brest, 16 septembre 1999.
[Ros02] J-P. Rosen. «Ada, Interfaces and the Listener Paradigm», Reliable Software Technologies - Ada-Europe 2002, Proceedings , Vienna, Austria, June 17-21, 2002, Lecture Notes in Computer Science, vol. 2361, Springer-Verlag, 2002.
[Ros03] J-P. Rosen. «Experiences in Developing a Typical Web/Database Application», proceedings of the ACM SIGAda Annual International Conference (SIGAda 2003) , ACM Press, ACM order number 825030, San Diego, USA, December 7-11, 2003.
[Rum94] J. Rumbaugh et alt. *OMT : Modélisation et conception orientées objet*, Masson, Paris, 1994
[Sch86] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, E. Schonberg. *Programming with Sets – An Introduction to SETL*. Springer Verlag, Berlin, 1986.
[Shl88] S. Shlaer , J. Mellor. *Object Oriented Systems Analysis, Modeling the World in Data*. Yourdon Press Computing Series, Englewood Cliffs, N.J.,1988.
[Sol91] J. Solderitsch. «Working Group Report, Library and Representation Subgroup of Methods and Tools for Design, Specification and Reuse», *Proceedings of the First Symposium on Environments and Tools for Ada,* Ada Letters Volume XI n° 3, ACM, New York, 1991.
[Spc92] Software Productivity Consortium. *Ada Quality and Style: Guidelines for Professional Programmers*, Version 2.01.01, SPC, décembre 1992.
[Tar93] Tartan. «Ada Outperforms C on C40 for Itek», *On Target* Vol 1, n°1, Tartan, été 1993.
[Tra89] W.J. Tracs et S. Edward. «Implementation Working Group Report», *Reuse in Practice Wokshop*. Software Engineering Institute, Pittsburgh PA, 1989.
[War85] P. Ward et J. Mellor. *Structured Developement for Real-Time Systems*. Prentice-Hall, Englewood-Cliffs, N.J., 1985.
[Wel83] A.J. Wellings, D. Keefe, G.M. Tomlinson. «A Problem with Ada and Resource Allocation», *Ada Letters*, Vol. III, No. 4, ACM, New York, juillet/août 1983.
[You79] E. Yourdon et L. Constantine. *Structured Design*. Yourdon Press, Englewood-Cliffs, N.J., 1979.

# Glossaire

Ce glossaire reprend les termes principaux dans le but de faciliter la lecture du livre, aussi avons-nous préféré donner des définitions simples et concrètes, aux dépens parfois de la rigueur d'une définition plus formelle.

**Agrégat** : valeur d'un type structuré (tableau ou article), fournie sous la forme d'une liste de valeurs entre parenthèses.

**Attente active** : boucle d'un programme n'effectuant rien d'autre que tester l'arrivée d'un événement.

**Attribut** : en termes de programmation orientée objet : structures de données faisant partie de la définition d'une classe. En termes Ada : construction syntaxique spéciale permettant de récupérer de l'information liée à un type, une variable, un sous-programme, etc.

**Classe** : en Ada : ensemble des types dérivés directement ou indirectement d'un type étiqueté. La classe d'origine T se désigne en Ada par T'Class. Plus généralement dans le monde orienté objet : abstraction d'un ensemble d'objets possédant des propriétés communes.

**Condition de course** : erreur d'un système parallèle due au fait que deux tâches effectuent simultanément une action à un moment indésirable.

**Elaboration** : processus par lequel une déclaration devient utilisable. En Ada, ceci peut nécessiter l'exécution de code ; l'élaboration est aux déclarations ce que l'exécution est aux instructions.

**Encapsulation** : regroupement dans une même entité de structures de programmes et de structures de données logiquement reliées.

**Facette** : ensemble de propriétés constituant une vue particulière des propriétés d'un objet ; un objet peut posséder plusieurs facettes. Par exemple, un cercle peut comporter une facette d'objet graphique et une autre d'objet géométrique.

**Garde** : condition booléenne contrôlant une entrée de tâche ou d'objet protégé. L'entrée ne peut être appelée que si la garde est `True`.

**Générique** : forme d'unité de programme paramétrable, permettant d'écrire des modules réutilisables sur divers types de données.

**Graphe de dépendances** : v. *topologie de programme*.

**Héritage** : mécanisme par lequel un type dérivé possède les attributs et les méthodes du type dont il est dérivé.

**Instance** : structure de donnée appartenant à une classe et représentant un objet abstrait. Une instance est à une classe ce qu'une variable est à un type.

**Instanciation** : création d'une unité de programme normale à partir du modèle constitué par un générique.

**Liaison dynamique** : propriété d'un appel de sous-programme (méthode) qui ne peut être résolu à la compilation, mais où le sous-programme appelé dépend de l'instance à laquelle il s'applique à l'exécution.

**Méthode** : en programmation orientée objet, structures de programme (opérations) faisant partie de la définition d'une classe.

**Nom complet** : nom simple d'une entité, précédé par le nom de l'entité dans laquelle elle a été déclarée (les deux noms sont séparés par un point).

**Nom simple** : identificateur qui a servi à déclarer une entité.

**Plan d'abstraction** : ensemble constitué de l'implémentation d'un objet *à concevoir* et des spécifications des objets *utilisés* nécessaires à sa réalisation.

**Point fixe** : représentation informatique des nombres réels fournissant une erreur absolue constante.

**Point flottant** : représentation informatique des nombres réels fournissant une erreur relative constante.

**Polymorphisme** : propriété d'une variable susceptible de contenir à l'exécution des valeurs de types différents, mais appartenant à une même classe. La variable peut donc contenir plusieurs «formes» de valeurs.

**Réentrance** : propriété d'un sous-programme qui peut être appelé simultanément par plusieurs tâches.

**Sémantique de référence** : propriété d'un type de donnée abstrait pour lequel l'affectation `X := Y` a la signification : «X et Y désignent la même instance».

**Sémantique de valeur** : propriété d'un type de donnée abstrait pour lequel l'affectation `X := Y` a la signification : «le contenu de Y est transféré dans X».

**Sous-programme abstrait** : sous-programme déclaré avec le mot **`abstract`**, qui ne possède pas de corps. On ne peut appeler un sous-programme abstrait, il doit être redéfini par les types qui en héritent.

**Sous-programme primitif** : sous-programme déclaré dans la même spécification de paquetage que le type d'un de ses paramètres. Seuls les sous-programmes primitifs sont hérités lors de la dérivation du type.

**Suite de validation** : ensemble de programmes de tests destinés à vérifier le bon fonctionnement d'un composant logiciel.

**Surcharge** : possibilité d'attribuer le même nom à plusieurs sous-programmes s'ils diffèrent par le nombre ou le type de leurs arguments ou par le type de la valeur renvoyée pour les fonctions.

**Topologie de programme** : organisation des relations de dépendance logique entre modules d'un programme. La représentation de la topologie de programme est le graphe de dépendances.

**Type abstrait** : type étiqueté déclaré avec le mot **`abstract`**. On ne peut déclarer d'objet ni créer de valeurs d'un type abstrait. Un type abstrait peut déclarer des sous-programmes abstraits.

**Type contrôlé** : type dérivé du type étiqueté `Finalization.Controlled` (ou `Finalization.Limited_Controlled`). Il est possible de définir l'initialisation et la finalisation de tels types, ainsi que l'affectation pour ceux qui ne sont pas limités.

**Type de donnée abstrait** : type défini par un ensemble de *valeurs* et muni d'un ensemble *d'opérations*. Ces valeurs sont *abstraites* parce que la vue qu'en a l'utilisateur est indépendante de la représentation interne adoptée.

# Index

# D



# E

# F



# G



# H



# I

# L



# M



# N



# O



# P

# R



# S



# T

# U



# V