République Algérienne Démocratique et Populaire
Ministère de L'enseignement Supérieur et de la Recherche Scientifique
Université Abbes Laghrour khenchela

Faculté des sciences et de la technologie
Département Mathématiques et informatique

Cours

Pour les étudiants de la 3ème année licence
Systèmes informatiques

Paradigmes de programmation

Dr. Abderrahim Siam

Le présent manuscrit est un support de cours de la matière « Paradigmes de programmation ». Il est destiné essentiellement aux étudiants de la 3ème année licence informatique. A ce stade de la formation, les étudiants sont bien familiarisés avec les concepts de base de la programmation dite impérative et celle orientée objets. D'autres paradigmes de programmation sont à découvrir à travers les différents chapitres constituant ce cours. La manière avec laquelle nous avons approché les différents paradigmes de programmation consiste à les situer dans un cadre générale de l'évolution de la programmation depuis le début de son histoire tout en insistant sur les concepts clés de chaque paradigme, ses pragmatiques et pratiques de programmation, ses innovations importantes ainsi que ses limites ouvrant les portes pour l'exploration de nouvelles voies donnant naissances à d'autres paradigmes de programmation .

Tables des matières

Chapitre IV : Les paradigmes : composants, services et agents.

Chapitre V : La programmation orientée aspect

Chapitre VI : La programmation fonctionnelle

Chapitre VII : La programmation logique

Chapitre VII : Tendances multi-paradigme

Chapitre I :

Introduction à l'étude des paradigmes de programmation

1 Introduction

Le développement de tout logiciel passe presque inévitablement par une phase de codage dans laquelle en utilisant des langages de programmation, les programmeurs implémentent les constituantes du logiciel sujet de développement et qui sont décrites et spécifiées dans les phases de développement antérieures notamment les phases de conceptions préliminaires et détaillées. Les langages de programmation utilisés dans la phase de codage sont généralement des langages dits de haut niveau ou évolués. Contrairement aux langages machine et langages d'assemblage qui sont liés directement aux caractéristiques physiques des machines cibles sur lesquelles les programmes tournent, les langages évolués proposent des abstractions sur les détails physiques des machines pour lesquelles les programmes sont destinés. Ainsi, les langages de programmation doivent offrir aux programmeurs des constructions en mesure de supporter efficacement les différentes manières de penser et les différents concepts mis en jeu lors de la conception des logiciels indépendamment des contraintes liées aux aspects physiques des ordinateurs.

Aujourd'hui, la communauté génie logiciels propose plusieurs manières de penser, de techniques et de méthodes de conception pour concevoir et développer des logiciels. Quand un ensemble de techniques et de méthodes de conception et de développement partagent une même philosophie et utilisent les mêmes concepts de la même manière ou de manières proches, cet ensemble de méthodes forment un courant ou un *paradigme* de développement. Par exemple, les méthodes de conception OMT, OBJECTORY, et OOAD partagent la philosophie de la conception orientée objet et manipulent les mêmes concepts tel que : *objet*,

classe,…etc. Puisque les langages de programmation sont des outils logiciels qui permettent la mise en œuvre des solutions conçues selon des points de vue qui dépendent des approches de développement, il est naturel de trouver plusieurs familles de langages de programmation. Les langages de programmation appartenant à une même famille ou *paradigme* supportent de manières similaires ou proches les unes des autres les concepts de base du paradigme de programmation auquel ils appartiennent. La diversité des approches de conception et de développement de logiciels n'est pas la seule raison derrière l'existence de plusieurs familles de langages de programmation, des besoins spécifiques dans quelques domaines comme l'intelligence artificielle ont poussé la naissance de nouvelles manières de programmer et par la suite de nouvelles familles de langages de programmation.

Les langages de programmation ont un effet profond quant à la manière avec laquelle les programmeurs agissent pour formuler des solutions aux problèmes. Les différents paradigmes imposent différents styles de programmation, d'autant plus, ils changent la façon dont le programmeur adresse les algorithmes. Dans ce cours, notre objectif consiste à étudier les différents paradigmes de programmation dans l'objectif d'élargir vos horizons pour découvrir d'un côté les différentes manières de penser en programmation, et l'ensemble de structures de programmation élégantes et expressives inventées au fil des 65 dernières années.

2 Paradigmes de programmation

Le concept de paradigme peut être défini d'une manière générale comme un point de vue particulier sur la réalité, un angle d'attaque privilégié sur une classe de problèmes, un état d'esprit, une manière de voir les choses, un modèle cohérent de vision du monde qui repose sur des bases bien définies, une forme de rail de la pensée dont les règles ne doivent pas être confondues avec celles d'un autre paradigme,.., etc.

Faisons une liaison entre cette vision de paradigme et les langages de programmation. Les langages de programmation permettent de décrire des comportements à un ordinateur. Ils décrivent des calculs de façon plus abstraite qu'en langages machines. Chaque langage de programmation propose des constructions permettant de faire une programmation supportant une façon particulière pour décrire des comportements et des calculs. L'ensemble des langages de programmation qui supportent une même philosophie de représentation et de description des comportements et des calculs forment une famille de langages de programmation que l'on appelle un *paradigme*. Par exemple, les langages de programmation Pascal, C, et Fortran permettent de faire le même style de programmation, ils appartiennent au même paradigme.

Chaque paradigme de programmation privilégie un ensemble particulier de stratégies d'analyse et de descriptions. Chacun impose une approche, un point de vue particulier sur tout problème. Certains types de problèmes se traitent plus facilement selon un certain

paradigme. Il existe plusieurs paradigmes de programmation tels que : la programmation *impérative*, la programmation *orientée objet*, la programmation *fonctionnelle*, la programmation *logique*, la programmation *concurrente* et la programmation par *acteurs*. Il existe également d'autres paradigmes de programmation que nous pouvons qualifiés d'avancés comme la programmation *à base de composants*, celle *orientée services*, la programmation *orientée aspects*, la programmation par *agents*,… etc. Nous étudierons dans ce cours chacun de ces paradigmes de programmation en mettant l'accent sur les concepts de base de chaque paradigme ainsi que ses points forts et ses limites.

Par exemple, examinons brièvement deux paradigmes de programmation, que vous connaissez, dans ce qu'ils sont et selon les stratégies de résolution de problèmes qu'ils supportent. Il s'agit du paradigme impératif et celui de la programmation orientée objet. Initialement les concepts de la programmation impérative (sans tenir compte des concepts avancés introduits dans les langages impératifs modernes comme ADA) sont ceux qui décrivent l'architecture et le fonctionnement des ordinateurs traditionnels. Une mémoire divisée en cellules individuellement adressables contenant de façon disjointe des instructions et des données ; Une instruction à la fois est exécutée par un processeur unique ; Les données sont examinées et modifiées une à la fois, de façon séquentielle ; Certaines instructions ont pour effet de déterminer quelle sera la prochaine instruction à exécuter. Les langages qui sont des descendants de Fortran (Basic, Pascal, C, etc) supportent ce paradigme. Il en est ainsi pour les langages binaires et assembleurs, quoique de façon moins abstraite. Les restrictions qu'impose ce paradigme ont leur origine uniquement dans les contraintes émanant de l'architecture de la machine. Le programmeur doit percevoir tout problème en termes de séquences d'opérations modifiant une à la fois des données de tailles fixes. Le seul avantage qu'offre ce paradigme est qu'il permet de concevoir des programmes très efficaces. Il est autrement très peu "naturel" et impose des contraintes qui sont à priori plus celles de la machine que celles du problème à résoudre.

Bien qu'il soit parfois "naturel" de percevoir un problème en termes de relations entre des individus, notre attention semble très souvent se porter d'abord sur les objets peuplant un problème et ensuite sur leurs interactions. Le paradigme orienté-objet utilise l'objet comme principe unificateur. Un objet a un état qui est l'ensemble de ses propriétés. Un objet a aussi un comportement : un ensemble de réactions (ou méthodes) qui peuvent être invoquées par d'autres objets via l'envoi de messages. Un objet préserve son intégrité en ne permettant à aucun autre objet de le modifier directement. Un objet peut seul se modifier, à la demande d'un autre possiblement. Les propriétés communes d'un type d'objet (attributs et réactions) sont décrites dans une classe et sont partagés par tous les membres de cette classe. Si cette classe a une ou plusieurs surclasses, ses membres héritent aussi les propriétés des surclasses.

Programmer avec un langage orienté-objet, tel C++ ou Java, amène le programmeur à identifier les "*acteurs*" qui composent un problème, puis à déterminer ce qu'est et ce que doit

savoir chaque acteur. En regroupant les aspects communs puis en les spécialisant, le programmeur établit une hiérarchie de classes. Puis, il cherche à décrire quels échanges de messages entre les acteurs produiront les comportements voulus. Décrire les propriétés et le savoir-faire d'un objet, généraliser ses attributs lors de la création de surclasses sont des habiletés qui dépassent le contexte de la programmation. Qu'elles soient perçues comme problématiques lors des activités de programmation indépendamment d'un langage lui-même, ce sont des habiletés qui sont en soi difficiles à maîtriser.

S'exposer à plus d'un paradigme ne se fait pas sans difficulté. S'initier à un paradigme de programmation très différent de celui qu'on maîtrise demande qu'on redevienne un novice. L'effort intellectuel qu'exige l'acquisition d'un nouveau mode de conception et de représentation est considérable et requiert une longue période de maturation. Les habiletés et les techniques acquises dans la pratique d'un autre paradigme font interférence.

Cet effort de transition, bien que considérable, tend à provoquer une réflexion de niveau méthodologique. Les anciennes habitudes de résolution de problèmes doivent être reconsidérées, et pour cela il faut être en mesure de les représenter afin de les évaluer objectivement. Il est considérablement difficile de réfléchir à propos de stratégies de modélisation informatique si on ne connaît qu'un seul paradigme de programmation. L'étude des différents paradigmes de programmation a plusieurs avantages tels : (*a*) augmenter la capacité d'exprimer des idées complexes puisque cette capacité est liée à une bonne connaissance des caractéristiques linguistiques de programmation ; (*b*) Différentes tâches informatiques demandent des caractéristiques linguistiques de programmation différentes ; (*c*) La connaissance des paradigmes de programmation permet une acquisition plus aisée de nouveaux langages ; (*d*) Une compréhension de la façon dont les langages sont implémentes permet un meilleur usage de ces derniers ; (*e*) La connaissance de différents paradigmes de langages de programmation permet une création plus aisée de nouveaux langages ; (*f*) Avec une meilleure connaissance des paradigmes et des langages de programmation, de meilleurs choix peuvent être faits pour choisir le paradigme et le langage de programmation les plus appropriés pour chaque type de problèmes.

3 Etudes des langages de programmation

Les premiers langages de programmation ont été développés durant les années 1950s. Et depuis, les langages de programmation présentent un domaine dans lequel les spécialistes n'ont pas cessé de produire et de proposer de nouveaux concepts informatiques et de développer de plus en plus des langages de programmation qui combinent de plus en plus : *le pouvoir expressif, la simplicité* et *l'efficacité.* L'étude des langages de programmation vise principalement d'analyser les langages de programmation existants dans l'objectif de spécifier et de concevoir de nouveaux langages et de les implémenter par la suite. L'étude des langages de programmation tient compte des aspects syntaxiques, sémantiques et pragmatiques. Les aspects syntaxiques concernent la forme des programmes où l'accent est

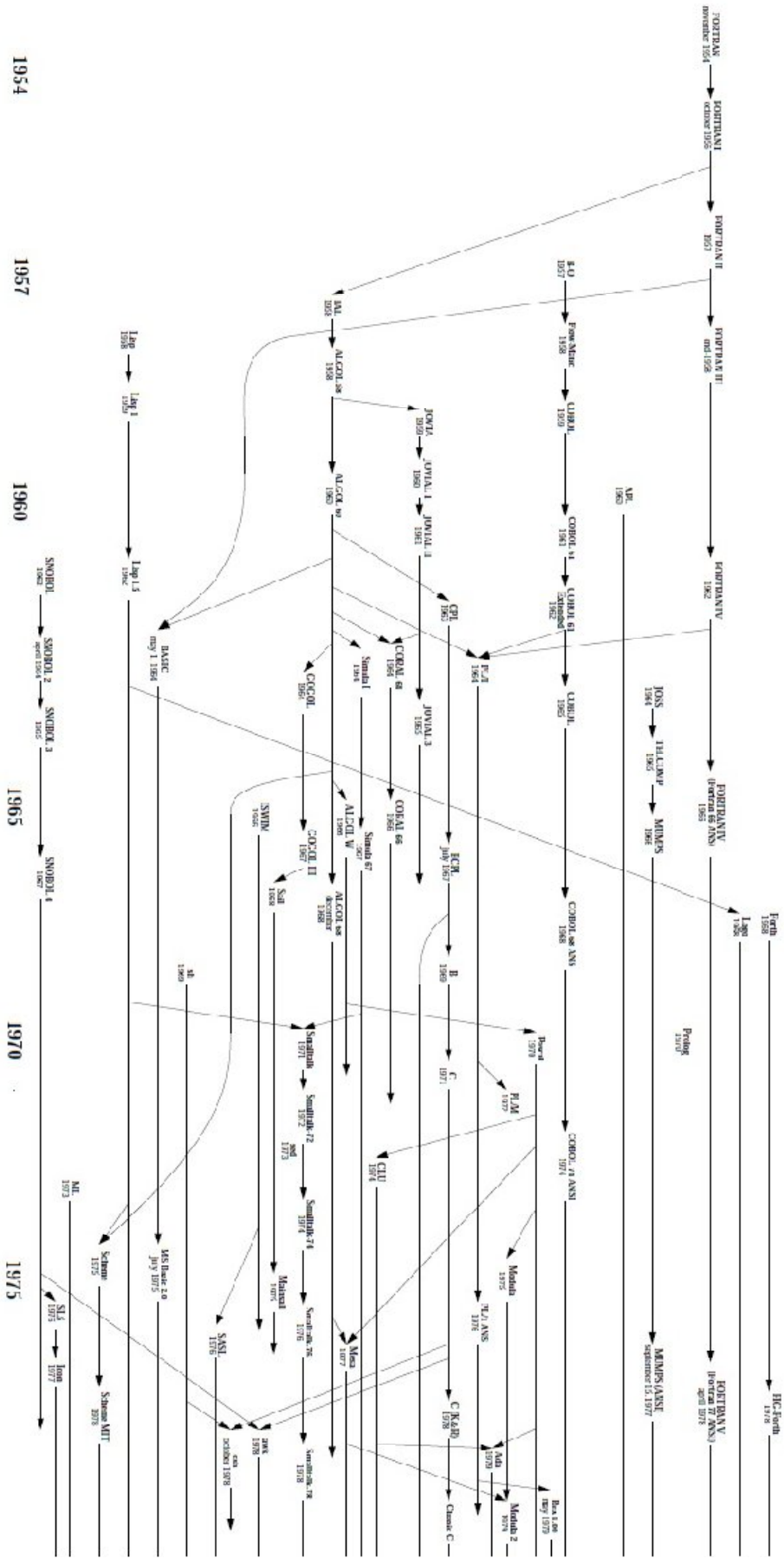
mis sur comment adapter les expressions, les commandes, les déclarations et tout autres éléments pour mettre les programmes sous de bonnes formes. Ces aspects influencent sur la manière dont les programmes sont écrits par les programmeurs, la manière dont les programmes sont lus par d'autres programmeurs et la manière dont les programmes sont compilés sur des ordinateurs. Les aspects sémantiques mettent l'accent sur : le comportement attendu d'un programme supposé bien formé quand celui-ci est exécuté sur un ordinateur. Ces aspects ont un impact sur la manière dont les programmes sont composés par les programmeurs, la manière dont les programmes sont compris par d'autres programmeurs et la manière dont les programmes sont interprétés par des ordinateurs. Finalement, Les aspects pragmatiques des langages de programmation concernent les façons d'utilisations possibles de ces derniers en pratiques. Ces aspects ont une influence sur le comportement pratique des programmeurs lors de la conception et l'implémentation de leurs programmes.

Les aspects syntaxiques s'avèrent moins importants par rapport aux aspects sémantiques et pragmatiques puisque la syntaxe n'intervient qu'en dernier stage de l'implémentation de la solution de n'importe quel problème. Généralement, les développeurs lors de la résolution d'un problème commencent par décomposer le problème en sous problèmes, suite à cette décomposition ils identifient les différentes unités logicielles de la solution à implémenter comme les procédures, les types abstraits de données, les classes,…etc. Après leurs identifications, il est question de concevoir les implémentations souhaitées pour chacune des unités en utilisant les concepts supportés par les langages de programmation comme les structures de contrôle, les exceptions,…etc. Finalement, il faut coder chaque unité, et là, la syntaxe des langages de programmation est un élément relevant. Par conséquence, l'étude des langages de programmation doit prêter une attention particulière pour les aspects sémantiques et pragmatiques. Un constructeur donné peut être fourni par plusieurs langages de programmation avec des syntaxes différentes. La variation des syntaxes de ce même constructeur n'est que superficielle. Le plus important c'est de souligner les différences sémantiques entre des constructeurs qui appariaient similaires ; d'identifier les confusions entre des concepts distincts supportés par un langage de programmation ; de faire des jugements si tel ou tel langage de programmation supporte de manières adéquates tel ou tel concept.

4 Historique des langages de programmation

Le développement des langages de programmation a commencé depuis les années 1950s. Et depuis, plusieurs concepts ont été proposés et incorporés successivement dans les différents langages de programmation dont les conceptions dans la plus part des cas ont été influencées par les expériences d'usages des langages de programmation antérieurs. Les figures 1.1, 1.2, 1.3, 1.4 et 1.5 illustrent le développement historique de la majorité des langages de programmation.

Figure 1.1 : Historique des langages de programmation



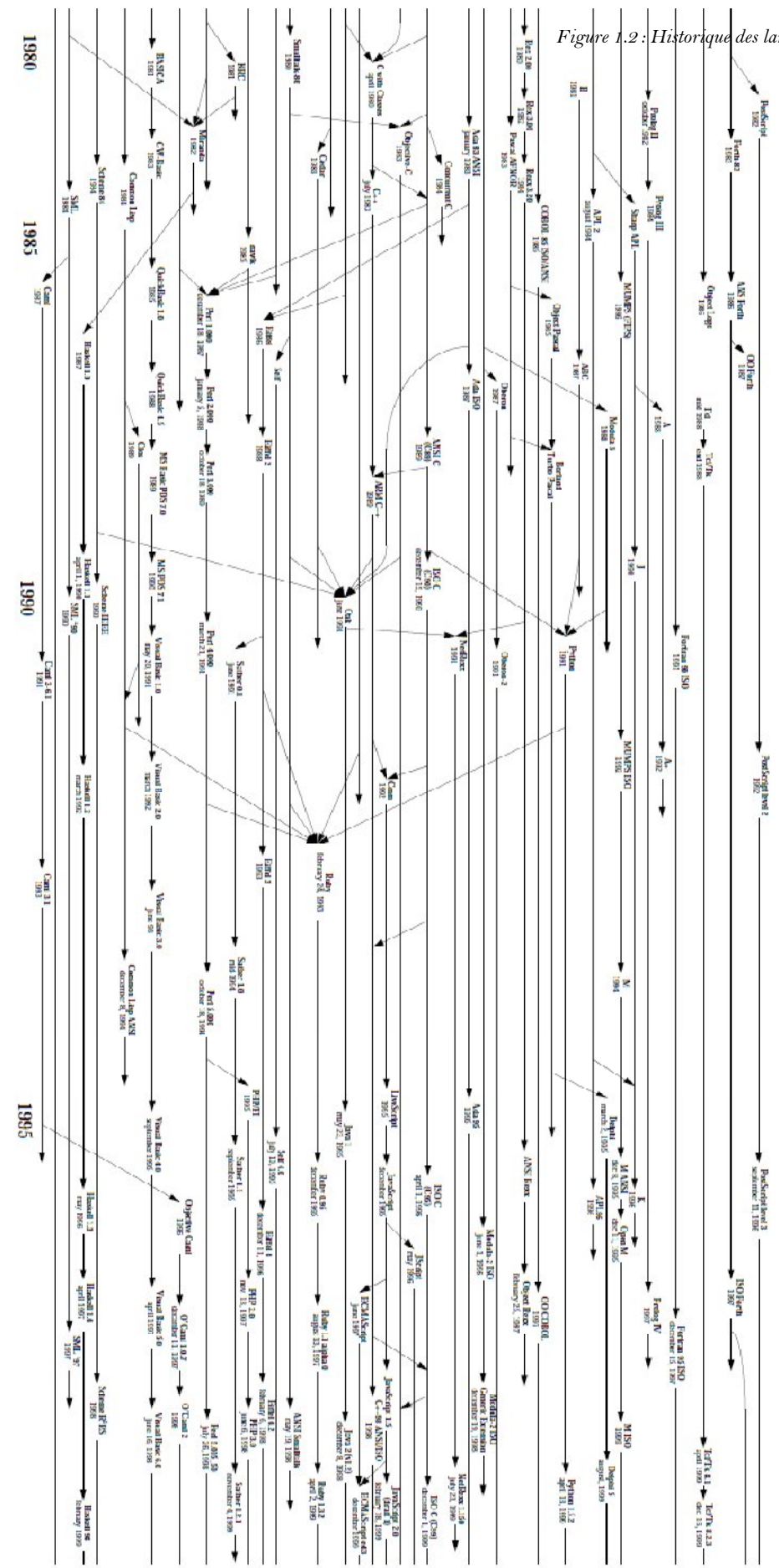


Figure 1.2 : Historique des langages de programmation

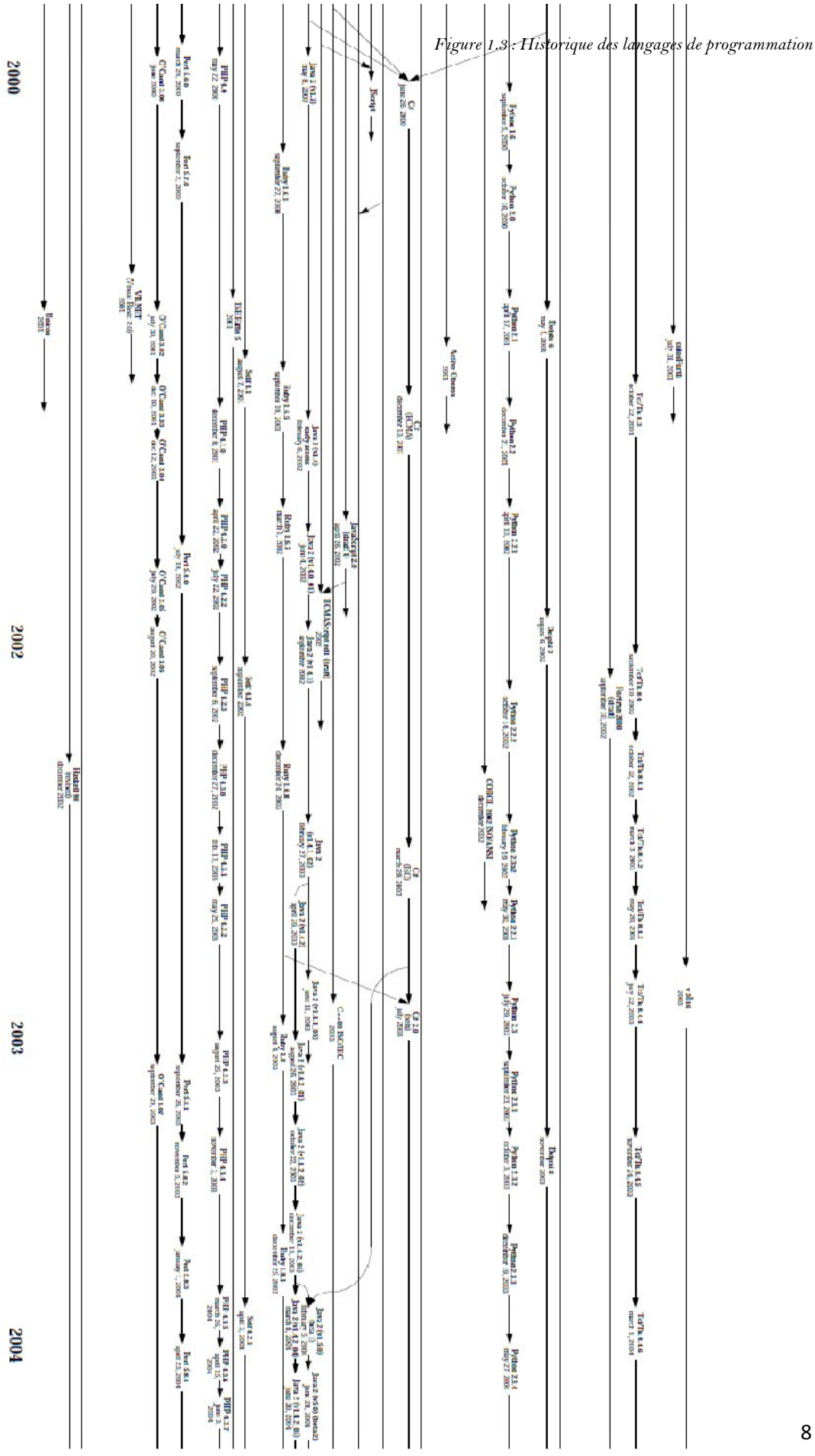


Figure 1.3 : Historique des langages de programmation

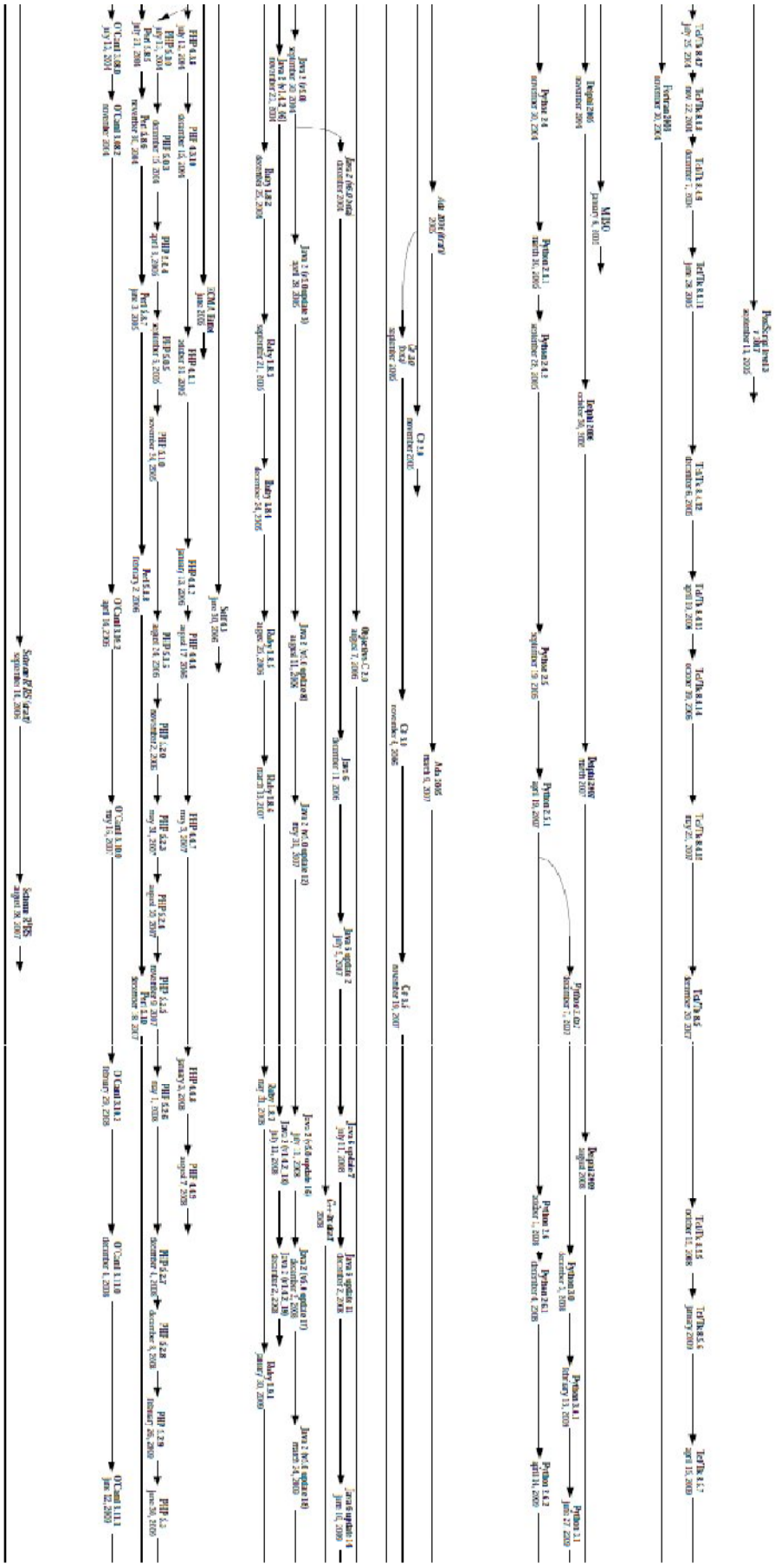


Figure 1.4 : Historique des langages de programmation

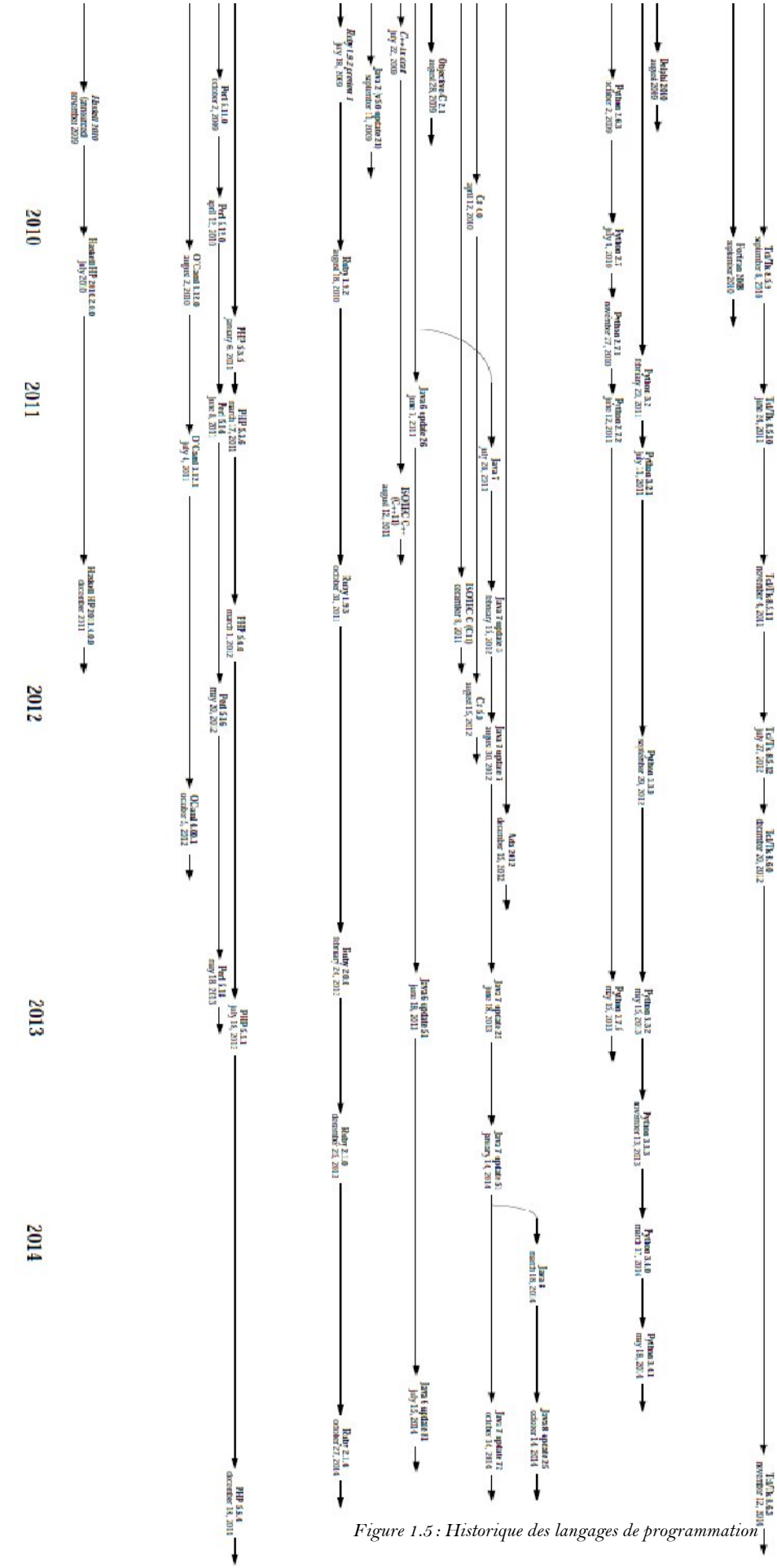


Figure 1.5 : Historique des langages de programmation

5 Concepts fondamentaux de la programmation

5.1 Noms et identificateurs.

Un *nom* est une *voie d'entrée* sur l'une des entités d'un programme. On réfère à une entité par un *nom* dans toutes les manipulations de cette dernière (création, utilisation, …, destruction). Généralement, toute entité manipulable lors de la programmation a besoin d'un nom (les constantes, les variables, les opérateurs, les types, les procédures, les fonctions, les modules, les programmes, les fichiers, les items de menus,…, etc.). Un *alias* est un autre nom pour une même entité (i.e., deux identificateurs désignant la même entité).

On désigne par un *identificateur* un nom sur lequel il peut exister des contraintes comme : un nombre maximum de lettres ; un ensemble de caractères permis ; la sensibilité aux majuscules ; ou la présence de mots réservés qui représentent la colle syntaxique d'un programme et que le programmeur ne peut pas les employer pour désigner les entités d'un programme.

5.2 Variables, types de donnée et expressions.

Une *variable* dans un programme est une abstraction d'une cellule de mémoire ou d'une collection de cellules. Cette notion est utilisée principalement dans les langages impératifs comme C et Ada. Le plus souvent, une restriction est faite par les langages de programmation sur les valeurs à mettre dans les variables dans le but de permettre aux compilateurs de générer un code précis pour la manipulation de ces valeurs ainsi que pour prévenir quelques erreurs courantes de programmation. Une variable peut être caractérisée par : son *nom*, son *adresse*, sa *valeur*, son *type*, sa *durée de vie* et sa *portée*. Le programmeur décide du nom et du type de la variable. L'emplacement de la déclaration décide de sa portée et de sa longévité. Son adresse est déterminée pendant l'exécution et sa valeur dépend des instructions dans lesquelles elle apparait.

Une *valeur* est une entité manipulable par un programme. Les valeurs peuvent être évaluées, stockées, passées comme arguments, retournées comme résultats de fonctions, et ainsi de suite. L'ensemble de valeurs sur lesquelles les mêmes opérations sont définies constituent un *type*. Les restrictions faites sur les variables par les langages de programmation sont généralement sous la forme d'informations de type. Le type d'une variable est une restriction sur les valeurs qu'elle peut tenir et quelles opérations peuvent être appliquées à ces valeurs. Quand les valeurs d'un type sont non décomposables, le type est primitif. Dans le cas où les valeurs d'un type sont décomposables, le type est dit composite. Différents langages de programmation prennent en charge différents types de valeurs. Par exemple, le langage C prend en charge les nombres entiers, les nombres réels, les structures, les tableaux, les pointeurs vers des variables et les pointeurs vers des fonctions. (Les entiers, les réels, et les pointeurs sont des valeurs primitives. Alors que les

structures et les tableaux sont des valeurs composites). Le langage C ++, supporte tous les types de valeurs supportés par le langage C plus les objets. Java prend en charge les booléens, les entiers, les réels, les tableaux et les objets. Le langage ADA propose des supports pour les types booléens, caractères, énumérés, entiers, réels, chaînes de caractères, tableaux, enregistrements, enregistrements variant , pointeurs vers les données, et des pointeurs vers des procédures.

```
1  variable

2            Premier : pointer to integer ;
3            Second : array  0..9  of
4                                Record
5                                Troisieme:  character;
6                                Quatrieme:  integer;
7                                Dernier : (Sam, dim, lun, mar, mer, jeu, ven)
8                                end;

9    begin
10   Premier := nil;
11   Premier := &Second[1].Quatrieme;
12   Premierˆ := 4;
13   Second[3].Quatrieme:=(Premierˆ+Second[1].Quatrieme)* Second[Premierˆ].Quatrieme;
14   Second[0] := [Troisieme : 'x'; Quatrieme : 0; Deriner : jeu];
15   end;
```

Listing 1.1 : exemple d'utilisation de variables et de types.

Examinons le code figurant dans le listing 1.1. Deux variables sont déclarées : *Premier* (ligne 2) et *Second* (lignes 3-8). Le code du listing 1.1 utilise le type *character* (caractères) et le type *integer* (entier) qui sont des types primitifs tout comme le type booléen et réel supportés par la majorité des langages de programmation. Par exemple, Le type *integer* englobe les valeurs numériques entières entre une valeur minimale et valeur maximale en dépendance des langages de programmation (ou dépendant de l'implémentation); Les valeurs de ce type peuvent agir comme opérandes dans des opérations arithmétiques. Les types énumérés sont aussi primitifs. Le code du listing 1.1 utilise un type énuméré dans la ligne 7; ses valeurs sont limitées aux valeurs spécifiées. Les types énumérés définissent souvent l'ordre de leurs éléments.

Les *types structurés* sont construits à partir d'autres types. Les tableaux (*arrays*), les enregistrements (*records*), et les pointeurs sont des types structurés. Dans le code du listing 1.1, trois sortes de types structurés standards sont utilisés. Les différents blocs qui constituent un type structuré sont dits les *composants du type*. La valeur d'un type structuré est composée des valeurs de ses composants. Le type pointeur *Premier* (linge 2) a un seul composant (entier). Le tableau *Second* déclaré dans les lignes (3-8) est constitué de dix valeurs de type *record* chacune. Habituellement, les valeurs qui composant un tableau sont de même type. Les tableaux sont indexés par des éléments d'un type d'index, généralement

sont une plage de nombres entiers, de caractères, ou un type énuméré. Ainsi, un tableau a deux types de composants (le type de base et le type d'index). Quelques langages de programmation proposent une structure de tableau dynamique. C'est-à-dire un tableau dont la taille n'est pas figée pendant toute l'exécution du programme dans lequel le tableau est créé, le nombre d'éléments dans le tableau change à fur et à mesure que l'exécution du programme progresse selon qu'on ajoute ou on retire des éléments. Certains langages de programmation comme ADA permettent la manipulation de *portions de tableaux* (*array slices*). Par exemple, `second[3..5]` est une portion du tableau `second` déclaré dans le listing 1.1.

La première constituante de l'*enregistrement* défini dans les lignes 4-8 du listing 1.1 est de type *caractère* ; la deuxième est de type *entier*. Quant à la troisième, elle est *énumérée*. Les enregistrements sont composés de plusieurs valeurs tout comme les tableaux à la différence que les valeurs qui composent un enregistrement ne sont pas forcement de même type et sont indexées par des *noms de champs* et non pas par des valeurs membres d'un type d'index.

Un *littéral* est une valeur, généralement d'un type primitif, explicitement notée dans un programme. Par exemple, 547 est un littéral entier. Dans le listing 1.1 plusieurs littéraux sont utilisés tels que : 0, 1, 3, 4, 9, et "x". Certaines valeurs sont fournies en tant que constantes pré-déclarées sous la forme d'identificateurs avec des valeurs prédéfinies et inchangeables comme *false* (Booléen) et *nil* (pointeur).

Une *expression* peut être : un littéral, une constante, une construction, une variable, une valeur de retour de l'invocation d'une fonction, une expression conditionnelle, ou un opérateur avec des opérandes qui sont eux-mêmes des expressions. Une *construction* est une expression qui construit une valeur composite à partir des valeurs de ses composantes. Dans certaines langues de programmation, les valeurs des composantes doivent être des littéraux; dans d'autres, les valeurs des composantes sont calculées par l'évaluation de sous-expressions. Un *opérateur* est un raccourci pour une invocation d'une procédure dont les paramètres sont les opérandes. Chaque opérateur fonctionne sur un nombre d'opérandes bien déterminé. On distingue des opérateurs unaires (à un seul opérande) et binaires (à deux opérandes). Les opérateurs unaires sont souvent écrits avant leur opérande par exemple -7, mais il existe des cas où les opérateurs unaires sont écrits après l'opérande par exemple : **ptrVar^**.

5.3 Systèmes de typage

Dans un langage de programmation, un système de type regroupe les valeurs dans des types. Il consiste d'un côté, en mécanismes permettant de définir les types et leurs associations avec les constructions du langage de programmation ; et un ensemble de règles de compatibilité, d'équivalence et d'inférence de type de l'autre côté. Cela permet une description efficace des données et de prévenir les programmes d'effectuer des opérations du genre multiplier un caractère par un entier. L'exécution d'une telle opération est appelée une

erreur de type. Les langages de programmation évolués diffèrent des langages de bas niveau par leurs possessions de systèmes de types. Dans un langage typique de bas niveau, les seuls " types " sont les octets et les mots, d'où des opérations qui n'ont pas de sens du genre multiplier un caractère par un entier ne peuvent pas être évitées.

5.3.1 Typage statique vs typage dynamique

Le système de types d'un langage de programmation permet un contrôle sur l'utilisation correcte des données que l'on appelle *vérification des types*. Avant la réalisation de toute opération, les types de ses opérandes doivent être vérifiés afin d'éviter une erreur de type. Par exemple, avant une multiplication des deux entiers, les deux opérandes doivent être vérifiés pour s'assurer qu'ils sont bien des entiers. Avant qu'une opération d'indexation d'un tableau soit effectuée, l'opérande de gauche doit être vérifié pour s'assurer qu'il est en effet un tableau. Ce contrôle peut être effectué au moment de la compilation ou au moment de l'exécution. Le moment de la vérification des types est à la base d'une classification importante des langages de programmation en : langues à typage statique dits aussi statiquement typés ou typés et langages à typage dynamique dits aussi typés dynamiquement ou non typé. Dans les langages de programmation typés ou à type statique chaque variable et chaque expression a un type fixe (qui est, soit explicitement indiqué par le programmeur ou inféré par le compilateur). En utilisant cette information, les types de tous les opérandes peuvent être vérifiés lors de la compilation. Dans les langages de programmation non typés ou à typage dynamique seules les valeurs ont des types fixes ; Les variables et les expressions n'ont pas de types fixes. Chaque fois qu'un opérande est calculé, il peut obtenir une valeur d'un type différent. Donc les types des opérandes doivent être vérifiés après leurs calculs, mais avant d'effectuer les opérations, au moment de l'exécution. Le contrôle des types est utile s'il s'exerce à la compilation (*typage statique)*, le code compilé n'a pas besoin de contenir de vérifications dynamique de types d'où il est potentiellement plus efficace. Cependant, le typage statique introduit aussi des contraintes. Certains programmes corrects peuvent être refusés par le vérificateur de types alors qu'ils ne provoquent pas d'erreurs à l'exécution. Certains programmes peuvent avoir des comportements dynamiques plus intéressants si certaines décisions sont prises à l'exécution et non pas à la compilation. En particulier pour les langages à objets, l'interprétation dynamique plutôt que statique des messages est un élément de souplesse important. La plupart des langages de programmation de haut niveau sont à typage statique. Smalltalk, Lisp, Prolog, PERL, et Python sont des exemples de langages de programmation typés dynamiquement.

Le listing 1.2 inclut deux codes, le premier est en C++ où le typage est statique. Le second est un code PYTHON qui est un langage à typage dynamique. Dans le premier code, bien que le compilateur ne connaît pas la valeur du paramètre *n*, il sait que cette valeur doit être de type *int*, car cela est indiqué dans la déclaration de *n*. A partir de cette connaissance,

le compilateur peut en déduire que les deux opérandes de "% " seront de type *int*, donc le résultat de l'opération "% " sera également de type *int*. Ainsi, le compilateur sait que les deux opérandes de " == " seront de type *int*. Enfin, le compilateur peut déduire que la valeur retournée sera de type *bool*, ce qui est cohérent avec la déclaration du type du résultat de la fonction. Considérons l'appel de fonction *Paire (i + 1)*, où la variable *i* est déclarée de type *int*. Le compilateur sait que les deux opérandes de " + " seront de type *int*, ainsi que son résultat sera aussi de type *int*. Ainsi, le type de l'argument de la fonction est connu pour être compatible avec le type du paramètre de la fonction. Donc, même sans connaissance de l'une des valeurs concernées (autres que les littéraux), le compilateur est en mesure de certifier qu'il y aura pas d'erreurs de type. Le second code exprime exactement la même chose que le premier à la différence que le langage utilisé est à typage dynamique. Nous remarquons que le type de la valeur de *n* n'est pas connu à l'avance, l'opération "% " a besoin d'un contrôle de type à l'exécution pour assurer que son opérande gauche est un entier. La fonction *Paire* peut être appelée avec des arguments de différents types, comme dans " *Paire (i + 1)* ", ou " *Paire(xyz)* ". Cependant, aucune erreur de type sera détectée jusqu'à ce que l'opérande gauche de "% " avère ne pas être un entier.

```
bool Paire  (int n) {
return (n % 2 == 0);
}
```

Code 1 en C++

```
def Paire  (n) :
return (n % 2 == 0)
```

Code 2 en PYTHON

Listing 1.2 : codes illustrant le typage statique et le typage dynamique.

Un langage de programmation est dit *fortement typé* si : (i) il a un typage statique (toutes les erreurs de types sont détectables à la compilation) et (ii) si aucune conversion implicite de type (*coercition*) n'est autorisée. Par exemple, en *C* qui est fortement typé l'instruction suivante n'est pas autorisée : (`float a= 2 ;`). La conversion explicite (*casting)* est autorisée entre des types compatibles. Par exemple, il est permet d'écrire en *C* pour une variable réelle *x* et une variable entière *n* : `x= float(n) * 3.14;`

La conversion des types explicite ou implicite peut parfois changer le format des données. Cependant, il est parfois nécessaire de traiter une expression d'un type donné comme si elle était d'un autre type, sans aucune conversion du format des données. Par exemple, un message pourrait ressembler à un tableau de caractères pour une procédure, alors qu'une autre procédure doit le traiter comme un record avec des champs de données. Le langage *Wisconsin Modula* introduit l'opérateur de conversion qui ne change pas de format des données *qua* à cet effet. En *C*, qui n'a pas un tel opérateur, le programmeur qui souhaite réaliser une conversion de type sans changer de format des données doit convertir un pointeur sur le premier type en un pointeur vers le deuxième type; les pointeurs ont la même représentation, peu importe ce qu'ils pointent dans la plupart des implémentations C. Le code dans le listing 1.3 illustre les deux méthodes. La ligne 10 montre comment *P* peut être converti sans conversion (sans changement de format) dans le deuxième type en Wisconsin

Modula. La Ligne 11 montre la même chose pour C, où on utilise le nom du type *SecondTypePtr* comme une routine de conversion explicite. L'opérateur '&' produit un pointeur à *P*. Dans les deux cas, si les deux types n'entendent pas sur la longueur de la représentation, du chaos peut en résulter.

```
1        type
2        PremierType = ... ;
3        SecondType = ... ;
4        SecondTypePtr = pointer to SecondType;
5        variable
6        P : PremierType;
7        S : SecondType;
8        begin
9        ...
10       S := P qua SecondType; -- Wisconsin Modula
11       S := (SecondTypePtr(&P))^; -- C
12       end;
```

Listing 1.3 : conversion de type sans changement de formats des données.

Les différences essentielles entre un typage statique et un typage dynamique peuvent être résumées dans les points suivants :

- Le typage dynamique exige des vérifications de type souvent répétées en exécution d'où un ralentissement de l'exécution du programme en résulte. Le typage statique ne nécessite que le temps de contrôle des types à la compilation qui est fait une seule fois, dont le coût en termes de temps est minimal. Le typage statique est plus efficace concernant ce point.
- Le typage dynamique impose un marquage ou un étiquetage de toutes les valeurs (dans le but de rendre possible les contrôles de type à l'exécution), ces étiquettes nécessitent de l'espace de stockage supplémentaire. Le typage statique ne nécessite pas de tels marquages ou étiquettes. Donc encore une fois, le typage statique est plus efficace.
- Le typage statique est plus sûr: le compilateur peut certifier que le programme ne contient pas d'erreurs de type. Dans un langage à typage dynamique on ne trouve pas une telle garantie. Ce point est important car les erreurs de type représentent une proportion importante des erreurs de la programmation.
- Le typage dynamique permet une grande flexibilité, ce qui est requis par certaines applications où les types de données ne sont pas connus à l'avance.

5.3.2 Compatibilités et équivalences de types

Concéderons un langage de programmation à typage statique. Soit un opérateur *op* qui agit sur un opérande de type T_1. Si l'opérateur *op* est utilisé pour agir sur une valeur d'un type T_2, il est question de vérifier que le type T2 est compatible avec le type T1 ou que les deux types sont équivalents.

L'équivalence entre deux types T_1 et T_2 peut être définie de deux manières différentes ; une *équivalence structurelle* et une *équivalence par nom*. Les deux types T_1 et T_2 sont

structurellement équivalents si et seulement si les deux types ont le même ensemble de valeurs. L'équivalence structurelle entre deux types T_1 et T_2 est appelée ainsi parce qu'elle peut être vérifiée en comparant les structures des deux type T1 et T2 puisque Il est inutile, et même en général impossible d'énumérer toutes les valeurs de ces types. Contrairement à l'équivalence structurelle, l'équivalence par nom ou *équivalence de déclaration* indique que pour que deux variables soient du même type, elles doivent être déclarées avec le même nom de type. Examinons le code dans le listing 1.4. Les variables *x*, *y* et *z* sont équivalentes structurellement. Selon une équivalence par nom, les variables *x* et *y* ont le même type mais la variable *z* est d'un type différent. L'équivalence structurelle stipule que deux types sont équivalents si, après le remplacement de tous les identificateurs du type par leurs définitions, la même structure est obtenue. Cette définition est récursive, parce que les définitions des identificateurs de type peuvent eux-mêmes contenir des identificateurs de type. Avoir une même structure c'est un peu vague. T_4, T6 et T_3 peuvent être considérés structurellement équivalents dans certains langages de programmation mais pas pour tous les langages de programmation. Tester l'équivalence structurelle entre deux types n'est pas toujours trivial, puisque les types peuvent être définis d'une manière récursive. Les types TC et TD définis dans le listing 1.4 sont structurellement équivalents, bien que leurs développements soient infinis.

Le fait que deux types de données partagent la même structure ne signifie pas qu'ils représentent la même abstraction. T_1 et T_3 du listing 1.4 peuvent représenter deux réalités différentes. T_1 représente les tailles de dix étudiants, tandis que T3 représente la moyenne pondérée cumulative de dix étudiants dans un cours donné. Cette interprétation, nous conduit à ne pas considérer T_1 et T_3 comme équivalents!

Un langage de programmation peut permettre l'affectation même si le type de l'expression source et le type de la variable de destination ne sont pas équivalents; ils doivent être compatibles pour l'affectation. Par exemple, sous le nom de l'équivalence, deux types de tableaux pourraient avoir la même structure mais sont inéquivalents car ils sont générés par des différentes instances du constructeur de type tableau (comme T1 et T3 du listing 1.4). Néanmoins, le langage peut autoriser l'affectation si les types sont assez proches. Dans le même sens, deux types peuvent être compatibles avec le respect à toute opération, telles que l'addition, même si ils ne sont pas de types équivalents. Il est souvent un sujet de divergence de dire si un langage utilise l'équivalence par nom mais à des règles laxistes de compatibilité ou de dire qu'il utilise l'équivalence structurelle. (Deux types sont compatibles si les valeurs de l'un peuvent être converties dans l'autre ou si toutes les valeurs de l'un des types appartiennent à l'ensemble des valeurs de l'autre type. Par exemple, si l'un des types et un sous type de l'autre).

```
Type

T1, T2 = array[1..10] of real;
T3 = array[1..10] of real;
T4 = array[2..11] of real;
T5 = array[2..10] of real;
T6 = array[blue .. red] of real;
TC = record
     Data : integer;
     Next : pointer to TC;
     end;

TD = record
     Data : integer;
     Next : pointer to TD;
     end;

Var
x,y: T1;
z:T3
```

Listing 1.4 : équivalence structurelle et équivalence par nom des types

5.4 Portées, liaisons et visibilité

Dans la plupart des langages de programmation, le même nom peut être réutilisé dans des contextes différents et peut représenter des entités différentes. L'emplacement de la déclaration d'une entité décide de sa portée et de sa longévité. À chaque apparition d'un nom, un type; une adresse et une valeur lui son associés. Chaque apparition a une durée de vie différente et une portée différente. Dans le code figurant dans le listing 1.5, le nom *max* a deux apparitions différentes. Dans la première apparition, *max* est le nom d'une procédure ; dans la deuxième, *max* est une variable locale de la fonction b.

```
procedure max;
Var b: char;
Begin
…
End;
Function b;
Var max: integer;
Begin
…
end;
```

Listing 1.5 : portée d'une déclaration.

A Chaque entité d'un programme sont associés des attributs à différent moments dans la vie du programme. Ces liaisons peuvent être établies au moment de la compilation (*compile time*), au moment du chargement (*load time*), au moment de l'exécution (*run time*), celui de la liaison à d'autre programme (*Link time*),…etc. On dit qu'une liaison est statique si

elle prend place avant l'exécution du programme et reste inchangée pendant son exécution. Elle est dynamique si elle prend place pendant l'exécution du programme ou si elle peut changer au cours de son exécution. Par exemple, à une variable sont associés le nom, l'adresse, le type, la valeur et la portée. La liaison entre le nom et la variable se fait à la déclaration (*compile time*); la liaison entre une variable et son adresse se fait au moment du chargement ou au moment d'exécution. La liaison entre une variable est un type se fait dans certains langages comme Pascal au moment de la compilation, dans d'autres langages comme le Smaltalk, la liaison se fait au moment de l'exécution. La liaison entre une variable et une valeur peut se faire au moment du chargement dans le cas d'initialisations ou pendant l'exécution. La portée d'une variable est précisée lors de sa déclaration (*compile time).*

La *portée* d'une déclaration est la partie d'un programme dans laquelle la déclaration prend effet. De même, la portée d'une liaison est la partie du programme dans laquelle s'applique la liaison. Dans les premiers langages de programmation, la portée de chaque déclaration était l'ensemble du programme. Dans les langues modernes, la portée de chaque déclaration est influencée par la structure syntaxique du programme. En particulier, l'arrangement de *blocs*. Dans les langages modernes, les déclarations et les instructions d'un programme sont groupées en *blocs* dans le but de grouper ensemble les étapes d'une instruction non-élémentaire et de permettre l'interprétation des noms adéquatement. La *portée* d'un nom *N* dans un programme représente tous les emplacements dans le programme où le nom *N* réfère à la même entité. Chaque langage de programmation dispose de ses propres formes de blocs:

- Les blocs d'un programme en C sont : *les commandes de bloc* ({...}), *Les corps des fonctions*, *les unités de compilation* (fichiers sources) et *le programme dans son ensemble.*
- Les blocs d'un programme en Java sont : *les commandes de bloc* ({...}), *Les corps des méthodes*, *les déclarations des classes*, *les packages*, et *le programme dans son ensemble.*
- Les blocs d'un programme ADA sont : *les commandes de bloc* (**declare** … **begin** … **end ;**), *Les corps des procédures*, *les packages*, *les tâches*, *les objets protégés*, et *le programme dans son ensemble.*

Les noms introduits dans un bloc sont appelés *liaisons locales*. Un nom mentionné mais pas défini dans un bloc doit avoir été défini dans l'un des blocs de l'entourage. Les blocs constituant un programme peuvent être *nommé*s (avoir un nom). Les procédures et les fonctions sont des exemples de blocs nommés. Un bloc anonyme (sans nom) est semblable à une procédure sans nom. Ce genre de blocs sont appelés immédiatement et une seule fois. Généralement, on utilise un bloc anonyme pour implémenter des calculs nécessaires qu'une seule fois. Les variables utilisées dans ces calculs ne sont nécessaires que dans ces derniers d'où il est préférable de ne pas les déclarer à l'extérieur du bloc.

Les blocs dans un programme peuvent être *emboités (un bloc peut contenir des blocs).* Selon ce point, trois situations sont à distinguer. La *Structure de bloc monolithique* (figure 1.6 (a))et qui est la structure de bloc la plus simple. Cette structure malgré sa simplicité pose beaucoup de problème en particulier pour l'écriture de grands programmes. Les

programmeurs doivent veiller à ce que toutes les déclarations ont des identificateurs distincts. Ceci est très gênant pour les grands programmes élaborés par une équipe de programmeurs. Dans un langage avec *une structure à blocs séparés* (figure 1.6 (b)), le programme est divisé en plusieurs blocs sans chevauchement. Par exemple, en FORTRAN les corps de procédures ne peuvent pas se chevaucher, chaque corps de procédure agit comme un bloc indépendant. La portée d'une variable déclarée à l'intérieur d'un corps d'une procédure particulière est le corps cette procédure. Le champ d'application de chaque variable globale (et la portée de chaque procédure) est l'ensemble du programme. Les inconvénients d'une structuration en blocs séparés peuvent être résumés en deux points : (i) chaque procédure et chaque variable globale doit avoir un identifiant distinct ; (ii) une variable qui ne peut être locale à une procédure particulière est contrainte d'être globale, et donc sa portée est l'ensemble du programme, même si elle est accessible seulement que par deux procédures. Dans un langage à structure de blocs emboités (figure 1.6 (c)), les blocs peuvent être imbriqués dans d'autres blocs. Plusieurs langages de programmation permettent cette structuration. En ADA par exemple, les blocs corps de procédures, blocs de commandes, packages,… peuvent être librement imbriquées l'un à l'intérieur de l'autre. L'avantage de la structure de blocs imbriqués est qu'un bloc peut être situé la où les identifiants qu'il référence doivent être déclarés.

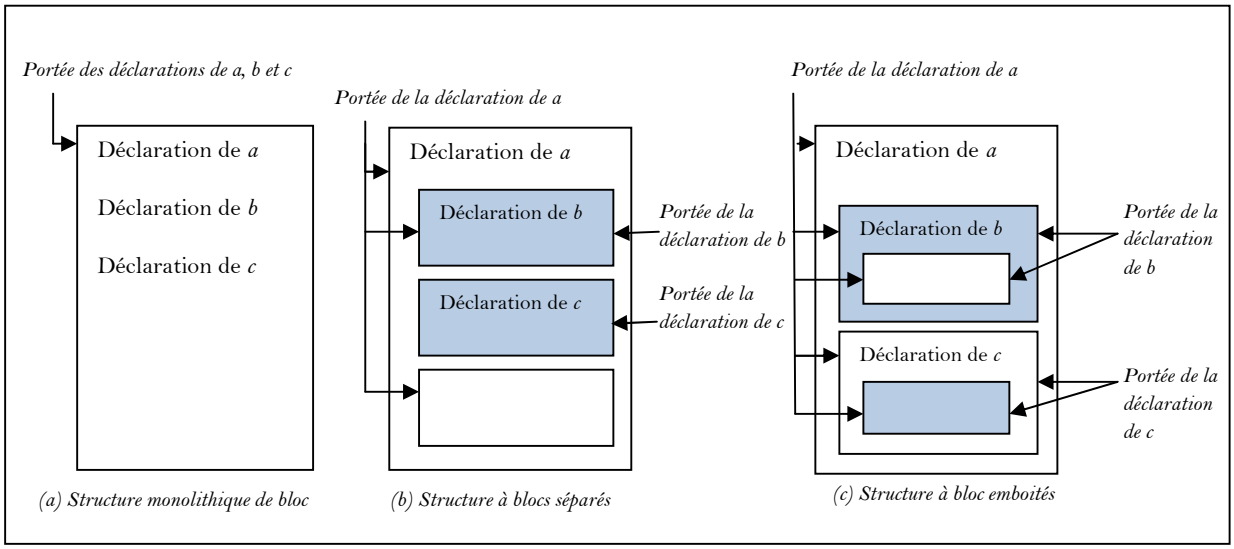


Figure 1.6 : structuration de programmes en blocs.

5.4.1 Visibilité

Dans un programme qui contient plus d'un bloc, il est possible qu'un même identificateur x soit déclaré dans des blocs différents. En général, x désigne une entité différente dans chaque bloc. Cela permet aux programmeurs de choisir librement les identificateurs à déclarer et à utiliser dans un bloc donné, sans se soucier de savoir si (par hasard) les mêmes identificateurs auraient été déclarés et utilisés dans d'autres blocs. Dans le cas où un identificateur x est déclaré dans deux blocs emboités A et B ($B \subset A$) comme il est

illustré dans la figure 1.7, la visibilité du *x* défini en A est perdue dans le bloc B qui ne voit que le *x* défini en B.

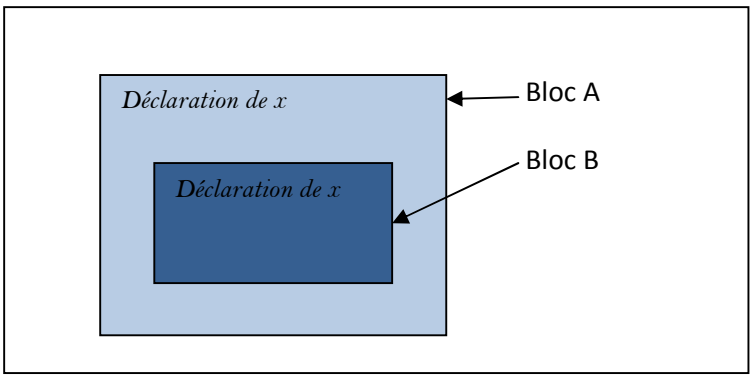


Figure 1.7 : visibilité d'une variable.

5.4.2 Portée statique et protée dynamique

Examinons le code C figurant dans le listing 1.6. Le résultat de l'appel de la fonction *func* dépend de la façon dont le langage interprète *s* au point (*) dans le corps de la fonction *func*. Le corps de la fonction *func* peut être exécuté dans l'environnement de définition de la fonction. Dans cet environnement *s* contient la valeur 5, et la fonction *func* retourne la valeur de son argument multiplié par 5, indépendamment de l'emplacement dans le programme où la fonction est appelée. La fonction *func* peut produire des résultats différents selon l'environnement dans lequel la fonction est appelée. Dans le point (**), la fonction *func* multiplie la valeur de *y* par 5, étant donné qu'à ce moment *s* désigne 5. L'appel de fonction *func* (*z*) au point (***) retourne la multiplication de la valeur de *z* par 4, car à ce point le, *s* dénote 4.

```
      const int s = 5;
      int func (int x) {
(*)   return s * x;
      }

      void p (int y) {
(**)  print(func(y));
      }

      void q (int z) {
      const int s = 4;
(***) print(func(z));
      }
```

Listing 1.6 : Portée statique et protée dynamique

Dans un langage de programmation *à portée statique* le corps d'une procédure ou d'une fonction est exécuté dans l'environnement de sa définition. Ainsi, les liaisons des identificateurs aux valeurs qu'ils représentent sont décidées à la compilation. Par contre,

dans un langage à *portée dynamique*, le corps d'une procédure est exécuté dans l'environnement de l'appel de la procédure. Cet environnement varie d'un appel de la procédure à un autre, d'où il n'est pas possible de décider avant l'exécution les liaisons entre les identificateurs et les valeurs qu'ils représentent. La portée dynamique cherche un nom dans la chaine des procédures appelées. Cette chaine considère les règles de visibilité mais pas l'emboitement. La portée dynamique n'a pas d'avantage à part le fait qu'elle soit plus simple à implémenter.

5.5 Structures (instructions) de contrôle.

Un programme informatique est constitué d'une suite d'instructions (commandes). Une *instruction informatique* désigne une étape dans un programme informatique. Une instruction dicte à l'ordinateur l'action nécessaire qu'il doit effectuer et dont le résultat est la production de valeurs ou/et une mise à jour de l'état d'une variable généralement. Les instructions sont une caractéristique des langages l'impératifs, orientés objet et les langages concurrents.

Les instructions peuvent être formées de plusieurs manières. Et nous distinguons : les instructions *primitives* ou atomiques (assignation, Instruction vide, Appel à une procédure, exit, next, break, continue goto (jump) et les *instructions structurées* souvent appelées structures de contrôle qui décrivent l'enchaînement des instructions. Les structures de contrôle permettent des traitements *séquentiels* de type {*S1 S2*}, *conditionnels* de la forme *if(C) S1 else S2* ou *répétitifs* (itératifs) de la forme *while (C) S*. Toute autre instruction de contrôle structurée est dérivée de ces trois mécanismes de base.

6. Evaluation des langages de programmation

Avec la prolifération et l'existence d'un nombre important de langages de programmation, il devient de plus en plus important de faire des évaluations aux différents langages de programmation pour les améliorer ; pour permettre leurs sélections d'une manière raisonnable et pour s'assurer de l'adéquaté d'un langage de programmation avec des besoins donnés. Plusieurs caractéristiques des langages de programmation sont étudiées pour être à la base de l'évaluation de ces derniers. Il est important de mentionner que ces caractéristiques ne sont pas forcement toutes nécessaires pour faire une évaluation d'un langage de programmation mais il y a certaines dont on déplore l'absence comme :

a) La simplicité

Cette caractéristique dénote qu'il devrait y avoir le plus peu possible de concepts (éléments) de base dans un langage de programmation. D'où, il est nécessaire d'éliminer d'un langage les éléments qui sont inutiles, difficile à lire ou à compiler. Beaucoup de programmeurs considèrent le langage PL / I, par exemple, d'être beaucoup trop grand comme langage de programmation. Certains critiquent Ada pour cette même raison. La simplicité d'un langage de programmation a un impact important tant sur la facilité de lecture des programmes que sur la facilité de leurs écritures pour les raisons suivantes :

- Si dans un langage de programmation, il existe beaucoup de composantes de bases, il est difficile de les connaître toutes.
- Aussi, s'il existe plusieurs façons d'exprimer une même commande, il est difficile de les connaître toutes.
- Si les opérateurs peuvent être surchargés, cela constitue une autre source de difficulté pour la lecture.
- Si un langage de programmation contient un large nombre de constructions différentes, il est fort possible que certain programmeurs ne les connaissent pas toutes. Dans ce cas, il se peut que ces programmeurs n'utilisent pas les meilleures constructions présentes.
- De même, il se peut que les programmeurs ne connaissent certaines constructions que superficiellement et les utilisent de manières erronées.

Néanmoins, la simplicité ne doit pas être poussée trop loin car, paradoxalement, elle cause la difficulté de l'écriture des programmes.

b) Uniformité

L'uniformité dans un langage de programmation signifie que ses concepts de base doivent être appliqués de manière cohérente et universellement. Les programmeurs doivent être en mesure d'utiliser les fonctionnalités du langage dans des contextes différents sans changer leurs formes. L'absente de cette caractéristique dans un langage peut être gênante. En Pascal par exemple, les constantes ne peuvent pas être déclarées avec des valeurs données par des expressions, même si les expressions sont acceptées dans tous les autres contextes où une valeur est nécessaire. L'absence de l'uniformité peut également être source d'erreurs. En Pascal, quelques boucles **For**, peuvent prendre qu'une seule instruction comme corps, mais les boucles **Repeat** peuvent prendre un certain nombre d'instructions dans le corps. Il est facile d'oublier de placer entre crochets les multiples instructions dans le corps d'une boucle **for**.

c) Orthogonalité

Le terme orthogonalité se réfère à la propriété d'être en mesure de combiner différentes constructions d'un langage dans toutes les combinaisons possibles, où chaque combinaison étant significative. Par exemple, un langage qui permet l'écriture d'expressions qui produisent des valeurs, et il fournit également une construction conditionnelle qui évalue une expression pour obtenir une valeur vraie ou fausse. Ces deux construction du langage, *expression* et *conditionnelle* sont orthogonales si une expression peut être utilisée et (évaluée) à l'intérieur de la construction conditionnelle.

L'orthogonalité est un concept important qui aborde comment un nombre relativement faible de composants peuvent être combinés dans un petit nombre de façons pour obtenir les résultats souhaités. Généralement l'orthogonalité est associée à la simplicité; plus le langage est orthogonal, il y a moins d'exceptions. Cela rend le langage plus facile à apprendre, et facilite aussi la lecture et l'écriture des programmes.

d) Possibilités d 'Abstraction

L'abstraction peut être définie comme étant la possibilité de définir des structures ou des opérations complexes tout en cachant leurs détails et permettre des accès très limités aux leurs constituantes. Selon qu'on défini des structures ou des opérations complexes, on distingue deux formes d'abstraction : *une abstraction des données* et *une abstraction des processus*. Un langage qui propose des supports pour l'abstraction, en plus que ce dernier doit permettre la réalisation de ces deux formes d'abstraction, il doit assurer des mécanismes permettant de sortir les patterns récurrents. L'abstraction est très importante dans l'écriture d'un programme car elle peut rendre l'écriture beaucoup plus aisée. Quand un processus est abstrait dans un sous-programme, son code n'est pas répété à chaque fois qu'il est requis, un simple appel du sous-programme est suffisant. Plusieurs langages de programmation comme ADA, C++ et JAVA permettent de faire abstraire les données sous forme d'objets à interfaces simples.

e) Présence de structures adéquates de contrôle

Il est très important, pour la lisibilité des programmes écrits dans un langage de programmation que ce dernier dispose de structures de contrôle adéquates. La première version du Basic n'avait comme structure de contrôle que l'instruction **goto**. Cela créait des situations où le lecteur était renvoyé à différents points du programme et rompait son flot régulier. Aujourd'hui, la majorité des langages de programmation proposent des structures de contrôle adéquates.

f) Types et structures des données

La présence de moyens appropriés pour définir des types et des structures de données dans un langage peut beaucoup aider la lisibilité. Par exemple, la présence du type *Boolean* dans un langage de programmation peut clarifier certaines instructions par rapport aux langages où on utilise des variables entières pour exprimer une situation à deux états. De même, la possibilité de créer des enregistrements à type varié (*record*) peut clarifier des expressions qui seraient, sans leurs présences, exprimées par plusieurs tableaux à indices communs.

g) Syntaxe aidant la lisibilité des programmes

Les choix de syntaxe peuvent influencer la lisibilité. Par exemple, une forme des identificateurs où ces derniers sont trop courts peut rendre la lecture d'un programme plus difficile. L'utilisation de mots spéciaux peut aider à la compréhension des programme (par exemple *begin* , *end*, … etc.). Néanmoins, si ces mots peuvent aussi être utilisés comme identificateurs, cela rend la lecture plus difficile.

h) Expressivité

L'expressivité d'un langage de programmation dénote à quel point le langage est équipé de moyens simples et commodes pour spécifier les calculs. Dans un sens formel, tous les langages utilisés en pratique peuvent exprimer exactement les mêmes algorithmes. Cependant, la facilité avec laquelle un programmeur peut réaliser un programme approprié est un élément de mesure essentiel de l'expressivité. Par exemple, comparer deux arbres binaires est assez difficile à faire dans la majorité des langages de programmation, mais c'est assez facile à réaliser avec le langage CLU. Pour la commodité, l'instruction *i*++ qui remplace *i=i+1* dans le langage C peut servir d'exemple.

i) Clarté

Les mécanismes offerts par un langage de programmation doivent être bien définis, et le résultat et les conséquences du code doivent être aisément prévisibles. Les programmeurs devraient être en mesure de lire des programmes de les comprendre aisément. Beaucoup de gens ont critiqué le langage C pour la confusion entre l'opérateur de l'affectation (=) et l'opérateur de test d'égalité (==).

j) Vérification des types

La vérification des types signifie qu'un langage de programmation est capable de détecter les erreurs relatives aux types des données au moment de la compilation ou au moment de l'exécution. Par exemple, l'étendue des tableaux n'est pas vérifiée en C, et lorsque les indices dépassent les limites, le programme ne s'arrête pas bien que les valeurs atteintes n'aient pas de sens. Quand un langage, comme le C, ne détecte pas ces erreurs, le programme peut-être exécuté, mais les résultats ne sont pas significatifs. Cette caractéristique affecte directement la fiabilité des programmes réalisés par de tels langages.

h) Prise en charge des exceptions

La prise en charge des exceptions est aussi une caractéristique qui a une influence sur la fiabilité des programmes. La possibilité pour un programme d'intercepter les erreurs faites pendant l'exécution, de les corriger, et de continuer l'exécution augmente beaucoup la fiabilité du langage de programmation. Des langages de programmation comme Ada, C++ et Java ont des capacités de prise en charge d'exceptions, d'autres comme le C ou le FORTRAN n'ont pas de telles capacités.

i) Autres caractéristiques

Plusieurs autres caractéristiques peuvent avoir des influences plus ou moins importantes sur la qualité d'un langage de programmation. En plus des caractéristiques présentées ci-dessus, les caractéristiques suivantes sont les plus couramment utilisées pour

servir de base aux évaluations des langages de programmation : la présence d'Alias, le Coût d'un langage de programmation, la portabilité, la généralité, la précision et la complétude de la description, la modularité, l'efficacité,…etc.

7. Conclusion

Ce chapitre présente une introduction pour l'étude des langages de programmation en les plaçant dans le contexte de l'étude des différents paradigmes de programmation. Dans ce chapitre, plusieurs concepts et éléments de base des langages de programmation sont présentés de manières concises. Ces concepts ainsi que d'autres seront détaillés dans les prochains chapitres où nous étudierons chacun des paradigmes de programmation mentionnés dans section 2 de ce chapitre de manières suffisamment détaillées.

Chapitre II:

La programmation impérative

1. Introduction

La programmation impérative est ainsi appelée parce qu'elle est basée sur les commandes (instructions) qui mettent à jour des variables occupant une ou plusieurs cellules dans la mémoire principale. Dans les années 1950s, les concepteurs des premiers langages de programmation ont reconnu que les variables et les commandes d'affectation constituent une simple abstraction, mais utile de la mémoire et des jeux d'instructions des ordinateurs. Cette relation étroite avec l'architecture informatique permet aux programmes impératifs d'être mis en œuvre très efficacement, du moins en principe.

La programmation impérative est demeurée le paradigme dominant jusqu'aux années 1990, quand elle a été contestée par la programmation orientée objet. Une grande partie des logiciels commerciaux actuellement en service ont été écrit dans des langages impératifs. La plupart des programmeurs professionnels d'aujourd'hui sont qualifiés largement ou exclusivement dans la programmation impérative.

Dans ce chapitre, nous allons étudier les concepts clés qui caractérisent les langages de programmation impératifs; les pragmatiques de la programmation impérative. A la fin du chapitre qu'un aperçu sur la conception du langage impératif ADA est donné.

2. Concepts clés

Les concepts clés de la programmation impérative sont classiquement: les variables, les commandes (instructions), les procédures et plus récemment: l'abstraction des données.

Les variables et les instructions sont des concepts clés de la programmation impérative. De nombreux programmes sont écrits pour modéliser des processus du monde réel manipulant des entités du monde réel ; une entité du monde réel possède souvent un état qui varie avec le temps. Ainsi, une entité du monde réel peut être modélisée naturellement par

des variables ; les processus du monde réel sont modélisés par des commandes qui inspectent et mettent à jour ces variables.

Les variables sont également utilisées dans la programmation impérative pour tenir les résultats intermédiaires des calculs. Toutefois, les variables utilisées de cette façon ne sont pas au centre de la programmation impérative, et en effet, elles peuvent souvent être éliminées par une reprogrammation visant une optimisation dans ce sens. Par exemple, on peut écrire à la fois une version itérative et une version récursive pour une procédure qui calcule *b ! (la factorielle d'un entier b)*. La version itérative a besoin de variables locales pour le contrôle de l'itération et pour accumuler le produit *2* x *3*x…x *b*. La version récursive n'a pas besoin de variables locales du tous.

Idéalement les variables ne servent que pour modéliser les états des entités du monde réel. Toutefois, pour éviter tout besoins de variables pour stocker les résultats intermédiaires, les langages impératifs doivent avoir un répertoire riche d'expressions, y compris les blocs d'expressions, les expressions conditionnelles et les expressions itératives, ainsi que les fonctions récursives. En pratique, les principaux langages impératifs (tel que C et ADA) sont pas assez riches à cet égard.

Les procédures sont un concept clé de la programmation impérative, car elles font une abstraction sur les commandes. Nous pouvons distinguer entre le comportement observable d'une procédure et l'algorithme (commandes) par lequel la procédure réalise ses comportements, une séparation utile des préoccupations entre les utilisateurs de la procédure et ses concepteurs.

L'abstraction de données n'est pas strictement indispensable à la programmation impérative. En effet, elle n'est pas prise en charge par les langages impératifs classiques tel que C et Pascal, mais elle est devenue un concept clé dans les langages impératifs les plus modernes tel que ADA. Nous pouvons distinguer entre les propriétés d'un type abstrait et sa représentation, entre le comportement observable des opérations sur le type abstrait et les algorithmes par lesquels ces opérations sont réalisées, encore une utilité de la séparation des préoccupations entre les utilisateurs du type abstrait et son réalisateur.

3. Pragmatiques de la programmation impérative

Les concepts clés de la programmation impérative influencent *l'architecture d'un programme* ainsi que le codage des programmes impératifs. Par *l'architecture d'un programme*, nous entendons la façon dont il est décomposé en *unités de programme*, avec les *relations* entre celles-ci.

La qualité de l'architecture d'un programme est importante pour les ingénieurs des logiciels car elle affecte directement le coût de la mise en œuvre et, plus tard de la maintenance du programme. Une mesure importante de la qualité est *le couplage*, qui

signifie la mesure à laquelle les unités de programmes sont sensibles aux changements dans d'autres. Un groupe d'unités de programme sont étroitement (fortement) couplées, si des modifications dans une unité sont susceptibles de forcer de majeures modifications aux autres unités, tandis qu'elles sont faiblement couplées si des modifications dans une unité sont susceptibles de vigueur modifications mineures dans la plupart des autres unités. Idéalement, toutes les unités d'un programme devraient être faiblement couplées. Un programme avec cette qualité est susceptibles d'être plus facile (et moins coûteux) à maintenir, car une unité de programme peut être modifiée si nécessaire sans forcer des modifications majeures dans les autres unités du programme.

Afin d'illustrer la programmation impérative, dans ce chapitre nous allons utiliser un programme correcteur orthographique simple comme un exemple de fonctionnement. Le correcteur orthographique utilise un dictionnaire de mots connus, qui sera chargé à partir d'un fichier dans la mémoire principale lorsque le programme est exécuté. Le correcteur orthographique est nécessaire pour copier les mots et la ponctuation à partir d'un document d'entrée à un document de sortie, interactivement le correcteur consulte l'utilisateur sur chaque mot inconnu. Si l'utilisateur choisit pour accepter le mot inconnu, le mot doit être ajouté au dictionnaire. Si l'utilisateur choisit d'ignorer le mot inconnu, ce mot et toutes les occurrences suivantes du même mot doivent être ignorées. Si l'utilisateur choisit de remplacer le mot inconnu, un nouveau mot entré par l'utilisateur doit remplacer le mot inconnu dans le document de sortie.

Un programme impératif traditionnel consiste en un ensemble de procédures où chaque procédure peut appeler d'autres procédures et un ensemble de variables globales accessibles par ces procédures. Dans la figure 2.1 est illustrée l'architecture du programme correcteur orthographique conçu selon une vision impérative. Le fait que les procédures appellent d'autres ne crée que des couplages faibles. C à d, une modification de la mise en œuvre d'une procédure est peu susceptible d'affecter ses appelants. Toutefois, l'accès aux variables globales crée un couplage fort où toute modification de la représentation d'une variable est susceptible d'imposer des modifications dans toutes les procédures qui y accèdent, d'ailleurs, chaque procédure est sensible à la façon dont la variable globale est initialisée, inspectée et mise à jour par d'autres procédures.

La figure 2.1 montre en effet une architecture possible pour le correcteur orthographique. La variable globale "*main-dict*" contient le dictionnaire qui a été chargé à partir d'un fichier (et qui sera ensuite enregistré dans le même fichier). La variable globale «*Ignored*» contient un dictionnaire temporaire de mots que l'utilisateur a choisi pour être ignorés. Les variables globales «*in-doc*» et «*out-doc*» font référence à l'entrée et la Sortie des documents. Les procédures de plus bas niveau ("*load-dict*", "*put-word*") opèrent sur les variables globales. Les procédures de niveau supérieur (« *consult-user* », «*Process-document*", "*main*") fonctionnent en grande partie en appelant les procédures du niveau inférieur.

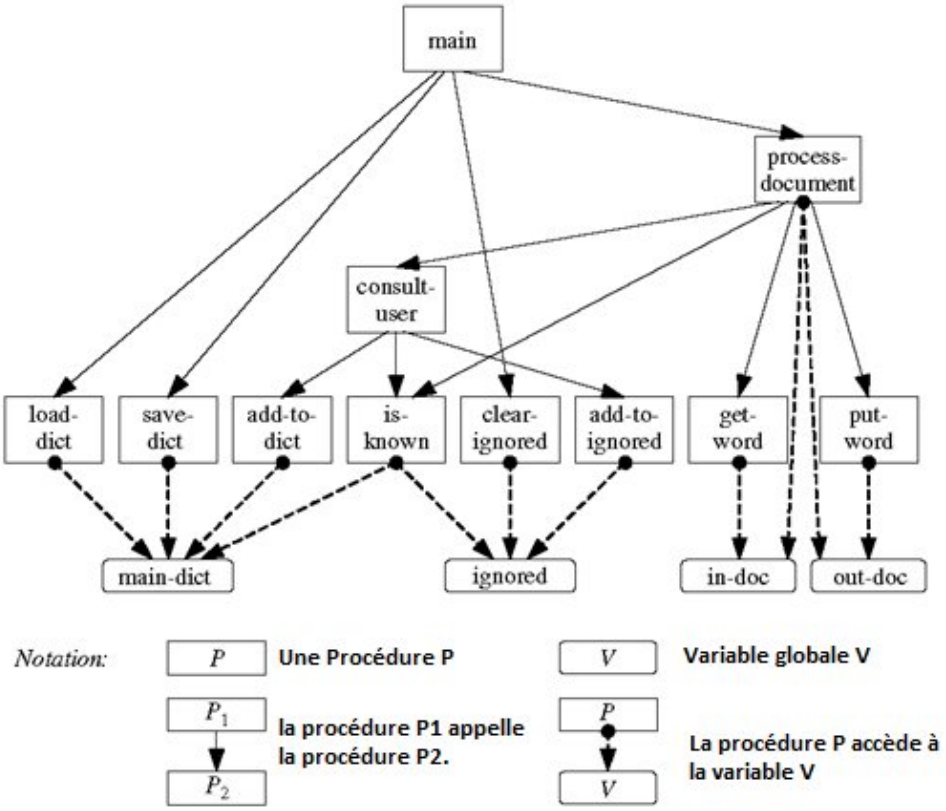


Figure 2.1 : *Architecture d'un programme impératif avec des variables globales*

Les programmes conçus selon des architectures similaires à celle présentée dans la figure 2.1 sont difficiles et coûteux à entretenir. Les programmeurs de maintenances seront incapables de comprendre toute procédure individuellement sans comprendre le rôle des variables globales auxquelles accèdent les procédures ainsi que les rôles des autres procédures qui accèdent aux mêmes variables globales. Toute modification apportée à une procédure pourrait déclencher une cascade de modifications dans d'autres procédures. De tels problèmes expliquent pourquoi l'abstraction des données est devenue un concept clé dans les langages impératifs les plus modernes (et, en effet, aussi pourquoi la programmation orientée objet est devenue importante). Les unités d'un programme impératif qui sont bien conçues maintenant sont les procédures et les types abstraits.

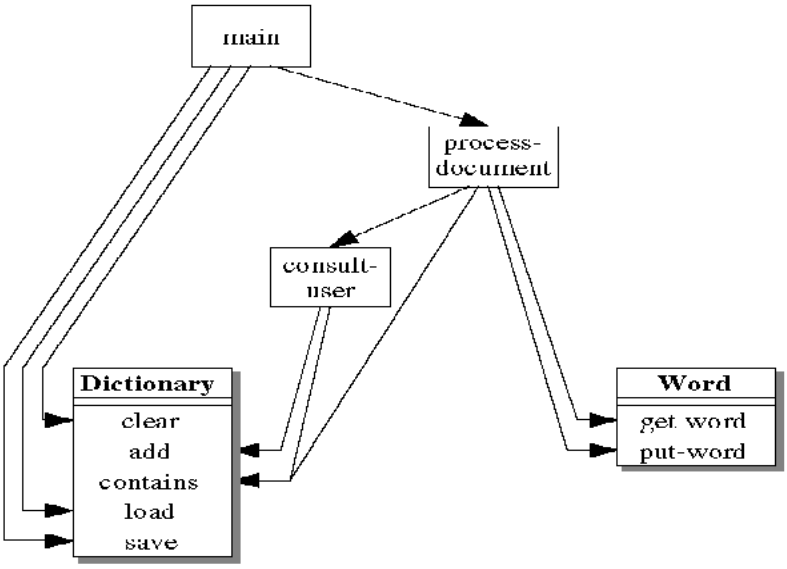


Figure 2.2**:** *Architecture d'un programme impératif avec abstraction des données*

La figure 2.2 montre une architecture alternative pour le correcteur orthographique, basé sur les types abstraits. Le type abstrait *Word* est équipé d'opérations de lecture et d'écrire des mots dans les documents. Le type abstrait *Dictionnaire* est équipé avec les opérations de chargement, de sauvegarde, d'effacement, d'ajout, et de la recherche. Les deux dictionnaires "*main-dict*» et «*ignored*» sont maintenant des variables locales de «*main*», et sont transmises comme paramètres à «*process-document*» et «*consult-user*».

La figure 2.2 illustre l'architecture d'un programme où il n'y a aucune variable globale, seules les variables locales et des paramètres (non représentés). Deux types abstraits de données ont été conçus, équipés chacun d'opérations suffisantes pour cette application. Maintenant toutes les unités du programme sont faiblement couplées les unes des autres. Le programmeur chargé de la maintenance sera en mesure de comprendre chaque unité individuellement, et peut modifier une unité sans craindre de provoquer une cascade de modifications à d'autres unités.

4. La programmation impérative à travers un langage comme ADA

Le langage de programmation ADA a été conçu dans les années 1970 comme un langage de programmation impératif et concurrent, adapté en particulier pour la mise en œuvre des systèmes à grande échelle et les systèmes embarqués. L'équipe de conception d'ADA était grande, et elle a reçu des contributions d'un nombre encore plus important de personnes

intéressées ; mais l'effort de conception a été dominé par Jean Ichbiah. Dans les langages de programmation conçus par un nombre important de personnes, le risque d'une prolifération de fonctions mal intégrées est important en tentant de satisfaire les exigences contradictoires de tous les membres du comité. Le Contrôle ferme de Jean Ichbiah a assuré qu'ADA a largement évité ce sort.

Comme conséquence de la façon dont il a été développé, ADA est un langage très large, mais raisonnablement cohérent. Il supporte presque tous les concepts liés à la programmation. En particulier, contrairement aux langages impératifs classiques tel que C et Pascal, Ada supporte l'abstraction des données, l'abstraction générique, et les exceptions.

ADA a été étendu dans les années 1990, principalement pour soutenir la programmation orientée objet. On utilise généralement les appellations ADA 83 et ADA 95 lorsqu'il est nécessaire de distinguer entre les deux versions du langage. Cette partie du cours propose un aperçu sur les parties impératives d'ADA.

4.1 Valeurs et Types

ADA a un répertoire complet des types primitifs: *Booléen*, *Caractère*, les types énumérés et les types numériques (*entiers*, *flottants*, et du *virgule fixe*). Fait inhabituel, ADA permet aux programmeurs de déclarer leurs propres types numériques, ce qui est intéressant pour la portabilité.

Le langage ADA propose un répertoire adéquat de types composites: les types tableaux, types d'enregistrement, types d'enregistrement à discrimination (disjoint unions) et les enregistrements variant. Les types récursifs ne sont pas supportés directement en ADA. Au lieu de cela, les programmeurs doivent déclarer des types récursifs en utilisant des pointeurs.

ADA est conforme au principe de complétude de type. Les Constantes, les variables, les paramètres et les résultats de fonction peuvent être de tout type. Les opérations de test d'égalité "=" et "/=", et l'opération d'affectation ":=", peuvent être appliquées à des opérandes de tout type (sauf si le programmeur décide autrement en déclarant un type à être limité).

ADA supporte les sous-types systématiquement dans le sens que tous les types peuvent avoir des sous-types. Cependant, un sous-type ADA n'est pas en soi un type. Une déclaration de type dans ADA crée un type nouveau et distinct, mais une déclaration de sous-type nomme simplement un sous-type d'un type existant. Un type est compatible avec n'importe lequel de ses sous-types. Par exemple, un type *natural* peut être déclaré comme un sous type d'*integer*. Le listing 2.1, contient une déclaration du sous type *natural*, ainsi que son utilisation comme le type du second paramètre de la fonction *power*.

```
Subtype Natural is Integer  range 0 .. Integer ' last ;

.....
function power (b: Float; n: Natural) return Float;
```

Listing 2.1 : Déclaration et utilisation d'un sous type.

Dans l'appel de la fonction *power*, le compilateur vérifie simplement que le type du second argument est compatible avec *natural* (c'est à dire, son type est entier ou un sous-type de Integer). Toutefois, un temps d'exécution supplémentaire peut être nécessaire pour s'assurer que la valeur de l'argument est dans le sous-type naturel. Dans le cas d'appel d'une procédure où le type d'un paramètre est *integer*, par exemple la procédure *inc* déclarée comme : `procedure inc (i: in out Integer);` le compilateur vérifie simplement que le type de l'argument est compatible avec integer (c à d, son type est *integer* ou un sous-type de *Integer*). Aucun temps d'exécution supplémentaire pour la vérification des types n'est nécessaire.

4.2 Variables et contrôle

ADA supporte les variables globales, locales, et les variables tas. Une variable tas de type T est allouée par une expression de la forme «new T», ou encore «new T '(...)" qui initialise aussi la variable tas.

Le répertoire de commandes (instructions) de l'ADA est classique pour un langage moderne, il regroupe les commandes skip, les commandes d'affectation, les appels de procédure, commandes séquentielles, commandes conditionnelles (IF et le case), commandes itératives (while, for, loop), commandes de bloc, et les commandes d'exception.

4.3 Liaisons et portées

Un programme ADA se compose d'unités de programme: les procédures, les packages, et les unités génériques. Le programmeur spécifie laquelle de ces procédures est traitée comme le programme principal.

Au sein d'une unité de programme ou d'un bloc de commande, on peut déclarer: des types et des sous-types, des constantes, des variables, des exceptions, des tâches, et même d'autres unités de programme.

Les Procédures, les packages, et les unités génériques sont les unités de programmes majeurs et sont normalement déclarées globalement ou à l'intérieur d'autres unités du programme. Bien qu'il soit autorisé en ADA de déclarer une unité de programme à l'intérieur d'un bloc intérieur pour permettre à l'unité de programme d'accéder aux variables déclarées dans des blocs extérieurs, cela donne lieu à un couplage fort.

4.4 Abstraction

L'abstraction peut être définie comme étant la possibilité de définir des structures ou des opérations complexes tout en cachant leurs détails et permettre des accès très limités aux leurs constituantes. ADA propose des constructions pour supporter trois sortes d'abstractions : une abstraction procédurale, une abstraction des données et une abstraction générique.

4.4.1 Abstraction procédurale

Le langage ADA supporte les concepts de procédures et de fonctions. Trois modes de paramètres sont possibles. Ces modes mettent l'accent sur le sens des flux de données entre une procédure et son environnement.

- Paramètres d'entrée (in parameters): le paramètre formel est une constante.
- Paramètres d'entrée - sortie:(in- out parameters) le paramètre formel est une variable, et permet à la fois l'inspection et la mise à jour de l'argument.
- Paramètre de sortie: :(out parameters) le paramètre formel est une variable, et ne permet que la mise à jour de l'argument.

4.4.2 Abstraction de données

ADA supporte une abstraction des données principalement par des moyens de paquetages (packages). Chaque package est constitué d'une spécification et d'un corps. La spécification du package sert à déclarer les composantes publiques du package, et donc de spécifier les *API* du package. Le corps du package sert à fournir les détails d'implémentation: définitions des procédures publiques, et les déclarations de tous les composants privés.

En général, les composants d'un package ADA peuvent être tout ce qui peut être déclaré dans le langage: les types et les sous-types, les constantes et les variables, les exceptions, les procédures, les unités génériques, des packages intérieurs, et ainsi de suite. Les packages ADA prennent en charge l'encapsulation.

Un cas particulier important est les cas d'un package qui définit un type abstrait. Dans ce cas, le package de spécification déclare le type abstrait lui-même, et précise les procédures publiques qui opèrent sur le type abstrait. Le nom du type abstrait est public, mais sa représentation est privée. Le corps du package définit les procédures publiques, et déclare les procédures privées. A titre d'exemples, considérons le programme correcteur orthographique introduit dans les sections précédentes, les listings 2.2, 2.3, 2.4 et 2.5 présentent respectivement : la spécification d'un package (paquetage) nommé *Words* ; l'implémentation ou le corps (body) du package *Words* ; la spécification d'un package nommé *Dictionaries* et l'implémentation ou le corps (body) du package *Dictionaries*. Un code pour une utilisation possible du package *Dictionaries* est illustré dans le listing 2.6. Le listing 2.7 présente un code ADA de quelques procédures du programme correcteur orthographique avec l'exploitation des deux packages.

```
with Ada.Text_IO; use Ada.Text_IO;
package Words is
type Word is private;
-- Each Word value is a single word.
procedure get_word (in_doc, out_doc: in out File_Type;
wd: out Word);
-- Read the next word from in_doc into wd, copying any preceding
punctuation
-- to out_doc. Raise end_error if there is no next word to be
read.
procedure put_word (out_doc: in out File_Type;
wd: in Word);
-- Write wd to out_doc.
private
type Word is . . .; -- representation
end Words;
```

Listing 2.2 : La spécification du package (paquetage) *Words*.

```
package body Words is
. . . -- auxiliary procedures
procedure get_word (in_doc, out_doc: in out File_Type;
wd: out Word) is
begin
. . .
end;
procedure put_word (out_doc: in out File_Type;
wd: in Word) is
begin
. . .
end;
end Words;
```

Listing 2.3 : Implémentation ou le corps (body) du package *Words*.

```
with Words; use Words;
package Dictionaries is
type Dictionary is limited private;
-- Each Dictionary value is a set of words.
procedure clear (dict: out Dictionary);
-- Make dict empty.
procedure add (dict: in out Dictionary; wd: in Word);
-- Make wd a member of dict.
function contains (dict: Dictionary; wd: Word)
return Boolean;
-- Return true if and only if wd is a member of dict.
procedure load (dict: out Dictionary;
filename: in String);
-- Load dict from filename.
procedure save (dict: in Dictionary;
filename: in String);
-- Save dict to filename.
private
type Dictionary is . . .; -- representation
end Dictionaries;
```

Listing 2.4 : La spécification du package (paquetage) *Dictionaries*.

```
package body Dictionaries is
. . . -- auxiliary procedures
procedure clear (dict: out Dictionary) is
begin
. . .
end;
procedure add (dict: in out Dictionary; wd: in Word) is
begin
. . .
end;
function contains (dict: Dictionary; wd: Word)
return Boolean is
begin
. . .
end;
procedure load (dict: out Dictionary;
filename: in String) is
begin
. . .
end;
procedure save (dict: in Dictionary;
filename: in String) is
begin
. . .
end;
end Dictionaries;
```

Listing 2.5: Le corps (body) du package *Dictionaries*.

```
use Dictionaries;
dict: Dictionary;
current_word: Word;
….
Load(dict);
loop
 if not contains(dict, current_word) then
….
End if ;
End loop;
```

Listing 2.6 : Un code pour une utilisation possible du package *Dictionaries*

```
with Words; use Words;
with Dictionaries; use Dictionaries;
procedure main is
procedure consult_user (current_word: in out Word;
main_dict, ignored: in out Dictionary) is
-- Ask the user what to do with current_word, which is unknown.
-- If the user chooses to accept the word, make it a member of main_dict.
-- If the user chooses to ignore the word, make it a member of ignored.
-- If the user chooses to replace the word, get the user to enter a
replacement word,
-- and update current_word.
begin
. . .
end;

procedure process_document (
main_dict, ignored: in out Dictionary) is
-- Copy all words and punctuation from the input document to the output
-- document, but ask the user what to do with any words that are unknown
(i.e.,
-- not in main_dict or ignored).
in_doc, out_doc: File_Type;
current_word: Word;
begin
open(in_doc, in_file, "indoc.txt");
open(out_doc, out_file, "outdoc.txt");
loop
get_word(in_doc, out_doc, current_word);
if not contains(main_dict, current_word) and then
not contains(ignored, current_word) then
consult_user(current_word, main_dict, ignored);
end if;
put_word(out_doc, current_word);
end loop;
exception
when end_error =>
close(in_doc); close(out_doc);
end;
main_dict, ignored: Dictionary;
begin
load(main_dict, "dict.txt");
clear(ignored);
process_document(main_dict, ignored);
save(main_dict, "dict.txt");
end;
```

Listing 2.7 : Un code ADA de quelques procédures du programme correcteur orthographique.

Les valeurs de type dictionnaire ne peuvent être manipulées qu'à travers l'appel des opérations publiques du package *Dictionnaires*. Le compilateur ADA empêche toute tentative par le code de l'application pour accéder à la représentation d'un dictionnaire. Ainsi, le code d'application peut être maintenu indépendamment du package.

Tous les types ADA, y compris les types abstraits, sont équipés par défaut avec l'opération d'affectation et les opérations de test d'égalité, à l'exception des types définis comme étant limités. L'affectation d'une structure de données statiques implique la copie de tous ses composants. D'autre part, l'affectation d'une structure de données dynamique (structures construites en utilisant des pointeurs) implique la copie des pointeurs, mais pas leurs référents. Cette incohérence crée un problème lorsque nous concevons un type abstrait puisque le type de la représentation est caché et n'est pas censé influer sur le comportement du code de l'application. Dans le cas où il est certain que le type abstrait est représenté par une structure de données statiques il faut le déclarer comme étant privé (**`Private`**). Si le type abstrait pourrait éventuellement être représenté par une structure de données dynamique, il faut le déclarer comme privé limité (**`limited private`**). Si dans ce cas, le type abstrait a besoins d'une affectation et / ou des opérations de test d'égalité, il est question de les définir dans le package du type abstrait.

4.4.3 Abstraction générique

Les unités de programme comme les packages et les types abstraits peuvent dépendre d'entités comme des valeurs et des types qui sont définies dans le programme en dehors des unités elles-mêmes d'où leurs réutilisations n'est pas poussées très loin. La clé de leurs réutilisations est de paramétrer ces unités de programme.

Une unité générique est une unité de programme qui est paramétrée par rapport à des entités dont elle dépend. L'instanciation d'une unité générique génère une unité de programme ordinaire, dans laquelle chacun des paramètres formels de l'unité générique est remplacé par un argument. Ce que revient à définir des algorithmes identiques opérant sur des données de types différents. Ainsi, on peut programmer par exemple une pile avec une procédure qui prend l'élément supérieur de la pile, indépendamment du type de données contenues. Une unité générique peut être instanciée plusieurs fois autant que nécessaire, ce que génère à la demande une famille d'unités de programmes similaires. Ce concept met un outil très puissant dans les mains des programmeurs. Une unité générique unique peut être instanciée plusieurs fois dans le même programme, évitant ainsi la duplication de code du programme. Elle peut également être instanciée dans de nombreux programmes différents, facilitant ainsi sa réutilisation.

ADA permet à n'importe quel package d'être générique. Donc, chaque package peut être paramétré à l'égard de valeurs, des variables, des types et des procédures dont il dépend. Les listings 2.8, 2.9 et 2.10 contiennent un exemple d'un package générique *tables* défini indépendamment de ce qu'il peut contenir.

```
generic
type Row is private;
type Key is private;
with function row_key (r: Row) return Key;
package Tables is
type Table is limited private;
-- A Table value contains a number of rows, subject to the constraint
-- that no two rows have the same key.
exception table_error;
procedure clear (t: out Table);
-- Make table t contain no rows.
procedure add (t: in out Table; r: in Row);
-- Add row r to table t. Throw table_error if t already contains
-- a row with the same key as r.
function retrieve (t: Table; k: Key) return Row;
-- Return the row in table t whose key is k. Throw table_error if
-- t contains no such row.
private
capacity: constant Positive := . . .;
type Table is
record
size: Integer range 0 .. capacity;
rows: array (1 .. capacity) of Row;
end record;
end Tables;
```

Listing 2.8 : spécification d'un package générique

Le package générique est paramétré avec les types *Row* et *Key* et la fonction *row_key*. Le package utilise *row_key* pour implémenter les opérations *add* et *retrive*.

```
package body Tables is
procedure clear (t: out Table) is
begin
t.size := 0;
end;
procedure add (t: in out Table; r: in Row) is
k: Key := row_key(r);
begin
for i in 1 .. t.size loop
if row_key(t.rows(i)) = k then
raise table_error;
end if;
end loop;
t.size := t.size + 1;
t.rows(t.size) := r;
end;
function retrieve (t: Table; k: Key) return Row is
begin
for i in 1 .. t.size loop
if row_key(t.rows(i)) = k then
return t.rows(i);
end if;
end loop;
raise table_error;
end;
end Tables;
```

Listing 2.9 : Le corps d'un package générique

Tant que *Row* et *Key* sont déclarés privés alors ils sont équipés des opérations d'affectation et de test d'égalité. Le listing 2.10 contient un code pour utiliser le package générique *Tables*.

```
subtype Short_Name is String(1 .. 6);
subtype Phone_Number is String(1 .. 12);
type Phone_Book_Entry is
record
name: Short_Name; number: Phone_Number;
end record;
function entry_name (e: Phone_Book_Entry)
return Short_Name is
begin
return e.name;
end;
package Phone_Books is new Tables(
Phone_Book_Entry, Short_Name, entry_name);
use Phone_Books;
phone_book: Table;
. . .
add(phone_book, "Amine", "+61733652378");
```

Listing 2.10 : utilisation d'un package générique

L'instanciation génère un package nommé *Phone_books* par la substitution de *Row* par *Phone_book_entry* et *key* par *entry_name*.

5. Conclusion

Dans ce chapitre, Nous avons identifié les principaux concepts de la programmation impérative: variables, commandes, abstraction procédurale, et l'abstraction de données. Nous avons étudié la pragmatique de la programmation impérative, en comparant les inconvénients de la programmation avec des variables globales avec les avantages de l'abstraction des données. Nous avons étudié aussi ce paradigme de programmation à travers l'un des langages impératifs les plus modernes. Il s'agit de l'ADA qui supporte en plus de tous les concepts de base de la programmation impérative, trois différentes sortes d'abstraction. La programmation impérative est demeurée le paradigme dominant jusqu'aux années 1990, quand elle a été contestée par la programmation orientée objet. C'est toujours les besoins progressifs en termes d'abstractions et de découplages entre les différentes constituantes d'un programme qui ont conduit à la naissance de la programmation orientée objets qui fera l'objet du prochain chapitre.

Chapitre III :

La programmation orientée objet

1. Introduction

Ce chapitre est consacré à l'étude des concepts clés qui caractérisent les langages de programmation orientés objet (POO) ainsi que la pragmatique de la POO. Ce cours est destiné aux étudiants de la licence en troisième année. A ce stade de la formation, les étudiants sont bien familiarisés avec les concepts de base de la programmation orientée objet. Par conséquence, la façon dont on approche ce paradigme de programmation se base sur la situation de la programmation orientée objet dans un cadre d'analyse représentant l'évolution de la programmation tout en insistant sur les pratiques, l'empreinte et les innovations importantes de la POO.

L'histoire de la programmation débute avec des concepts très proches de la machine (instruction, nombre entier,. . .), puis identifie un certain nombre d'abstractions de plus en plus de haut niveau (procédure, fonction, structure de donnée, sémaphore, processus, objet, message, composant, modèle. . .). Pour mieux approcher la programmation orientée objet, il est intéressant de la situer dans un cadre d'une perspective générale d'évolution de la programmation. Ce cadre est un repère dont la première dimension représente le niveau d'abstraction alors que la seconde représente le couplage entre différentes entités logicielles dans un programme. La projection sur la dimension du niveau d'abstraction reflète le besoin progressif d'abstraction qui n'a pas cessé d'augmenter. Ce besoin est témoigné par l'évolution de la programmation et du développement du logiciel allant des procédures et des structures de données, passant par les objets, les acteurs, les composants, les services, les modèles, les ontologies, les agents, les connaissances jusqu'à l'arrivée aux organisations et aux sociétés artificielles. La deuxième dimension représente les aspects liés aux liaisons entre différentes entités logicielles mises en jeu. Cette dimension reflète le besoin croissant en matière de description des liaisons et des relations indépendamment des entités qu'elles relient. Cette dimension exprime également la flexibilité des couplages signifiant la capacité

de mettre des relations entre différentes entités logicielles. Le troisième aspect exprimé à travers cette dimension est le besoin de repousser le plus tard possible la décision de choisir l'action à exécuter et qu'elle est l'entité responsable de son exécution.

Ce cadre d'analyse peut servir de repère pour étudier n'importe quel paradigme de programmation. Nous utilisons ce même cadre pour étudier dans les prochains chapitres les paradigmes aspects, composants, services et agents qui présentent des avancées technologiques par rapport aux objets tant sur le plan de l'abstraction que sur la flexibilité du couplage.

2. Rappels sur les concepts clés de la POO

Nous avons vu que l'abstraction des données est devenue un concept important dans les langages impératifs modernes. En fait, nous pouvons concevoir des programmes entiers en termes de types abstraits de données. La POO est une continuation naturelle de l'utilisation des types abstraits de données qui présentent des structures de données sur lesquelles sont définies exclusivement des opérations. La POO se base sur un concept de *classe* qui est très proche et presque similaire à un type abstrait de données. En plus du concept de classe, les concepts clés de ce paradigme de programmation sont: les objets, l'héritage et le Polymorphisme.

Un *objet* possède une ou plusieurs composantes variables (attributs), et est équipé avec des méthodes qui opèrent sur ses attributs. Les attributs de l'objet sont généralement privés, et donc peuvent être consultés que par les méthodes de l'objet. Les objets présentent un moyen pour modéliser les entités physiques ou virtuelles du monde réel d'une manière naturelle. Par exemple, considérons un programme de contrôle de vols. L'avion a un *état* composé de sa position, altitude, vitesse, charge utile, charge du carburant, …, et ainsi de suite. Notez que les lois physiques ne permettent de pas changer tout à coup l'état de l'avion : sa position, l'altitude et la vitesse changent graduellement au cours du vol, sa charge de carburant diminue graduellement, et sa charge utile ne change pas du tout. L'état de l'avion pourrait être modélisé par un objet, équipé avec des méthodes qui lui permettent de changer uniquement en conformité avec des lois physiques. C'est plus fiable que la modélisation de l'état de l'avion par un groupe de variables ordinaires, que le programme pourrait les mettre à jour de manière arbitraire.

La *classification* des objets est une caractéristique clé des langages orientés objet. Une classe est une famille d'objets ayant les mêmes composantes variables (attributs) et les mêmes méthodes. Une *sous-classe* B d'une classe A, est une classe qui possède en plus de tous les attributs et les méthodes de A, des attributs supplémentaires et / ou des méthodes supplémentaires (ou redéfinies). Chaque sous-classe peut avoir ses propres sous classes, afin que nous puissions construire une hiérarchie de classes. *L'héritage* est également une caractéristique des langages orientés objet. Une sous-classe hérite (partage), toutes les méthodes de sa superclasse, à moins que la sous-classe explicitement remplace par

redéfinition certaines méthodes. En effet, toute une hiérarchie de classes peut hériter les méthodes d'une classe ancêtre. L'héritage a un impact majeur sur la productivité des programmeurs en mettant en avant la réutilisation.

Le *Polymorphisme* est un concept clé, permettant à un objet d'une sous-classe d'être traité comme un objet de sa superclasse. Cela nous permet, par exemple, de construire un ensemble hétérogène d'objets de classes différentes, à condition que la collection soit définie en termes d'une classe ancêtre commune.

La programmation orientée objet a connu un grand succès, elle est devenue le paradigme de programmation dominant dans les années 1990s. Les raisons de son succès sont claires, l'objet offre un moyen très naturel pour modéliser les entités à la fois du monde réel et virtuel. Les classes et les hiérarchies de classes offrent la possibilité d'avoir des unités réutilisables pour construire des programmes de grande envergure. La programmation orientée objet correspond bien à l'analyse et la conception orientée objet, soutenant ainsi le développement souple des grands systèmes logiciels.

3. Pragmatiques de la POO

Les unités d'un programme orienté objet sont des classes. Les classes peuvent être liées les une aux autres par la dépendance (les opérations d'une classe appellent les opérations d'une autre classe), par l'inclusion ou l'extension (une classe est une sous-classe d'une autre classe), ou par contenance (l'objet d'une classe contient des objets d'une autre classe sous forme d'agrégation ou de composition).

Une classe est analogue à un type, dont la représentation est déterminée par les attributs de la classe. Si les attributs sont publics, ils peuvent être consultés directement par d'autres classes, donnant lieu à un couplage fort. Si les attributs sont privés, la classe est proche à un type abstrait, assurant un couplage faible; un changement de la représentation de la classe aura peu ou aucun effet sur les autres classes. Cependant, la relation d'inclusion est une source potentielle de couplage fort. Si une sous-classe peut accéder aux attributs de sa classe de base directement, les deux classes sont étroitement couplées, car tout changement aux composantes variables de la superclasse va imposer des changements à la mise en œuvre de la sous-classe. D'un autre côté, si la sous-classe ne peut pas accéder aux attributs de sa superclasse directement, elle les utilise indirectement en appelant les méthodes de la superclasse. Dans une grande hiérarchie de classes, le problème de couplage fort est significatif: tout changement aux attributs d'une classe ancêtre pourraient imposer des changements à l'ensemble des implémentations de ses sous-classes. La Figure 3.1 illustre une version orientée-objet de l'architecture du programme correcteur orthographique introduit dans le chapitre précédent en montrant les relations de dépendance et d'inclusion. En comparant cette architecture avec l'architecture impérative avec abstraction de données présentée dans la figure 2.2 du chapitre II, nous constatons que les différences principales sont : un programme orienté objet peut contenir des sous-classes ; un programme orientée objet est entièrement constitué de classes, de sorte que les procédures individuelles d'un programme impératif deviennent des méthodes de classe dans un programme orienté objet.

Par définition, un objet est un ensemble de composantes variables, équipé d'un ensemble d'opérations qui accèdent à ces variables. Donc, il est possible de mettre en œuvre un seul objet par un package dont les composantes sont des variables privées et des opérations publiques. Cependant, Dans de nombreuses applications, on a besoin de créer de nombreux objets similaires. Cela conduit naturellement à la notion de classe.

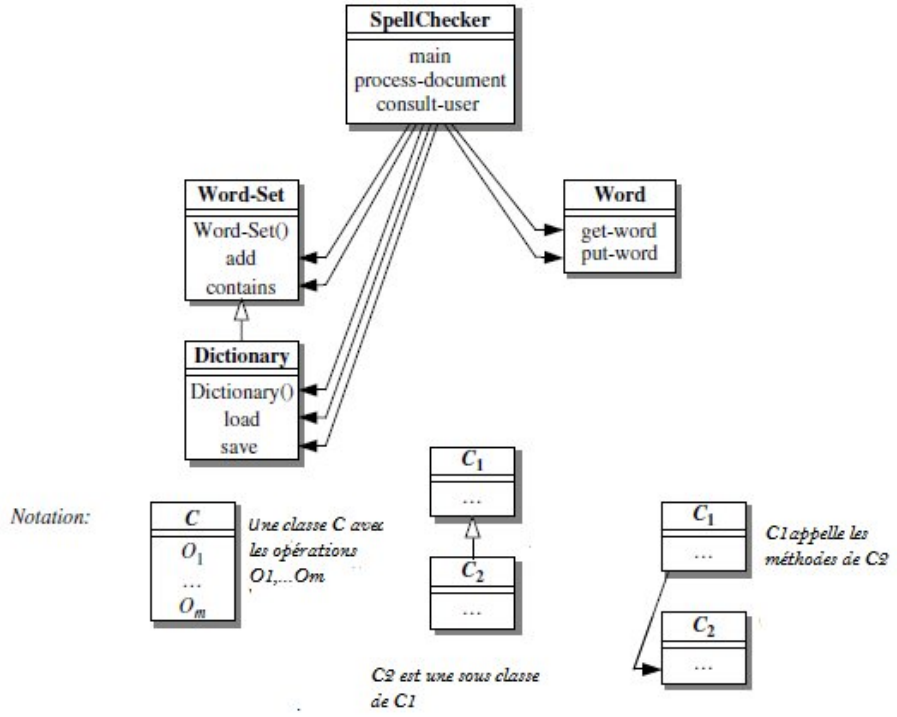


Figure 3.1 : *Une version orientée-objet de l'architecture du programme correcteur orthographique*

3.1 Classes

Une classe est un ensemble d'objets similaires. Tous les objets d'une classe donnée ont les mêmes composantes variables (attributs), et sont équipés avec les mêmes opérations. Les classes sont supportées par tous les langages orientés objet tel que C ++ et Java et indirectement par d'autres langages comme ADA95. Dans la terminologie orientée objet, les composantes variables d'un objet sont diversement appelées : variables d'instance ou des variables membres, et ses opérations sont généralement appelées constructeurs et méthodes.

Un *constructeur* est une opération qui crée (et initialise généralement) un nouvel objet de la classe. Dans les deux langages C ++ et Java, Un *constructeur* est toujours nommé d'après la classe à laquelle il appartient. Une *méthode* est une opération qui inspecte et / ou met à jour les attributs d'un objet existant de la classe.

Le code java présenté dans le listing 3.1 illustre une implémentation possible de notre exemple de dictionnaire.

```
class Dictionary {
// Un objet Dictionnaire représente un ensemble de mots.
private int size;
private String[] words;
// Ce dictionnaire est représentée comme suit: size contient le nombre de
// mots, , et words[0], . . . ,words[size-1] contiennent les mots
// dans un ordre aléatoire..
  public Dictionary (int maxsize) {
    // Construit un dictionnaire vide, avec un  espace pour maxsize mots.
    this.size = 0;
    this.words = new String[maxsize];
  }

  public void add (String wd) {
    // Ajouter le mot wd à ce dictionnaire dans le cas où le mot n'est pas
    //déjà existant.
    if (! contains(wd)) {
    this.words[this.size++] = wd;
    }
  }

  public boolean contains (String wd) {
    // Returne true si le mot wd existe dans le dictionnaire.
    for (int i = 0; i < this.size; i++) {
    if (wd.equals(this.words[i])) return true;
    }
    return false;
  }
}
```

Listing 3.1 : Une implémentation possible en Java de l'exemple "dictionnaire".

Tous les objets de la classe *Dictionary* ont comme attributs privés *size* et *words* et équipés par les méthodes publiques *add* et *contains*.

Pour créer des objets particuliers de la classe dictionnaire on doit appeler le constructeur comme il est illustré dans le listing 3.2 :

```
Dictionary mainDict = new Dictionary(10000);
Dictionary userDict = new Dictionary(1000);
```

Listing 3.2 : Instanciations de la classe *Dictionary*

L'expression " `new Dictionary (...)` " Crée et initialise un nouvel objet de la classe *Dictionary*. Le code ci-dessus crée deux objets dictionnaires, et rend les variables *mainDict* et *userdict* des poignées de ces objets. Ces deux objets sont similaires mais distincts. Ces objets sont manipulables à travers l'invocation de leurs méthodes comme illustre l'exemple présenté dans le listing 3.3.

```
if (! mainDict.contains(currentWord)
&& ! userDict.contains(currentWord)) {
. . .
userDict.add(currentWord);
}
```

Listing 3.3 : Exemple de manipulation d'objets dictionnaires

L'appel de la méthode "`UserDict. Add (currentWord)`" fonctionne comme suit : D'abord, le mécanisme d'appel prend l'objet sur lequel pointe *UserDict*, qui est appelé l'objet cible. Puis il sélectionne la méthode nommée *add* avec laquelle l'objet cible est équipé. Ensuite, il appelle cette méthode avec la valeur de *currentWord* comme argument. Dans le corps de la méthode, **this** dénote l'objet cible. Ainsi, «`this.size++`» incrémente *UserDict.size*, et "`this.words`" désigne en fait *UserDict. words*. Le code d'application ne peut pas accéder aux attributs *size* et *words* directement, car ils sont privés. D'où une écriture comme : `UserDict. Size=0`; est non autorisée.

JAVA adopte une sémantique de référence pour les tests d'égalité et pour l'affectation d'objets. Si on a deux objets de la classe *Dictionary* : *dicta* et *dictb*. L'expression `if (dicta == dictB) ...` présente un test d'égalité qui serait effectivement de comparer les références à deux objets *Dictionary* distincts. Ce test est évalué toujours à faux même si les deux dictionnaires arrivent à contenir les mêmes mots. Si nous voulons supporter des tests d'égalité appropriés des objets *Dictionary* , la classe doit fournir une méthode pour cet effet comme le montre le code du listing 3.4.

```
class Dictionary {
. . .
   public boolean equals (Dictionary that) {
  // Retourner true si ces dictionnaires contiennent les mêmes mots
    if (this.size != that.size) return false;
    for (int i = 0; i < that.size; i++) {
    if (! contains(that.words[i])) return false;
    }
  return true;
  }
}
```

Listing 3.4 : Exemple d'implémentation de test d'égalité entre objets

Ainsi, un test d'égalité entre les deux dictionnaires *dicta* et *dictb* est à implémenter par l'expression : **if** `(dictA.equals(dictB))…` .

Une autre implémentation de notre exemple de dictionnaires en C++ est l'objet des listings 3.5 et 3.6. Le listing 3.5 contient la déclaration de la classe *Dictionary* où le constructeur de la classe et les deux méthodes *add* et *contains* sont déclarés en précisant leurs

noms, les types de leurs paramètres ainsi que les types de leurs valeurs de retours. La définition du constructeur et des méthodes de la classe est présentée dans le Listing 3.6.

```
class Dictionary {
// un dictionnaire consiste en un ensemble de mots.
private:
int size;
String words[];
// Un dictionnaire est représenté comme suit : size contient le nombre de
// mots, et  les words[0], . . . , words[size-1] les mots dans un ordre
// quelconque.
public:
Dictionary (int maxsize);
// constuire un dictionnaire vide dont maxsize est la taille.
void add (String wd);
// Ajouter le mot wd au dictionnaire.

boolean contains (String wd) const;
// Retourner la valeur true dans le cas où le mot wd existe dans le
//dictionnaire.
```

Listing 3.5 : Déclaration C ++ de la classe *Dictionary*

```
Dictionary::Dictionary (int maxsize) {
this->size = 0;
this->words = new String[maxsize];
}
void Dictionary::add (String wd) {
if (! contains(wd)) {
this->words[this->size] = wd;
this->size++;
}
}
boolean Dictionary::contains (String wd) const {
for (int i = 0; i < this->size; i++) {
if (strcmp(wd, this->words[i])) return true;
}
return false;
}
```

Listing 3.6 : Implémentation C ++ de la classe *Dictionary*

Dans la déclaration de la classe *Dictionary*, les composantes variables (attributs) *size* et *words* sont déclarées privées ; le constructeur et les méthodes *add* et *contains* sont publiques. La création d'instances particulières de la classe *Dictionary*, c à d, des objets dictionnaires se fait par l'appel du constructeur comme il illustré dans le listing 3.7.

```
Dictionary* main_dict = new Dictionary(10000);
Dictionary* user_dict = new Dictionary(1000);
```

Listing 3.7 : Instanciations de la classe *Dictionary*

L'expression " `new Dictionary (...)` " Crée et initialise un nouvel objet de la classe *Dictionary*, ce qui donne un pointeur vers cet objet. Ainsi, le code ci-dessus crée deux objets dictionnaires, et rend les variables *main_Dict* et *user_dict* des pointeurs vers ces objets. Ces deux objets sont similaires mais distincts. Ces objets sont manipulables à travers l'invocation de leurs méthodes comme illustre l'exemple présenté dans le listing 3.8.

```
if (! main_dict->contains(current_word)
&& ! user_dict->contains(current_word)) {
. . .
user_dict->add(current_word);
}
```

Listing 3.8 : Exemple de manipulation d'objets dictionnaires

L'appel de la méthode " `user_dict-> add (...)` " est en fait l'abréviation de " `(* User_dict) .add (...)` ". Ainsi, l'objet cible est (* *user_dict*), à savoir, l'objet référencé par le pointeur *user_dict*. Dans le corps de la méthode, *this* désigne un pointeur vers l'objet cible. Ainsi *this-> size* référence à la *taille user_dict-> size.*

De la même manière qu'en JAVA, en C++, L'expression `if (*dicta == *dictB)` présente un test d'égalité qui serait effectivement de comparer les références à deux objets *Dictionary* distincts. Ce test est évalué toujours à faux même si les deux dictionnaires arrivent à contenir les mêmes mots. Si nous voulons supporter des tests d'égalité appropriés des objets *Dictionary*, la classe doit fournir une opération pour cet effet en faisant une *surcharge* (overloading) de l'opérateur d'égalité "==" comme le montre le code du listing 3.9. Le listing 3.10 contient une implémentation possible de cette surcharge.

```
class Dictionary {
. . .
public:
. . .
friend bool operator== (Dictionary* dict1,
Dictionary* dict2);
// Retourner true si les dictionnaires dict1 et dict2 contiennent le même
// ensemble de mots.
// Le mot clé friend précède le nom d'une fonction non-membre ou d'une autre
//classe  pour lui accorder un accès aux membres privés et protégés de la
//classe.
}
```

Listing 3.9 : *Surcharge* (overloading) de l'opérateur d'égalité "==" en c++

```
bool Dictionary::operator== (Dictionary* dict1,
Dictionary* dict2) {
if (dict1->size != dict2->size) return false;
for (int i = 0; i < dict2->size; i++) {
if (! dict1->contains(dict2->words[i]))
return false;
}
return true;
}
```

Listing 3.10 : implémentation de la *surcharge* de l'opérateur "=="

Tant en Java qu'en C++, le programmeur est autorisé à choisir parmi les composantes d'un objet celles à déclarer privées et est celles à déclarer comme étant publiques. Les composantes constantes (attributs de classes) sont souvent publiques, il est permet que ces dernières soient directement inspectées (mais évidemment pas mises à jour) par le code de l'application. Les composantes variables (attributs d'objets) doivent toujours être privées, sinon les avantages de l'encapsulation seraient perdus. Les constructeurs et les méthodes sont généralement publics, sauf pour certaines méthodes qui présentent des opérations auxiliaires à utiliser uniquement à l'intérieur de la déclaration de classe.

Certains langages de programmation orientés objet comme Java permettent la déclaration de classes abstraites. Une classe abstraite est une classe qui n'a pas de constructeur et qui n'est pas instanciée pour créer des objets, elle utilisée pour servir de superclasse dans une hiérarchie de classe.

Les concepts de type abstrait et de classe ont beaucoup de points en commun. Chacun des de concepts permet au code de l'application de créer plusieurs variables d'un type dont la représentation est privée, et de manipuler ces variables que par des opérations prévues à cet effet. Cependant, il ya des différences entre les types abstraits et les classes.

3.2 Sous classe et héritage

Une classe *C* est un ensemble d'objets similaires. Tous les objets de la classe *C* ont les mêmes attributs, et sont équipés des mêmes opérations. Une *sous classe* de la classe *C* est un ensemble d'objets qui sont semblables les uns aux autres, mais plus riche que les objets de la classe *C*. Un objet de la sous-classe possède toutes les composantes variables des objets de classe *C*, mais peut avoir des éléments supplémentaires de variables (attributs). De même, un objet de la sous-classe est équipé de toutes les méthodes de la classe *C*, mais peut être équipé avec des méthodes supplémentaires.

Si S est une sous-classe de *C*, également on dit que *C* est une superclasse de S. Les objets instances d'une sous-classe *héritent* de sa superclasse les attributs et les méthodes. Dans certaines circonstances, cependant, une sous-classe peut remplacer certaines méthodes de sa super-classe, en fournissant des versions plus spécialisées de ces méthodes.

Soit le code Java présenté dans le listing 3.11. Ce code matérialise la déclaration d'une classe *Point* en Java. Chaque point consiste en deux attributs *x* et *y*, et équipé des méthodes *distance*, *move*, et *draw*. Considérons maintenant la déclaration de la classe *Cercle* figurant dans le listing 3.12. La clause " `extends Point`" précise que la classe *Circle* (classe des cercles) est une sous-classe de la classe *Point* (et donc que la classe *Point* est la superclasse de la classe *Circle*). Chaque objet cercle est constitué des composantes variables *x*, *y* et *r*, et est équipé des méthodes nommées *distance*, *move*, *draw* et *getDiam*. La composante variable *r* figure dans la déclaration de la classe *Circle*, mais pas les composantes variables *x* et *y* qui sont héritées de la superclasse. De même, la méthode supplémentaire *getDiam* est définie dans la classe *Circle*, mais pas les méthodes *distance* et *move* qui sont héritées de la

superclasse. Nous remarquons la définition d'une nouvelle version de la méthode *draw*, qui remplace la méthode *draw* de la superclasse (redéfinition).

```
      class Point {
      // un objet point représente un point dans le plan.
      protected double x, y;
      // Le point est représenté par ces coordonnées cartésiennes(x,y).
      public Point () {
      // Construire le point à (0, 0).
      x = 0.0; y = 0.0;
      }
(1)   public double distance () {
      // Retourner la distance entre ce point et le point (0, 0).
      return Math.sqrt(x*x + y*y);
      }
(2)   public final void move (double dx, double dy) {
      // Déplacer ce point de dx dans la direction x et de dy dans direction y.
      x += dx; y += dy;
      }
(3)   public void draw () {
      // Dessiner le point sur l'écran.
      . . .
      }
      }
```

Listing 3.11 Implémentation Java de la classe *Point*

```
      class Circle extends Point {
      // Un objet Circle représente un cercle dans le plan.
      private double r;
      // Le cercle est représenté par les cordonnées de son centre
      // (x,y) et par le rayon (r).
      public Circle (double radius) {
      // Construire un cercle de rayon r centré à (0, 0).
      x = 0.0; y = 0.0; r = radius;
      }
      (4) public void draw () {
      // Dessiner le cercle sur écran.
      . . .
      }
      (5) public double getDiam () {
      // Retourner le diamètre d'un cercle.
      return 2.0 * r;
      }
      }
```

Listing 3.12 : Implémentation Java de la classe Circle.

Un attribut privé est visible que dans sa propre classe, tandis qu'un attribut publique est visible dans toutes autres classes du programme. Un attribut protégé **`protected`** est visible non seulement dans sa propre classe, mais également dans toutes les sous-classes. L'état protégé est donc intermédiaire entre l'état privé et l'état publique.

Une sous-classe peut avoir des sous-classes. Cela donne lieu à une hiérarchie de classes. L'héritage est important car il permet aux programmeurs d'être plus productifs. Une fois qu'une méthode a été mise en œuvre pour une classe particulière *C*, cette méthode est automatiquement applicable non seulement aux objets de la classe *C*, mais aussi à des objets de toutes sous-classes qui héritent la méthode de *C* (et leurs sous-classes, et ainsi de suite). Dans la pratique, ce gain de productivité du programmeur est important.

Chaque méthode d'une classe de base peut être soit héritée ou redéfinie par une sous-classe. En Java par exemple, les méthodes sont hérité par défaut ; et donc, la superclasse et la sous-classe partages les mêmes méthodes. Si une classe de base et une sous-classe sont équipées d'une méthode avec le même identifiant, les mêmes types de paramètres et les mêmes types des valeurs de retours, on dit que la méthode est redéfinie. Dans ce cas les deux classes sont équipées de deux versions différentes de la méthode. Une méthode est redéfinissable si une sous-classe est autorisée à la redéfinir. En Java, une méthode est redéfinissable si elle n'est pas déclarée : **`Final`**. En C++ une méthode est redéfinissable si elle est déclarée **`virtual`**.

Les listings 3.13 et 3.14 contiennent une implémentation C++, des classes *Point* et *Circle* implémentées en Java dans les listings 3.11 et 3.12.

```
class Point {
// un objet point représente un point dans le plan.
protected:
double x, y;
public:
Point ();
virtual double distance ();
void move (double dx, double dy);
virtual void draw ();
};
```

Listing 3.13 : Implémentation C++ de la classe *Point*

```
class Circle : public Point {
private:
double r;
public:
Circle (double radius);
void draw ();
double getDiam ();
}
```

Listing 3.14 : Implémentation C++ de la classe *Circle*.

L'héritage entre les classes peut être *simple* où chaque classe peut avoir une seule superclasse. Dans le cas où une classe peut avoir plusieurs super classe on parle d'héritage *multiple*. L'héritage simple est supporté par tous les langages de programmation orientés objet. Par contre, la majorité des langages de programmation orientés objet ne supportent pas l'héritage multiple. Par exemple, En Java, une sous-classe ne peut avoir qu'une seule superclasse. Le langage C++, propose des supports pour l'héritage multiple comme il est illustré dans l'exemple du listing 3.15 implémentant une partie de l'hiérarchie de classe présentée dans la figure 3.2.

```
class Animal {
        private:
        float poids;
        float vitesse;
        public:
        . . .
        }

       class Mammifere: public Animal {
       private:
       float durée_gestationnelle;
       public:
        . . .
        }
       class Oiseau : public Animal {
       private:
       float taille_des_oeufs;
       public:
        . . .
       }
       class aviateur: public Animal {
       private:
       float envergure_des_ailes;
       public:
       . . .
       }
       class felin : public Mammifere {
       public:
        . . .
        }
        Class chauve-souris: public Mammifere, public aviateur {
        private:
        float sonar_range;
        public:
        . . .
        }
```

Listing 3.15 : Héritage multiple en C++

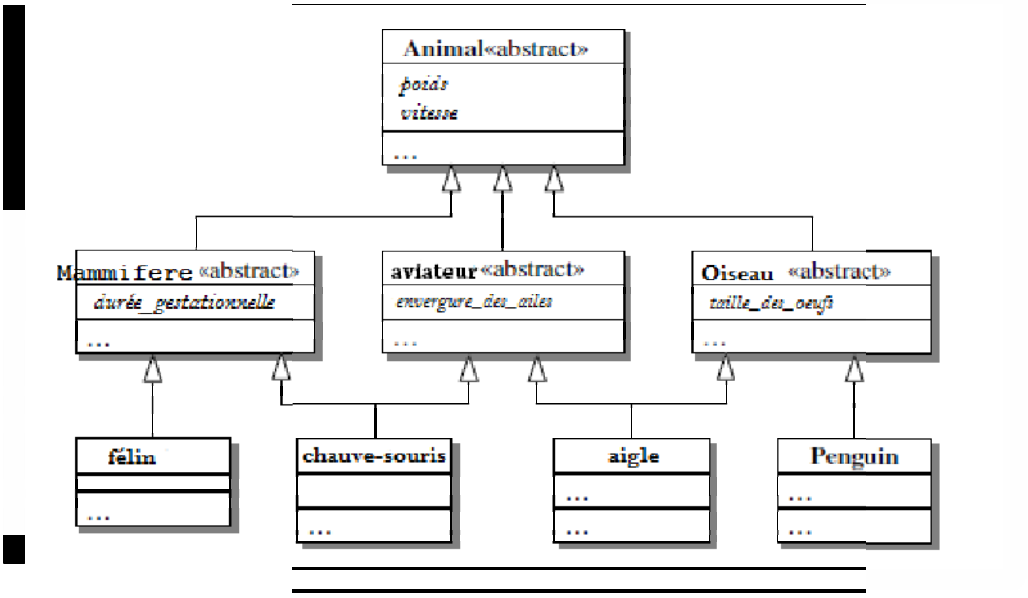


Figure 3. 2: Une hiérarchie de classe avec héritage multiple

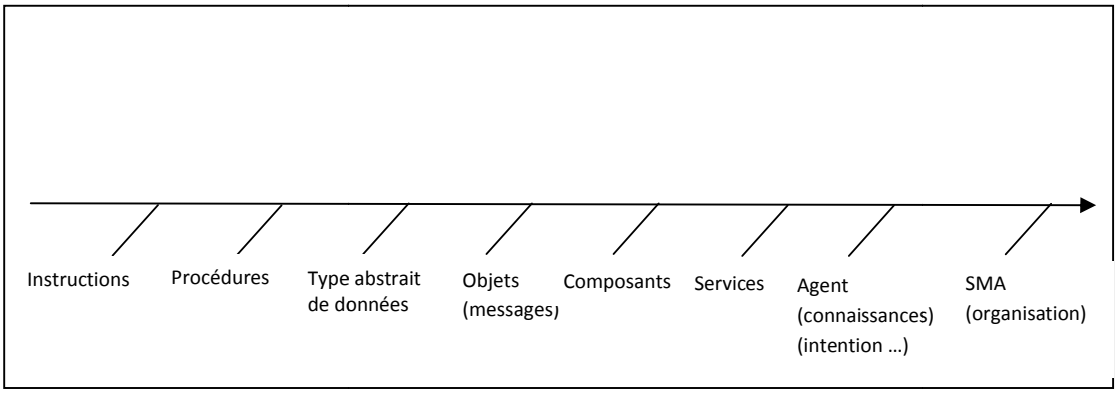


Figure 3.3 : *Evolution du niveau d'abstraction en programmation*

Les langages orientés objet permettent aussi l'abstraction générique à travers les classes génériques. Une classe générique est une classe qui peut être réutilisée pour des objets de différents types. Elle peut être paramétrée à l'égard de valeurs, variables, types, et les fonctions dont elle dépend. Le langage C++ permet la déclaration de classe générique ainsi que la définition de fonctions génériques. Le code du listing 3.16 implémente une classe générique en C ++ encapsulant un type abstrait dont les valeurs sont des files d'attente limités de caractères. La classe est paramétrée par rapport à la capacité de la file d'attente. La clause <int capacity> entre les crochets indique que *capacity* est un paramètre formel de la classe générique, et qu'il représente un nombre entier de valeur inconnue. Ce paramètre formel est utilisé dans les déclarations des composantes de la classe.

```
template
<int capacity>
class Queue {
// Un objet Queue est une file d'attente dont les  éléments sont des
// caractères ; la langueur maximum de la file est capacity.
private:
char elems[capacity];
int front, rear, length;
// La file est représentée par un tableau cyclique, les éléments sont
// stockés dans : elems[front..rear-1] ou  dans
// elems[front..capacity-1] et elems[0..rear-1].
public:
Queue ();
// construire une file d'attente vide.
void add (char e);
// Ajouter l'élément e dans la queue de la file.
char remove ();
// Enlever et retourner l'élément du début de la file.
}
```

Listing 3.16 : Implémentation d'une classe générique en C ++

Le listing 3.17 contient une autre version de la classe générique *Queue*. Cette fois la classe est paramétrée par rapport au type d'éléments de la queue et par rapport à une fonction *less* déterminant une priorité. Les définitions des méthodes *add* et *remove* utiliseront la fonction *less* pour comparer les valeurs de deux éléments. La classe générique *Queue* peut être instanciée par un code similaire à celui du listing 3.18 où une définition de type génère une classe ordinaire, nommé *Print_Queue* à partir de la classe générique en substituant *Print_Job* pour *Element*, et *Earlier* pour *Less*.

```
template
<class Element,
bool less (Element x, Element y)
>
class Priority_Queue {
// un objet Priority_Queue est une file d'attente à priorité, dont
// les éléments sont de n'importe quel type et équipée d'une fonction de
//priorité.
private:
. . . // representation
public:
Priority_Queue ();
// Construire une file vide.
void add (Element e);
// Ajouter e dans la file.
Element remove ();
// Supprimer le dernier élément de la file.
}
```

Listing 3.17 Implémentation d'une classe générique en C ++

```
struct Print_Job {
int owner_id;
int timestamp;
char* ps_filename;
}
bool earlier (Print_Job job1, Print_Job job2) {
return (job1.timestamp < job2.timestamp);
}
typedef Priority_Queue<Print_Job, earlier>
Print_Queue;
```

Listing 3.18 : Une instanciation de la classe générique *Priority_Queue*

4. La POO dans le cadre d'analyse de l'évolution de la programmation

Comme il est mentionné dans l'introduction de ce chapitre, pour mieux approcher la programmation orientée objet, il est intéressant de la situer dans un cadre d'une perspective générale d'évolution de la programmation. Ce cadre est un repère dont la première dimension représente le niveau d'abstraction alors que la seconde représente le couplage entre différentes entités.

Sur le plan de l'abstraction trois éléments semblent essentiels pour définir l'espace dans lequel nous pouvons étudier et comparer les concepts et les technologies ayant un apport dans la programmation et le développement de logiciel d'une manière générale. Le premier élément représente les possibilités offertes pour identifier des abstractions de plus en plus de haut niveau ; le deuxième élément résume l'évolution dans la représentation et l'explicitation des données et de connaissances manipulées ; le troisième élément concerne l'évolution en termes de représentations et d'explicitations des données et des connaissances échangées et communiquées.

La figure 3.3 illustre les positions dans un axe présentant l'évolution du niveau d'abstraction en programmation des différents concepts servant de base aux différents paradigmes de programmation comme les objets, les composants, les agents et les systèmes multi agents. D'une manière générale la programmation est passée de la manipulation des données à la manipulation des concepts. Les objets représentent plus que des structures de données, ils représentent des objets du monde réel et introduisent le concept de message qui est un concept de haut niveau d'abstraction. Les langages de programmation orientés objet comme C++ et Java permettent une abstraction des données à travers le concept de classe qu'ils supportent. Ces deux langages de programmation permettent la distinction entre les attributs de classes et ceux des instances.

La mesure de la flexibilité du couplage peut être faite selon deux volets. Un premier volet dans lequel la mesure concerne l'aspect de sélection de l'action à exécuter et un deuxième qui concerne la flexibilité du couplage.

Pour la sélection de l'action à exécuter, la programmation au début de son histoire était simple. Les différentes instructions sont identifiées par l'intermédiaire de leurs numéros de lignes, ce que signifie que la sélection de l'action à exécuter est exprimée globalement d'une manière statique. La situation est améliorée légèrement avec la programmation structurée où la modularité et l'encapsulation du code ont un impact sur la sélection de l'action à exécuter dans le sens où l'action à exécuter est exprimée via un nom symbolique et non par un numéro de ligne. Les langages de programmation orientés objet apportent une innovation importante avec la réunion des données et des procédures appelées méthodes. La liaison tardive était une avancée déterminante, c'est-à-dire la méthode à invoquer sera déterminées en fonction de la classe de l'objet effectivement invoqué et non pas en fonction de la déclaration. Nous pouvons remarquer que la sélection de l'action à exécuter est repoussée à l'exécution et non résolue statiquement.

L'aspect couvert par le deuxième volet de la flexibilité du couplage concerne les concepts architecturaux de mise en relation structurelle entre entités logicielles et les modes de communication entre celles-ci. La manipulation du couplage entre les objets n'est pas externe aux objets, un objet référence un autre par avoir un attribut dont la valeur est l'identifiant de l'objet référencé ce qu'implique des modifications à l'intérieur de l'objet en cas de besoin de modification de liaisons.

5. Conclusion

Dans ce chapitre Nous avons étudié le paradigme de la programmation orientée objet. Cette étude consiste principalement en un rappel sur les concepts de base de ce paradigme de programmation, les pragmatiques de la programmation orientée objet ainsi qu'une analyse dans laquelle on a fait une tentative pour situer la programmation orientée objet dans un cadre exprimant l'évolution de la programmation. Les empreintes de la POO en termes d'abstraction et de flexibilité de couplages (dimensions du cadre d'analyse) sont bien importantes ; mais, ce paradigme de programmation n'est pas en mesure de répondre aux besoins progressifs en termes d'abstraction et de flexibilité de couplages. Ces besoins ainsi que d'autres ont conduit à la proposition d'autres paradigmes tels le paradigme *composant*, le paradigme *service* et le paradigme *agent* dans lesquels des niveaux d'abstraction plus élevés sont atteints et des modes plus flexibles de couplages sont accédés. Le prochain chapitre fait l'objet de l'étude de ces paradigmes de programmation.

Chapitre IV :

Les paradigmes : composants, services et agents.

1. Introduction

Depuis des années le monde a connu une énorme utilisation des logiciels dans tous les domaines tels que l'industrie, l'enseignement, l'administration et même dans plusieurs aspects de la vie quotidienne. Cela se traduit en une augmentation des exigences sur la production du logiciel, on demande de plus en plus des logiciels fiables, flexibles, robustes, adaptables, mieux exploitables et que l'on peut facilement installer et déployer. Cette demande grandissante du logiciel ainsi que l'augmentation sur le plan des exigences de qualités ont provoqué une grande complexité à la fois au niveau des logiciels eux mêmes qu'aux niveaux des processus de développement de ces derniers. Comment vaincre cette complexité ou au minimum la réduire ? Comment adapter rapidement des logiciels face aux changements ? Et finalement, comment prendre en compte l'évolution du logiciel dés son développement ? Ces points présentent des défis très importants pour les développeurs de logiciels et de systèmes informatiques de manière générale. A cela s'ajoute un besoin progressif qui n'a pas cessé d'augmenter en termes de recherches de concepts de niveaux d'abstractions de plus en plus élevés et des modes plus flexibles de couplages entre ces concepts. Malgré que la programmation orientée objet a apporté quelques éléments de solutions pour ces points, ce paradigme de programmation reste loin de répondre aux exigences derrière chaque points.

La réutilisation présente une clé de solutions pour ces problèmes. L'idée de réutiliser des logiciels en tous ou en partie est très ancienne. Malgré quelques succès, la réutilisation n'est pas devenue une force motrice pour le développement du logiciel. Dans plusieurs approches comme en programmation orientée objet, la réutilisation n'est pas unie avec le processus de développement tout en constatant l'absence de définitions exactes de ce qui est réutilisable d'un côté et l'absence de formalisations du comment introduire des changements

aux éléments réutilisables d'un autre. L'approche de développement par composants (*component-based development (CBD)*) et l'approche de développement orientée services rétablirent l'idée de réutilisation en introduisant de nouveaux éléments. Dans ces approches, le logiciel est construit par assemblage de composants (approche de développement par composants) ou services (approche de développement orientée services) développés auparavant et préparés pour être intégrés. Cela influence très considérablement sur la gestion de la complexité, l'augmentation de la productivité et l'amélioration des qualités des logiciels.

L'approche de développement par agents ou orientée agents est aussi un autre paradigme de programmation ayant un impact important sur le développement de logiciels. Tout comme les approches composants et services, l'approche agent propose des abstractions pour organiser le logiciel comme une combinaison d'éléments logiciels, avec pour objectifs communs : réduire la complexité ; assurer une meilleure structuration du logiciel ; et de faciliter son évolution. Dans l'approche composant, le logiciel est organisé comme une combinaison de composants ; Pour l'approche service, le logiciel est organisé comme une combinaison de services ; pour l'approche agent, le logiciel est structuré sous forme d'une combinaison d'agent.

Le présent chapitre fait l'objet d'études des approches de développement à base de composants, de services et d'agents ainsi que la majorité des notions et des concepts en relation avec les concepts composants, services et agents. Également, ce chapitre propose une analyse des apports de ces approches en les situant dans le cadre d'analyse introduit dans le chapitre précédant et qui présente une perspective générale de l'évolution de la programmation.

2. Approche de développement par composants

L'approche de développement par composants ou basé composants (*component-based development (CBD)*) rétablie l'idée de réutilisation en introduisant de nouveaux éléments. Dans cette approche le logiciel est construit par assemblage de composants développés auparavant et préparés pour être intégrés.

Les développeurs du logiciel ainsi que leurs clients ont attendu beaucoup du CBD, mais les expériences ont montré que le développement à base de composant exige une approche systématique pour se concentrer sur les aspects du composant pendant le développement de logiciels. Les techniques du génie logiciel traditionnel doivent être ajustées à cette approche et de nouvelles procédures doivent être développées. Le génie logiciel à base de composants (*Component-based software engineering* (CBSE)) est reconnu comme nouvelle sous-discipline du génie logiciel où principalement on s'intéresse à fournir des supports pour le développement de composants autant qu'entités réutilisables, des supports pour l'assemblage de composants ainsi que des supports pour la maintenance et la mise à niveau des applications à base de composants par remplacements où personnalisations de composants.

2.1 De la programmation orientée objet vers la programmation à base de composant

La programmation par objets, qui a émergé depuis les années 1990s s'est imposée comme une évolution majeure par rapport à la programmation procédurale pour répondre à des exigences liées à la complexité et de la taille croissante des systèmes logiciels développés et des nouvelles exigences en termes de qualité et de fiabilité.

Le paradigme objet apporte, en effet, de bonnes propriétés, telles qu'un niveau d'abstraction plus élevé (concepts de classes et d'instances), une meilleure réutilisabilité du code et flexibilité de conception (notion de canevas logiciels avec l'héritage et le polymorphisme), l'encapsulation des données et du comportement avec la possibilité de séparer les interfaces et l'implémentation. Cependant, l'objet n'est pas suffisamment abstrait et reste à un niveau de granularité trop fin pour maitriser aisément la structuration des systèmes complexes. Cette structuration est extrêmement nécessaire à la maintenance des systèmes logiciels. Un concept d'architecture logicielle est ainsi défini comme : « *L'architecture d'un système logiciel définit le système en terme de composants et interactions entre ces composants. Les exemples des composants incluent des clients, des serveurs, des filtres et des couches d'un système hiérarchique. Les interactions entre les composants peuvent se composer des appelles de procédures, des variables partagées, et des événements asynchrones*». La prise en compte des limitations de l'approche objet pour la conception et l'implémentation des architectures logicielles et le besoin d'une structuration à granularité variable, éventuellement à grosse granularité, ont mené à la création d'un nouveau paradigme de programmation : le composant.

L'apport des composants par rapport à l'objet est notamment une plus grande modularité et facilité de réutilisation grâce à une encapsulation plus forte (notion de boîte noire) et un couplage plus faible entre entités logicielles avec une représentation explicite des dépendances requises et fournies.

La séparation des préoccupations entre architecture logicielle et implémentation est un point fort de la programmation par composants pour mieux structurer les logiciels sous forme d'assemblage de briques logicielles.

Le cycle de vie du logiciel peut ainsi être plus clairement décomposé en des phases de conception de l'architecture, de développement du code métier des composants, du déploiement de ces composants et enfin de leur exécution.

2.2 Définition d'un composant

Le composant est un paradigme de programmation en génie logiciel, au même titre que l'objet, pour construire des systèmes logiciels. D'une manière simple nous pouvons considérer un composant schématisé dans la figure 4.1 comme une unité réutilisable pour la

composition et le déploiement. Dans la littérature plusieurs définitions ont été proposées pour ce concept qui est au cœur du CBSE.

L'une des définitions qui semble avoir un consensus, c'est celle de Szyperski dans laquelle un composant est : « *Une unité de composition avec des interfaces spécifiées contractuellement et seulement des dépendances explicites vis à vis de son contexte. Un composant logiciel doit pouvoir être déployé indépendamment et fait l'objet de composition par des tiers.* ».

En analysant cette définition, on peut constater les points suivants: le composant communique avec son environnement à travers des interfaces, par conséquent celles-ci doivent être spécifiées clairement tout en encapsulant l'implémentation à l'intérieur du composant. Cette séparation entre spécification et implémentation n'est pas similaire à celle que nous pouvons voir dans certains langages de programmation tel que ADA où les déclarations sont séparées des implémentations, ou dans des langages de programmation orientés objet où les définitions des classes sont séparées de leurs implémentations, la différence réside dans les besoins d'intégration du composant dans une application, l'intégration du composant doit être indépendante de son cycle vie, en d'autre termes il n'est pas nécessaire de recompiler une application quand on ajoute un nouveau composant. Pour que le composant peut être *déployé indépendamment* il est nécessaire de faire une distinction claire entre le composant et son environnement ainsi qu'entre le composant et les autres composants.

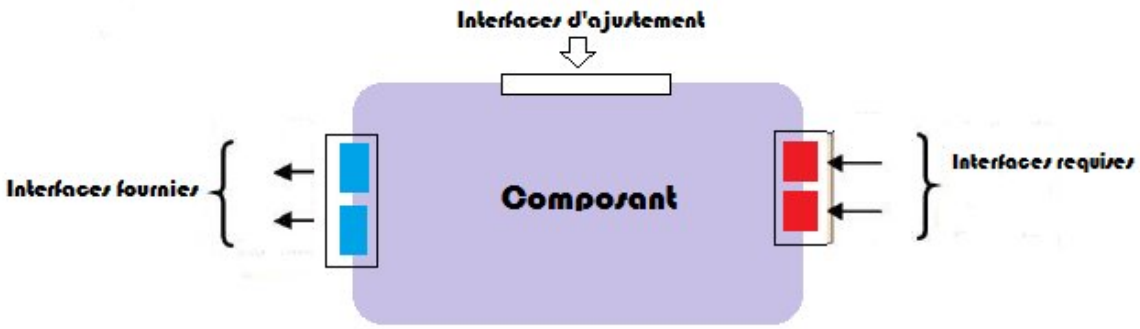


Figure 4.1 : *Composant logiciel*

Il existe plusieurs définitions pour le concept composant, dans chacune l'accent est mis sur un ou plusieurs aspects. Tous le monde est d'accord sur quelques points, le composant est une unité de composition, il doit être spécifié pour être composé avec d'autres composants ou intégré dans des applications ; les composants sont réutilisables, la réutilisation dans le génie logiciel à base de composant est différente de celle que nous pouvons trouver dans la technologie orienté objet ou d'autre technologies liées au génie logiciel traditionnel. Cette différence réside dans le fait qu'un composant peut être utilisé au moment de l'exécution « *run time* » sans le besoin de recompilation, le deuxième argument de cette différence apparait parce que le composant détache ses interfaces de son

implémentation ce qui permet la composition sans obligation de savoir aucun détail sur l'implémentation du composant.

2.3 Interfaces d'un composant

Une interface d'un composant est définie comme la spécification de ses points d'accès. Ces points sont utilisés par les clients du composant pour accéder aux services fournis par ce dernier. Une interface ne fourni aucun détail d'implémentation de ses opérations. Au lieu de cela, dans une interface on se contente de nommer une collection d'opérations et fournir des descriptions et des protocoles pour ces opérations. Ce mécanisme augmente l'adaptabilité du composant ainsi que ses performances dans le sens où nous pouvons ajouter de nouvelles interfaces et implémentations sans modifier les implémentations existantes ou même remplacer la partie implémentation de certains composants sans reconstruire un système entier.

Les interfaces servent aussi pour permettre la personnalisation d'un composant. Les interfaces définies dans des technologies standardisées peuvent exprimer des propriétés fonctionnelles dans lesquelles sont inclus les signatures des opérations ainsi que les comportements des composants. La majorité des techniques pour la description des interfaces comme les langages de définition d'interfaces (IDL) ne mettent l'accent que sur la partie signature (que les aspects syntaxiques). Ces techniques soufrent aussi de certaines incapacités pour décrire les propriétés non fonctionnelles telles que l'exactitude, la précision, la disponibilité, la sécurité, et la latence. Les interfaces d'un composant sont partitionnées en : les interfaces *fournies* qui décrivent les services fournis par le composant, Et les interfaces *requises* qui spécifient les services requis par le composant pour qu'il puisse fournir les services qu'il assure.

2.4 Spécifications des composants logiciels

Les composants logiciels sont accessibles qu'à travers leurs interfaces. Celles-ci doivent fournir toutes les informations nécessaires pour leurs utilisations et leurs déploiements dans différents contextes tout en cachant tous les détails des opérations que le composant implémente. La spécification d'un composant revient en la spécification de ses interfaces. Cette spécification regroupe la définition des opérations du composant ainsi que les dépendances aux contextes, c'est-à-dire, les aspects fonctionnels du composant ainsi que ses propriétés non fonctionnelles. Toutes ces informations sont utiles aussi pour les développeurs des composants dans le sens où la spécification d'un composant peut servir comme cadre pour la définition abstraite de sa structure interne.

2.4.1 Spécifications fonctionnelles des composants

Les spécifications des composants concernent les aspects fonctionnels ainsi que les aspects extra fonctionnels du composant. Puisque un composant est visible qu'à travers ses interfaces la spécification de ce dernier revient en une spécification de ses interfaces. Cette

spécification doit définir d'une manière précise et complète toutes les opérations ainsi que toutes dépendances aux contextes. Les techniques de spécifications utilisées dans ce domaine se devisent en deux classes. Des techniques limitées en ce que est appelées spécifications syntaxiques et des techniques qui permettent de décrire des aspects concernant la sémantique de ce que fait le composant.

2.4.1.1 Spécifications syntaxiques

Les composants logiciels implémentent et fournissent un ensemble d'interfaces ou de types. Chaque interface regroupe un ensemble d'opérations où chaque opération porte un nom et a un ensemble de paramètres en entrée et \ou en sortie. Les techniques de spécification syntaxiques associent des types pour tous ces éléments. Certaines techniques permettent de spécifier si le composant a besoin d'interfaces qui sont implémentées par d'autres composants. Les technologies de composant comme COM de Microsoft, ou CORBA (Object Management Group's Common Object Request Broker Architecture) et les JAVABeans de Sun permettent de développer et de déployer des composants réutilisables où les spécifications sont principalement syntaxiques. Pour les deux premières technologies, des langages de définition d'interfaces IDL sont utilisés alors pour les JAVABeans les interfaces sont décrites en JAVA.

Le listing 4.1 illustre un exemple de contenu d'un fichier de spécification d'un composant COM. Deux interfaces sont spécifiées contenant trois opérations qui constituent les fonctionnalités d'un composant vérificateur orthographique simple. Les deux interfaces héritent de l'interface COM standard IUNKOWN. Toutes les opérations retournent une valeur de type HRESULT pour indiquer un échec ou un succès de l'exécution de l'opération. Dans cette spécification on a une seule instance du composant associée à deux instances d'interfaces. La première interface est associée à une seule opération, cette dernière à son tour est associée à une seule instance de paramètre d'entrée et deux instances de paramètres de sortie.

Nous pouvons constater que cette spécification peut fournir des informations sur les points suivants : quelles sont les opérations fournies, quel est le nombre de paramètres et quels sont leurs types ; mais aucune information sur l'effet de l'invocation d'une opération n'est fournie. Les spécifications syntaxiques présentent des insuffisances pour traiter la substitution et l'évolution de composants. La substitution consiste en la possibilité de remplacer un composant par un autre. D'un point de vue syntaxique cette substitution n'est possible que si le nouveau composant implémente les mêmes interfaces que le composant à remplacer ou les interfaces du nouveau composant sont des sous-types des interfaces du composant à remplacer. L'évolution d'un composant est vue comme un cas spécial de la substitution.

```
interface ISpellCheck : IUnknown
{
  HRESULT check([in] BSTR *word, [out] bool *correct);
};

interface ICustomSpellCheck : IUnknown
{
  HRESULT add([in] BSTR *word);
  HRESULT remove([in] BSTR *word);
};

library SpellCheckerLib
{

  coclass SpellChecker
  {
      [default] interface ISpellCheck;
      interface ICustomSpellCheck;
  };
};
```

Listing 4.1 : Exemple de spécification d'un composant COM

2.4.1.2 Spécification des aspects sémantiques

Pour utiliser effectivement les composants, les spécifications syntaxiques ne suffisent pas seules, il est claire que l'on a besoin d'informations sémantiques sur les composants tels que les codes d'erreurs possibles pour chaque opération, les contraintes sur l'ordre d'invocations des opérations et les combinaisons de valeurs et paramètres que les opérations acceptent.

Depuis que l'idée de développer des logiciels à partir de bibliothèques de composants logiciels produits en masse est annoncée, plusieurs méthodes pour concevoir des composants ont été proposées, la majorité des méthodes ont la tendance d'inclure des informations sémantiques dans la spécification des composants. Par exemple la méthode *Design By Contrat* ainsi que plusieurs autres méthodes proches associent pour chaque opération des pré-conditions et des post-conditions qui représentent des assertions qui doivent être satisfaites avant l'invocation de l'opération et les assertions que le composant garantie après son invocation respectivement. En plus des pré-conditions et les post-conditions, le contrat liste les contraintes globales que le composant doit maintenir (invariants). Une assertion est une formule logique, un prédicat, permettant d'exprimer ce que le composant doit faire alors que le code exprime comment il doit le faire. Les assertions correspondent donc à la spécification de l'implantation. Un contrat peut se décomposer en quatre niveaux différents comme illustre la figure 4.2 : un niveau syntaxique où le contrat n'est assuré que par les signatures des opérations ; le deuxième niveau : contrat par contraintes, où il est décrit pour chaque service offert les conditions d'utilisation et le

résultat attendu ; le troisième niveau : contrat par synchronisation, garantit la bonne marche du système en cas d'utilisation concurrente de l'interface ; et finalement le quatrième niveau qui représente le contrat de qualité de service. Ce type de contrat décrit les conditions d'utilisation de l'interface permettant de garantir la QoS de l'interface. L'utilisation de telles assertions dépond d'un état du composant que ce dernier doit maintenir. Pour cela, pour chaque interface du composant sont ajoutées des informations qui présentent une partie de l'état du composant qui affecte et qui est affectée par l'invocation des opérations de l'interface.

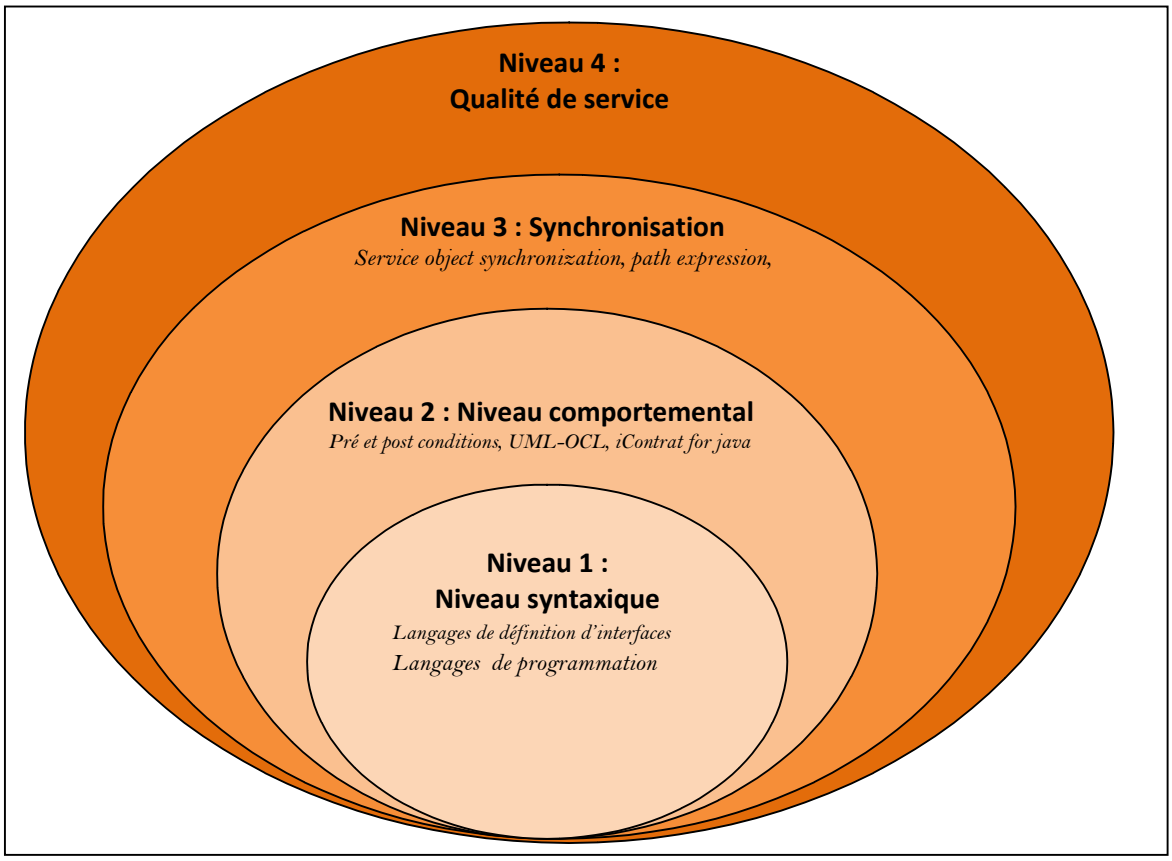


Figure 4.2 : *Les quatre niveaux de description d'un contrat.*

Sous le titre *« UML components »* J.Cheesman et J.Daniels ont publié un ouvrage pour proposer la méthode ***UML component*** dans laquelle ils utilisent UML et OCL (*Object Constraint Language*) pour écrire des spécifications de composants. Les auteurs utilisent la notation UML pour définir le concept du contrat d'assemblage que chaque composant doit remplir pour être intégré facilement dans un système à base des composants existant. La méthode a enrichi la notation de UML avec des stéréotypes pour désigner la spécification des composants, par exemple : *Component specification*, *Subcomponent*, *Date Type*, *Interface Type* et *Information Type*. D'un autre côté OCL est utilisé pour vérifier et contrôler les pré-conditions et les post-conditions des opérations qui caractérisent le comportement de chaque composant.

Une autre méthode dont ***UML component*** est inspirée a été proposée par D'Souza et A. Wills. La méthode est appelée ***Catalysis*** . Dans *Catalysis* un composant est défini comme un package logiciel cohérant, qu'on peut développer pour la construction ou l'extension d'un système plus large. Les composants de la méthode sont abstraits, et définis comme des 'Types'. chaque type est une classe stéréotypée. Un type présente un comportement dans un domaine. Le comportement externe de chaque 'Type' est défini par ses interfaces.

La méthode ***Component Unified Process*** et qui présente une adaptation du *Processus Unifié* orienté vers les systèmes à base de composants a été aussi une source d'inspiration pour les auteurs de ***UML component.*** La méthode utilise la notation UML pour exprimer le concept de composant durant le cycle de vie de développement des logiciels. L'accent est mis sur l'obligation de l'omniprésence du composant dans tout le cycle de conception du système, car dans la notation classique d'UML on trouve les composants dans la phase de déploiement.

Plusieurs autres méthodes peuvent également servir d'exemples où les spécifications des composants et leurs interfaces incluent des informations sémantiques. Nous pouvons citer quelques méthodes comme la méthode ***Business Component Factory*** , la méthode ***KobrA*** , et la méthode ***Rational's Unified Process*** proposée par Jacobson, Booch, et Rumbaug.

2.4.2 Spécifications des propriétés extra-fonctionnelles des composants

Pour réussir une bonne utilisation des composants on a besoin d'informations supplémentaires autres que les spécifications fonctionnelles. Ces informations peuvent être qualifiées de propriétés non fonctionnelles ou extra fonctionnelles.

Par propriétés extra fonctionnelles nous désignons les propriétés des composants qui déterminent leurs comportements, mais ces propriétés ne sont pas décrites sous forme de fonctions et par la suite elles ne sont pas invocables. Ces propriétés regroupent des contraintes temporelles comme le temps d'exécution et la latence ainsi que des contraintes liées à la fiabilité, la performance, l'efficacité, la synchronisation et la sécurité. Au niveau système, d'autres propriétés peuvent être considérées, ces propriétés sont en relation avec la maintenance, l'adaptabilité et l'utilisation du système. Quand on traite les propriétés extra fonctionnelles, l'un des problèmes les plus importants qui se pose est la détermination des relations entres les propriétés des composants et celles du système. Ainsi que la détermination des propriétés à considérer lors de l'évaluation des composants et lors de leurs compositions.

Depuis longtemps le besoin d'enrichir les spécifications des composants par des propriétés extra fonctionnelles a été reconnu. Plusieurs propositions ont vu le jour. Nous pouvons citer comme exemple le concept de ***FURPS scheme*** incorporé dans ***Rational Unified Process framework*** , ***UML QoS*** et ***Fault Tolerence Profile*** . Parmi les concepts proposés pour décrire les propriétés non fonctionnelles, figure le concept ***Credential*** qui

est un triplet (Attribut, valeur, crédibilité) où l'attribut représente la propriété, la valeur représente une mesure sur la propriété et la crédibilité indique la manière selon laquelle la mesure est obtenue ; ce concept a été incorporé dans l'approche de construction de système à partir de composant préexistants ***ensemble***. Les travaux portant sur les besoins non fonctionnel en génie logiciel ainsi que les descripteurs de propriétés non fonctionnelles peuvent également servir d'exemples.

2.5 Catégorisation des composants logiciels

Les composants peuvent être vus de différentes manières, leurs classifications dépendent de plusieurs points de vue comme le niveau de transparence, le niveau d'abstraction, la spécialisation, et tout autres points de vue permettant une catégorisation des composants. L'utilité des classifications des composants réside dans la possibilité de mieux entourer leurs utilisations.

Selon le degré de transparence signifiant un degré de visibilité et de détails apparents, les composants sont répartis en composants boites noires, composants boites blanches et composants boites grises.

- *Composants boites noires* : des composants dont le code binaire est vendu avec un mode d'emploi et des spécifications. La réutilisation d'un composant boîte noire est une sorte de réutilisation d'une mise en œuvre sans tenir compte sur autre chose que son interface et ses spécifications.

- *Composants boites blanches* : Totalement opposé aux composants boites noires, la solution dans le cas de composant boîte blanche est de fournir tout le code du composant. Dans ce cas, on constate une perte au niveau de possibilités de réutilisation dans le sens où il devient très peu probable que les composants peuvent être remplacés par de nouvelles versions car ceux-ci peuvent dépendre de certains détails d'implémentation qui peuvent être sujet de changement dans les nouvelles versions.

- *Composants boites grises* : il s'agit d'une solution intermédiaire entre les composants boites noires et les composants boites blanches. Les composants de cette catégorie dévoilent une partie contrôlée de leurs mises en œuvre dans le cadre de la spécification.

Selon la spécialisation des composants, trois types principaux de composants peuvent être distingués: les composants conceptuels, les composants logiciels et les composants métiers.

– *Les composants conceptuels*: Un composant conceptuel est une solution à un problème conceptuel sous la forme d'un modèle (ou une partie d'un modèle) destinée à être réutilisée. Cette solution peut être spécifiée avec un langage de modélisation

comme UML. Les composants conceptuels peuvent être des *composants produits* ou des *composants processus*. Un composant produit est une partie cohérente d'un modèle qui peut être réutilisée avec d'autres composants produits pour assembler un modèle complet. Un produit correspond au but à atteindre lorsqu'on utilise la solution offerte par le composant produit. Un composant processus est une partie cohérente d'un processus qui peut être réutilisée avec d'autres composants processus pour assembler un processus complet. Un processus correspond au chemin à parcourir pour atteindre la solution offerte par le composant processus.

– *Les composants logiciels* : un composant logiciel est défini comme un paquetage cohérent d'implantations logicielles et qui peut être indépendamment développé et délivré. Il possède des interfaces explicites spécifiant les services offerts et les services requis par le composant. Il peut être composé avec d'autres composants et être éventuellement paramétrable sans modifier son implémentation.

– *Les composants métiers*: Le concept de composant métier résulte de celui d'objet métier défini par l'OMG (Object Management Group) comme suit : *« Business Objects are representations of the nature and behaviour of real world things or concepts in terms that are meaningful to the enterprise. Customers, products, orders, employees, trades, financial instruments, shipping containers and vehicles are all examples of real-world concepts or things that could be represented by Business Objects »*. Ce que peut être traduit en: les objets métiers sont des représentations de la nature et des comportements des objets du monde réel ou des concepts en termes significatifs pour l'entreprise. Les clients, produits, ordres, employés, affaires, instruments financiers, conteneurs de transport et véhicules sont des exemples de concepts ou d'objets du monde réel représentables par les objets métiers. Un composant métier peut être vu comme un composant logiciel qui fourni des fonctions dans un domaine métier. Certains spécialistes définissent un composant métier comme une unité de réutilisation de connaissances de domaines d'un point de vue uniquement conceptuel.

2.6 Cycle de vie d'applications à base de composants

Une des particularités de la programmation par composants tient dans le fait que les étapes qui composent le cycle de vie d'une application à base de composants correspondent à celles qui décrivent habituellement le cycle de vie en génie logiciel. Cela signifie qu'on peut réutiliser un certain nombre de techniques ou de principes du génie logiciel pour la programmation par composants. Cependant, on peut distinguer deux cycles imbriqués dans le cas de la programmation par composants : un cycle spécifique au développement de composants et un cycle plus général qui définit des applications à base de composants.

2.6.1Cycle de vie général des applications à base de composants

Les étapes successives de ce cycle illustré dans la figure 4.3 sont analogues à celles qui sont définies en génie logiciel. Ces étapes sont :

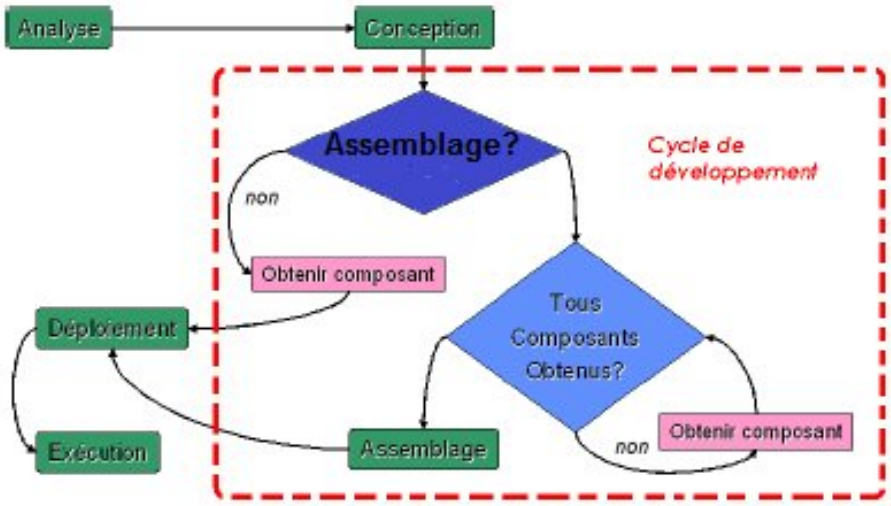


Figure 4.3 : *Cycle de vie d'une application à base de composants.*

- *Analyse:* Il faut analyser à la fois les besoins (fonctionnels et non fonctionnels) des composants (niveau composant) et les besoins d'interactions entre les composants (niveau application). Le résultat de l'analyse peut être représenté par un ensemble de diagrammes de type UML. Les spécifications de l'application sont obtenues par l'analyste à partir des informations que lui a fournies le client (i.e. l'utilisateur) de l'application.
- *Conception* : On fixe un modèle de composant (comme COM, CCM,…. (voir la section 2.7)) et on projette les spécifications des composants dans ce modèle. Cette étape est gérée par un concepteur.
- *Cycle de développement d'une application à base de composants* : Il s'agit de l'étape générale de développement qu'on retrouve en génie logiciel. En programmation par composants, la phase de développement des applications est décomposable en plusieurs étapes détaillées dans la section 2.6.2. Cette étape est gérée par un développeur.
- *Déploiement* : L'étape de déploiement signifie en général le chargement des composants sur une machine, leur initialisation (installation) et leur démarrage. Cette étape est gérée par un déployeur.
- *Exécution* : L'étape d'exécution signifie qu'un utilisateur administre l'application ainsi déployée et/ou emploie les services fournis par celle-ci. Cette étape est gérée par un ou plusieurs administrateurs ou usagers.
- *Maintenance* : si, arrivée en phase d'exécution et l'application doit être modifiée (phase de maintenance), alors il faut faire un bond en arrière dans le cycle de vie de l'application.

2.6.2 Cycle de développement d'une application à base de composants

Le développement d'une application à base de composants est fondé sur un processus cyclique visant à développer des composants, possiblement à partir de composants existants, qui peuvent être utilisés dans beaucoup d'applications. Par conséquent, le cycle de développement des applications à base de composants (figure 4.3) doit idéalement utiliser une ou plusieurs « bibliothèques de composants » où l'on peut *rechercher* des composants afin de les (ré) utiliser. De plus, chaque composant ainsi développé doit être *ajouté* à ces bibliothèques en vue d'une réutilisation ultérieure, éventuellement dans des projets différents. Le mécanisme d'obtention d'un composant est illustré sur la figure 4.4. Dans le cas d'un composite (composant composé d'autres composants) qui n'est pas présent dans la bibliothèque de composants, le mécanisme d'obtention d'un composant s'appelle récursivement pour essayer de récupérer des sous-composants.

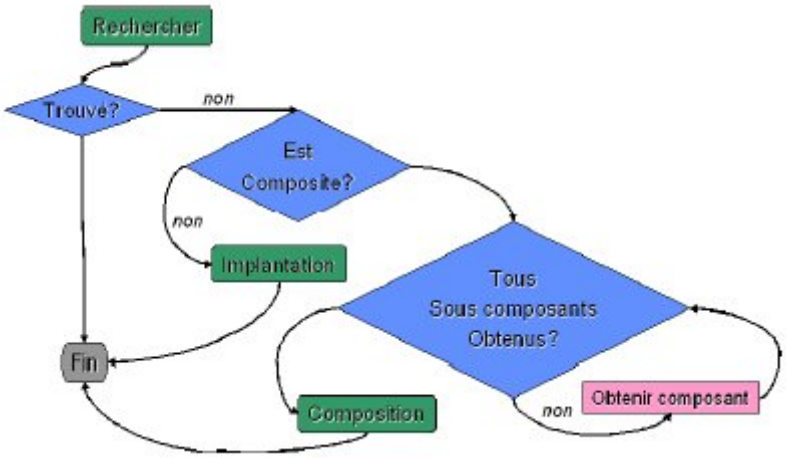


Figure 4.4 : *Mécanisme d'obtention d'un composant*

Ce cycle de développement décrit les étapes permettant de définir des composants primitifs (implantés directement), des composants composites (composants contenants plusieurs sous-composants primitifs ou composites) et des assemblages de composants (ensembles de composants interconnectés). Les étapes de ce cycle sont :

- *Implantation* : Cette étape marque le développement d'un composant dans un langage de programmation conformément au modèle de composants et aux spécifications de ce composant. Le composant peut être configuré à la fin de l'implantation, en fixant des valeurs aux propriétés configurables du composant.
- *Assemblage* : L'étape d'assemblage consiste à configurer individuellement un ensemble de composants et à les interconnecter en fonction du comportement global souhaité.
- *Composition* : La composition ressemble à l'étape d'assemblage, mais aussi à l'étape d'implantation puisqu'elle consiste à créer un composant contenant un

assemblage interne. Le processus de création d'un composant composite peut être défini comme suit : (1) Implantation d'un composite vide ; (2) Insertion des sous-composants (3) Assemblage des sous-composants ; (4) Importation/Exportation d'interfaces entre le composite et les sous-composants.

Une application développée à partir de composants existants peut aussi bien être représentée sous forme d'un assemblage de composants que sous la forme d'un seul composant (composite).

2 .7 Modèles et technologies de composants logiciels

Un modèle de composants regroupe un ensemble de conventions à respecter lors de la construction et l'utilisation des composants. Ces conventions permettent de définir et de gérer d'une manière uniforme les composants. Elles couvrent toutes les phases du cycle de vie d'un logiciel à base de composants : la conception, l'implantation, l'assemblage, le déploiement et l'exécution. Le concept de modèle de composant présente un appui essentiel pour le développement, la composition, la communication, le déploiement et l'évolution des composants. Un modèle de composant est un facteur important pour traiter les aspects de l'interopérabilité, l'évolutivité, la maintenabilité, ainsi que de nombreux autres attributs de qualité.

Contrairement aux ADLs (langages de description d'architecture) qui traitent principalement les activités de conception, les modèles et les technologies de composants académiques ou industriels comme COM de Microsoft et Java Beans de Sun mettent l'accent sur les dernières phases de développement à savoir l'implémentation, le déploiement et l'exécution. Il est tout à fait clair que les ADLs et les technologies de composants adressent des problèmes différents à des niveaux d'abstraction différents. Alors que les ADLs s'intéressent aux problèmes liés à la réalisation et le développement proprement dit, les modèles et les technologies de composants traitent les problèmes liés à la capacité d'analyser le fonctionnement des systèmes ainsi qu'aux supports d'exécution en mettant l'accent sur les aspects pragmatiques.

La majorité les modèles de composants existants offrent des possibilités d'ajouter de nouveaux services ou de nouvelles fonctionnalités aux systèmes logiciels d'une manière transparente. Cela est dû aux possibilités offertes par la majorité de modèles de composants entre autres le pouvoir de définir explicitement les composants et les connexions entre ces derniers ; le pouvoir de définir explicitement les implémentations de composants à partir de codes natifs ; et finalement définir explicitement les propriétés extra fonctionnelles des composants. La majorité des modèles de composants industriels visent certains objectifs communs comme l'augmentation de l'indépendance des codes ; augmenter la transparence des services ; la mise en œuvre de la distribution et la généralisation de service.

Il existe plusieurs modèles et technologies de composants logiciels comme COM, CCM, JAVA BEANS, .NET, FRACTAL et OSGI. Ces modèles de composants se diffèrent les uns des autres principalement dans les points suivants : les concepts clés de chaque modèle, la manière d'implémenter les composants, la manière dont sont réalisés les assemblages et finalement le cycle de vie du composant ainsi que celui de l'application à base de composants. Dans ce qui suit nous présentons en se basant sur ces points quelques modèles de composants logiciels ayant connu des succès importants et de larges utilisations, il s'agit de COM, CCM, JAVA BEANS et .NET.

2.7.1 Le modèle de composant COM

Les technologies COM (Component Object Model), DCOM, MTS, et COM+ proposées par Microsoft présentent de réelles tentatives pour augmenter l'indépendance autre les programmes d'un côté et pour vaincre l'hétérogénéité entre les langages de programmation d'un autre. La technologie COM se base sur les interfaces et sur des conventions d'interopérabilité, DCOM est une extension de COM qui traite la distribution, la technologie MTS présente une extension de DCOM qui offre des services de transaction ainsi qu'une certaine persistance, COM+ regroupe les trois technologies en un seul modèle. Dans la technologie COM une interface est vue comme une classe virtuelle c++ et prend la forme d'un ensemble de fonctions et de données sans codes associés. Un objet COM est un code binaire dont la source est écrite dans n'importe quel langage de programmation. Ce code binaire peut être sous la forme d'un exécutable ou d'une DLL (dynamic link Library) qui contient un minimum d'information nécessaire pour la liaison dynamique et l'identification de l'objet COM. La composition des COM est supportée par deux techniques, il s'agit de l'agrégation et la contenance qui signifie qu'un objet COM contient d'autres objets COM. Dans le cas de la contenance l'objet contenant déclare certaines interfaces des objets contenus, l'implémentation de ces dernières ce fait par délégation aux objets contenus. L'agrégation est un peu plus complexe, l'objet COM composé expose les interfaces des objets qui le composent comme si l'objet composé implémente réellement ces interfaces. Il est à noter que COM et COM+ sont des modèles de composant basés sur le moment d'exécution uniquement. C'est à dire, ces modèles ne proposent pas des supports pour le cycle de vie entier d'un composant ou d'une application à base de composants.

2.7.2 Le modèle CCM

Le modèle de composant industriel CCM (Corba Component Model) est développé par OMG. Dans CCM l'accent n'est pas mis uniquement sur l'assemblage de composants pour construire une application mais l'accent est mis également sur la conception et le déploiement des composants. CCM présente une solution pour le problème de complexité dans les services CORBA, cette complexité vient de la flexibilité offerte par CORBA aux développeurs, ce qui impose toujours un nombre important de choix. CCM est le résultat des travaux menés pour définir un modèle de composant dans lequel les standards d'EJB sont généralisés et le nombre de détails à spécifier est réduit ainsi que les risques d'inconsistance.

La vue externe d'un composant CCM est une extension du langage IDL de CORBA (langage de définition d'interfaces de CORBA) où chaque interface est constituée de cinq éléments : facettes, points de connexion décrivant l'aptitude d'utiliser des références fournies par des acteurs externes (*receptacles*), points de connexion émetteurs d'événements, points de connexion pour les quels quelques événements seront transférés, et des attributs pour la configuration du composant. Concernant l'assemblage des CCM, la connexion entre les composants est définie comme la référence d'objets, les composants sont connectés par la liaison des facettes aux *receptacles* et les points de connexion émetteurs d'événements aux points de connexion receveurs. Les connexions sont décrites explicitement dans un fichier XML appelé descripteur d'assemblage. Les connexions sont établies par le Framework CCM à l'initialisation. Dans CCM l'implémentation d'un composant consiste en un ensemble de segments présentant chacun un code exécutable implémentant au minimum un port. CCM propose un langage de définition d'implémentation de composant (CIDL) pour décrire les segments, les exécuteurs associés, les types de stockage et les classes de conteneurs. CCM donne de l'importance à la possibilité d'obtenir plusieurs services sans être obligé de faire des modifications au niveau du code tout comme MTS et EJB. Cette approche augmente la réutilisabilité et améliore la maintenabilité d'un côté, et affaiblit la complexité d'un autre côté. Les composants CCM utilisent des conteneurs pour implémenter les accès aux services en utilisant des patrons de conception collectionnés à travers des expériences dans la construction d'applications basées sur la technologie de l'objet.

2.7.3 Le modèle Java Beans

Le modèle Java Beans(1997) présente une première tentative d'intégrer la notion de composant dans le langage Java. Le modèle met l'accent sur la réutilisation et l'augmentation des capacités de composition statique et dynamique. L'objectif principal de Java Beans est de définir un modèle de composant pour java, ce modèle est utilisé pour créer des composants que l'on peut déplacer et composer ensemble dans une application. L'assemblage peut être réalisé visuellement avec un outil d'assemblage appelé *''builder Tools''*. Aucune solution spécifique pour l'assemblage propre à ce modèle n'est proposée, Java beans est conçu pour supporter plusieurs manières d'assemblage de composants. Un java bean est décrit à travers quatre types de port, *méthodes*, *propriétés*, *sources d'événements*, et *les ports d'écoute d'événements*. Ces ports présentent l'interface d'un composant java bean qui est implémentée généralement par un simple objet java, cette implémentation peut être vue comme une encapsulation de l'objet dans un composant. Parfois, des implémentations plus sophistiquées (collection d'objets ou envelopper un objet) sont nécessaires. Un autre volet bien couvert par java Beans concerne le paquetage des composants, ces derniers peuvent être paquetés dans des archives que l'on ouvert par un *'builder tool'* pour obtenir tous les composants archivés.

2.7.4 Le modèle .Net

Le modèle .Net de Microsoft ne présente pas une continuité des modèles COM, DCOM et COM+. Ce modèle est caractérisé par la limitation des interopérabilités binaires. .Net est basé sur un langage d'interopérabilité et d'introspection dénommé MSIL pour designer un langage interne de Microsoft. Ce langage est similaire au Byte code de java. MSIL est interprété par CLR (Common Language Runtime) similaire à la machine virtuelle java. A l'opposé de la vision adoptée par OMG où les informations relatives aux relations entre composants sont séparées de ces derniers, dans .Net les choses passent comme en langages de programmation. C'est-à-dire, les programmes contiennent les informations relatives aux relations entre composants. Un composant est décrit dans un descripteur (*manifest*) qui regroupe les informations : les méthodes importées et exportées, les événements, le code, les métadonnées, et les ressources. .Net est basé sur une liaison dynamique pour réaliser les connexions entre les composants pendant l'exécution. En ce qui concerne l'implémentation des composants, ces derniers consistent en modules qui sont des codes exécutables ou des DLLs (dynamic link libraries). La liste des modules qui composent le composant est donnée au compilateur pendant la compilation du module principale. A la fin de la compilation le *manifest* est généré dans le même fichier avec le module exécutable principal. Les modules ne peuvent pas être assemblés de modules à leurs tours et ils sont chargés qu'aux moments où ces derniers sont requis.

2.8 Points forts et limites de la programmation par composant

La programmation par composants possède plusieurs avantages. Elle favorise à la fois la modularité d'une application grâce à des composants ayant un comportement assez explicite et le plus découplé possible les uns des autres. La programmation par composants permet de pousser en avant la séparation des préoccupations (fonctionnelles et extra fonctionnelles). De plus, elle favorise la réutilisabilité des composants dans des contextes applicatifs différents. La réutilisation peut concerner les composants individuellement comme il peut s'agir d'une réutilisation d'assemblage entier de composants. La réutilisation a un impact direct sur le coût de développement et sur la qualité des logiciels. Elle augmente les possibilités de faire un développement plus rapide dans lequel les erreurs d'implémentation sont limitées. Le fait de réutiliser des composants plusieurs fois augmente la confiance dans ces derniers puisque ces derniers seront testés et retestés plusieurs fois ce que limite les risques d'erreurs causées par une mauvaise implantation. Les applications à base de composants ont l'avantage de pouvoir être réassemblées et adaptées pour différentes raisons par l'ajout, la suppression, le remplacement de composants ou par la reconfiguration d'assemblages de composant sans ajouter ou de retirer des composants. Enfin la programmation par composants est mise en œuvre grâce à un processus explicite et faisant intervenir des acteurs distincts, à la manière de ce qui est préconisé en génie logiciel.

Cependant, la programmation par composants doit faire face à un certain nombre de défis. Parmi les problèmes notoires des modèles de composants actuels, on peut souligner la

nécessité de pouvoir faire interagir des composants issus de modèles différents (compatibilité des modèles). De même, certains modèles mélangent des étapes du cycle de vie des applications à base de composants, ce qui oblige différents acteurs du cycle à en maîtriser plusieurs étapes simultanément (découplage des étapes du cycle). Enfin, beaucoup d'efforts doivent être fournis afin de faciliter la maintenance et l'évolution des applications (administration des applications). Cela passe par l'intégration de la « problématique de l'adaptation » au cœur même du cycle de vie des applications à base de composants. A cela s'ajoutent d'autres inconvénients résumés dans les points suivants :

- Les temps initiaux de développement des composants et les efforts nécessaires sont accrus. cet accroissement des coûts à court terme mènera néanmoins à leur diminution à plus long terme lorsqu'une politique de réutilisation aura réellement été mise en place au sein de l'entreprise de développement.
- Les exigences sont souvent peu claires et ambiguës. La gestion des exigences est déjà un défi en soit ; Ajouter à cela le fait que par nature un composant est destiné à être réutilisé dans diverses applications dont certaines sont encore inconnues.
- L'utilisation et la réutilisation sont des notions antagonistes. En effet, pour qu'un composant soit réutilisable, il faut qu'il soit assez général et adaptable, ce qui le rendra plus compliqué à utiliser.
- Les coûts de maintenance des composants sont plus élevés. Ceci est relié à la notion de gestion des exigences vue précédemment : comme un composant intervient dans divers projets/produits, après modification d'un composant, il faut vérifier sa conformité avec toutes les exigences.
- La fiabilité des applications est sensible aux changements. Ce point rejoint le deuxième vu précédemment, mais cette fois-ci, du côté de l'application. Tout changement au niveau de l'application, comme le changement d'autres composants, peut avoir un impact sur la fiabilité d'un composant par une méconnaissance de toutes les caractéristiques du composant.

3. Approche de développement orientée services

Semblablement aux approches de développement à base de composant regroupées dans le (*Component-based software engineering* CBSE), les approches de développement à base de service regroupées dans le (*Service-oriented software engineering* - SOSE) est un domaine établi du génie logiciel. Le CBSE s'inspire des autres disciplines de l'ingénierie telles que l'ingénierie mécanique ou électrique. Il repose sur l'idée de "construire des systèmes à partir de composants". Le SOSE se base sur le modèle de mondialisation des entreprises modernes. Le principe est de sous-traiter les fonctions à faible valeur ajoutée ou échappant au domaine de compétence de l'entreprise afin de se concentrer sur les aspects innovants du système. Le CBSE et SOSE partagent le principe de la réutilisation d'entités logicielles existantes, composants ou services. Tous deux reposent sur le concept

d'architecture logicielle dans lequel un système est vu comme une structure claire d'entités et de relations entre celles-ci. Le SOSE est un paradigme plus récent dont l'approche est influencée par le CBSE d'une façon similaire à l'incidence que l'objet a eu sur le composant.

Le SOSE a été proposé en réponse à une émergence de nouvelles applications qui sont très flexibles et innovantes. Ce type d'applications est le résultat d'une combinaison de plusieurs facteurs comme : l'explosion d'Internet, l'informatique ambiante et l'informatique mobile. Parallèlement à l'explosion d'Internet, Avec l'apparition des réseaux sans fil et la miniaturisation des composants électroniques, il a été possible de rendre mobile de nombreux appareils. Ces objets cohabitent aujourd'hui avec nous. Ainsi, nos téléphones portables aux multiples fonctions, les assistants personnels, les baladeurs numériques, les ordinateurs embarqués de voitures, les jouets de plus en plus perfectionnés peuvent être considérés comme des objets intelligents et communicants. Ces deux développements parallèles sont en train de se rejoindre. À travers ces objets communicants, Internet s'infuse dans la vie quotidienne. Il est possible d'accéder à Internet en utilisant par exemple un téléphone portable ou autres dispositifs personnels. Il est également possible que des appareils diffusent des informations provenant d'Internet ou à l'inverse de diffuser des informations du monde réel sur Internet. Cette intégration ouvre la voie à de nouvelles applications profitant de cet accès au monde réel pour fournir de nouveaux *services* via Internet.

Le développement d'applications innovantes exigeait de nouvelles propriétés difficiles à obtenir. L'intégration et la gestion des infrastructures d'exécution sont les éléments clés pour développer ces applications flexibles. Permettre l'intégration de manière souple et efficace d'applications avec d'autres ressources de l'entreprise ou d'autres entreprises est extrêmement complexe à réaliser. La gestion des infrastructures se concentre sur trois objectifs: le développement (et la réutilisation), l'automatisation de l'administration et la virtualisation de l'environnement. Le développement vise à aider les développeurs et les architectes à créer de nouvelles applications en réutilisant un maximum de briques déjà créées. L'automatisation des tâches d'administration vise à réduire la complexité de la gestion des applications ainsi que d'en améliorer la disponibilité, tout en réduisant les coûts. La virtualisation de l'environnement est la capacité à offrir une seule et même vue cohérente de toutes les ressources disponibles.

L'approche à service propose un nouveau moyen pour développer des applications flexibles. Cette approche utilise des services pour construire des applications plus rapidement et composables. Ainsi, cette approche propose des mécanismes permettant la mise en place d'applications plus facilement intégrables et gérables.

3.1 Approche à service

L'approche à service est un nouveau paradigme qui utilise les services comme la brique de base pour créer des applications. Le but de ce style architectural est d'utiliser des entités

(services) faiblement couplées afin d'améliorer la composabilité des applications. Un service consiste en une fonction ou fonctionnalité bien définie. C'est aussi un composant autonome qui ne dépend d'aucun contexte ou service externe. Il est divisé en opérations qui constituent autant d'actions spécifiques que le service peut réaliser.

3.1.1 Principes de l'approche à service

L'approche à service est un style architectural qui utilise les services comme entités de base. Les services interagissent et communiquent entre eux. Cette communication peut consister en un simple retour de données ou en une activité (coordination de plusieurs services).Le but principal de l'approche à service est de permettre la définition et l'utilisation de brique logicielle peu dépendante. En réduisant ces dépendances, chaque élément peut évoluer séparément. Ainsi les applications décrites en termes de services exhibent une plus grande flexibilité que les applications monolithiques. Ce faible couplage entre les entités de l'application provient du motif d'interaction proposé par l'approche à service et qui fait intervenir 3 acteurs (figure 4.5) :

- Les *fournisseurs* de services qui offrent des services;
- Les *consommateurs* de services qui requièrent et utilisent des services offerts par des fournisseurs ;
- Un *courtier* de service permettant aux fournisseurs de publier leurs services et aux consommateurs de découvrir et de sélectionner les services qu'ils veulent utiliser.

3.1.1.1 Interactions

Dans la figure 4.5, en plus des trois acteurs, nous remarquons l'élément *spécification de service*. C'est un élément central de l'approche à service qui contient la description de la fonctionnalité offerte par le service. Celle-ci peut être purement syntaxique ou contenir des éléments concernant le comportement ou la sémantique du service. La spécification de service est la seule information partagée par un fournisseur et un consommateur. Les fournisseurs de service implémentent une (ou plusieurs) spécification(s) de service. Les consommateurs de service savent comment interagir avec des fournisseurs implémentant la spécification de service qu'ils requièrent. Ainsi, l'approche à service propose trois primitives d'interaction comme il est illustré dans la figure 4.5 :

- La *publication*: les fournisseurs de service s'enregistrent auprès du courtier.
- La *découverte* : les consommateurs de service interrogent le courtier afin de trouver les fournisseurs de services qu'ils requièrent.
- La liaison : une fois découvert, le consommateur de service peut se lier à un fournisseur de service et utiliser le service qu'il fournit.

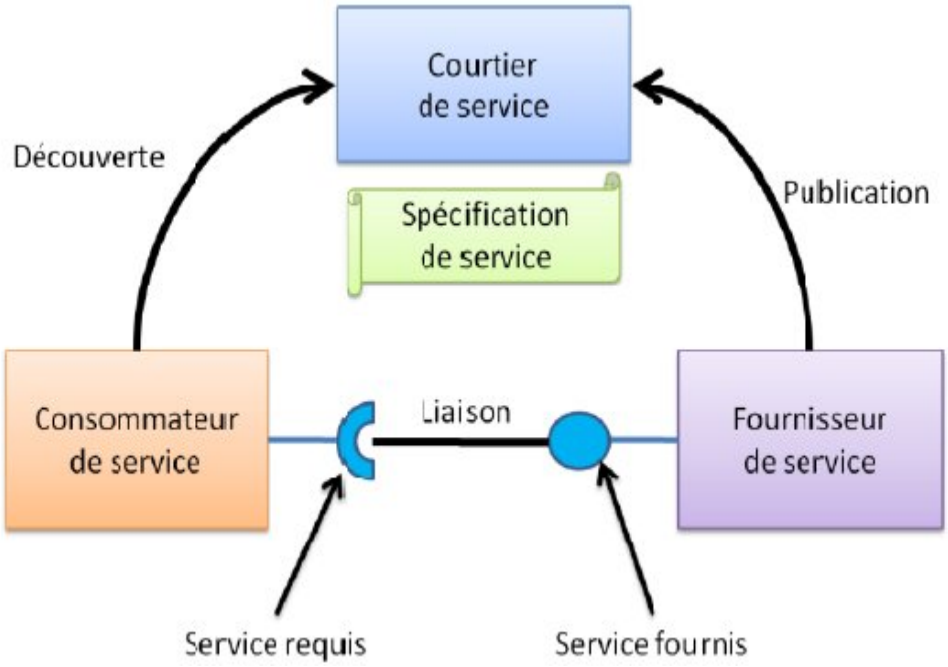


Figure 4.5 : *Acteurs et interactions dans l'architecture à service*

Ce principe d'interaction peut avoir lieu à n'importe quel moment dans le cycle de vie du logiciel; c'est-à-dire qu'il peut avoir lieu durant le développement afin de trouver un service adéquat (approche généralement utilisée avec les *services web* qui présentent un cas particulier de l'approche à services), mais également à l'exécution. Grâce à ce principe d'interaction, les applications à service ont des caractéristiques intéressantes telles que :

- Le faible couplage : comme la seule information partagée entre les fournisseurs et les consommateurs est la spécification de service, le couplage entre ces deux entités est faible.
- La liaison tardive : la liaison entre les fournisseurs et les consommateurs a lieu seulement lorsqu'un fournisseur est trouvé et lorsque le consommateur le demande.
- Le masquage de l'hétérogénéité : grâce au faible couplage, un consommateur n'a pas à connaitre les détails d'implémentation du fournisseur de service, ni à sa localisation précise.

3.1.1.2 Compositions de services

Afin de concevoir des applications à service plus complexe, il est nécessaire de composer des services ensemble afin de fournir des services de plus haut niveau, c'est-à-dire qu'un fournisseur peut requérir autres services afin de fournir le sien. Il existe deux tendances dans la composition de services, comportementale et structurelle. La composition comportementale tire ses sources des procédés logiciels. Une composition comportementale de services combine des services en décrivant le procédé logique permettant de fournir un service de plus niveau. La composition de services web utilise majoritairement ce style de

composition et plus exactement le langage *Business Process Execution Language for Web Services*. Cette composition est ensuite exécutée par un moteur d'orchestration qui gérera l'exécution des activités décrites dans la composition. Le deuxième style de composition est la composition structurelle. Ce style de composition se rapproche des langages de description d'architecture. Une composition structurelle de service décrit l'architecture d'un composant qui fournit un service. Cette architecture montre les composants participants et leurs interconnexions. Ces deux styles de compositions peuvent être combinés : une composition comportementale peut être l'implémentation d'un service utilisé dans une composition structurelle.

3.2 Architectures orientées service

Un concept important du SOSE est l'architecture à service (Service Oriented Architecture ou SOA). Un SOA est un ensemble de technologies permettant de mettre en place des applications suivant le paradigme de l'approche à service. Généralement, un SOA est un intergiciel permettant aux applications de fournir, publier, découvrir et utiliser des services. Un SOA implante les primitives de l'approche à service en utilisant diverses technologies. Par exemple, les services web sont un SOA, alors qu'OSGi en est un autre. Ces deux SOA utilisent des technologies très différentes. Les choix des technologies pour mettre en place un SOA dépendent du domaine métier et des objectifs poursuivis.

Un SOA pur n'adresse pas les problématiques de composition et de gestion nécessaire à la création d'applications flexibles. Un SOA *étendu* prend en compte ces préoccupations (Figure 4.6). Tout d'abord un SOA étendu se base sur un SOA basique. Ensuite, un SOA étendu propose des fonctionnalités pour le support de la composition. Parmi ces fonctionnalités, il doit être possible de coordonner des services, de gérer les compositions ainsi que d'assurer la conformité de la composition par rapport au service rendu. Une fois réalisé, un service composite peut ensuite être utilisé comme un service « normal ». Au dessus de la couche de composition, un SOA étendu fourni des fonctionnalités permettant la gestion des applications à service. Cette couche fournit les mécanismes de management des applications tel que le déploiement, la supervision, l'évolution, … . Deux fonctionnalités sont particulièrement cruciales : la capacité à vérifier la conformité des compositions de services, ainsi que la capacité à vérifier différents indices de qualité devant être atteints. De manière transversale, un SOA étendu doit permettre la vérification et la gestion des propriétés non fonctionnelles telles que la sécurité ainsi que de diverses formes de qualité de service.

Aujourd'hui, il existe de nombreux SOA ainsi que de nombreux SOA étendus. Il devient critique de pouvoir les faire communiquer. En effet, bien qu'à l'intérieur d'un SOA, les entités peuvent communiquer dans le langage supporté par ce SOA, des entités de deux SOA différents ne peuvent souvent pas communiquer.

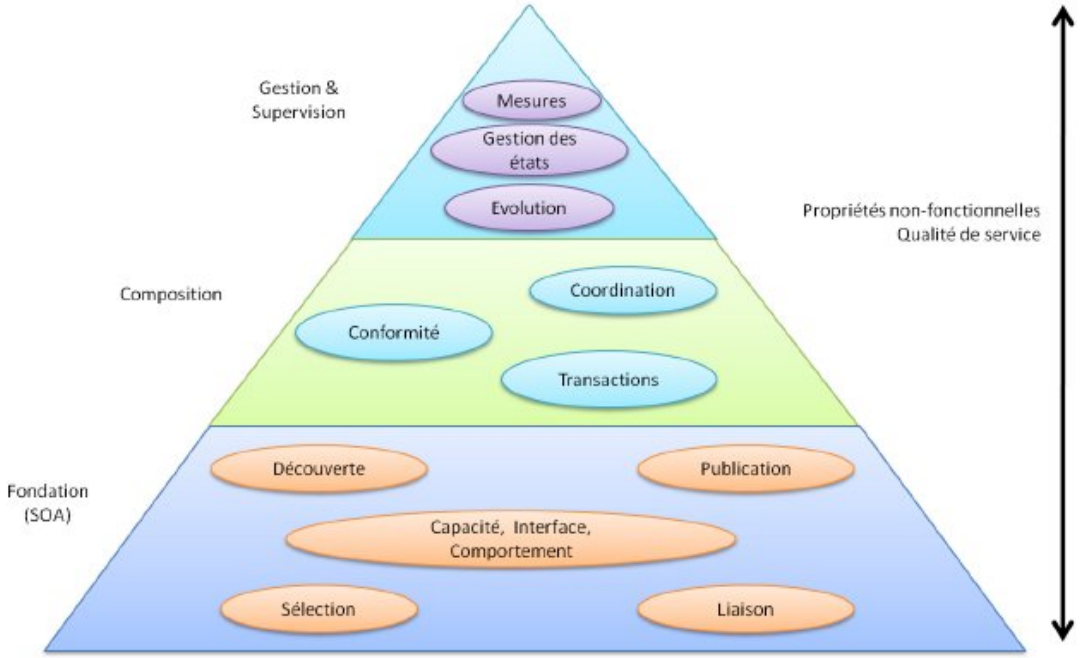


Figure 4.6 : *Fonctionnalités d'un SOA étendu*

3.3 Différences entre CBSE et SOSE

Le développement par composant et celui basé service ont une approche très similaire et partagent les activités d'identification des entités logicielles (composant ou service) qui répondent aux besoins, puis de combinaison de ces entités pour réaliser l'application globale. Ils reposent sur les mêmes notions de composition et de composite qui garantissent une approche homogène où toute entité est un composant ou un service. Cependant, bien que le développement par composant et le développement par service aient le même objectif global, les théories derrière les composants et les services sont différentes.

Un composant est dit *off-the-shelf* (sur l'étagère) par adoption d'une pièce technologique, le composant, qui est disponible pour les développeurs. Un service est centré sur l'utilisation d'une fonction fournie par un tiers. Pour illustrer cette première distinction prenons un exemple de l'industrie des jeux vidéo sur PC. Cette industrie repose sur deux modèles de distribution de contenus. Le premier modèle classique correspond à un joueur qui achète une copie de son jeu. Le joueur est ensuite responsable du déploiement de cette copie sur sa propre machine. Ce modèle correspond à une approche composant. Typiquement, le jeu (composant) vient avec un manuel d'instructions (documentation) sur la configuration système minimale requise pour être capable d'exécuter ce jeu ; puis sur les opérations nécessaires pour l'installer ; et enfin sur comment y jouer. Tous ces éléments définissent des contraintes du côté client. Le second modèle appelé Cloud gaming illustre la notion de service. Le joueur paie le droit de jouer à un jeu qui est exécuté sur une plate-forme à distance sous la responsabilité du fournisseur. Il a uniquement besoin d'une interface et d'une connexion pour accéder à cette plate-forme. Il n'est plus responsable du déploiement du jeu sur sa machine. Les seules informations qui lui sont nécessaires sont : comment

accéder à cette plate-forme et comment jouer au jeu. Ainsi, le joueur n'a plus à s'occuper des contraintes du système pour exécuter le jeu qui conserve une qualité constante. Au contraire du premier modèle où il peut être amené à améliorer la configuration de sa machine pour obtenir une certaine expérience du jeu qui va varier d'un système à un autre.

Ainsi, l'orienté service pousse la responsabilité du propriétaire à son extrême. En effet, l'approche off-the-shelf de CBSE implique que seul le développement, la qualité de service et la maintenance sont sous la responsabilité du fournisseur, tandis qu'en SOSE le fournisseur est aussi responsable du déploiement, de l'exécution et la gestion du service. La localisation physique du service et ses évolutions sont transparentes pour l'utilisateur.

Ce concept de responsabilité implique aussi la nature multi-tenant du service, c'est- à-dire que le service exécuté à distance doit être capable de gérer de multiples connexions parallèles. Ce principe n'est pas nécessaire à un composant. En effet, bien que pouvant appartenir à de multiples compositions, au niveau de l'exécution, différentes instances du composant sont créées et chacune d'elles sous la responsabilité d'un client.

Ainsi, CBSE et SOSE ont des points de vue différents sur les relations entre client et fournisseur qui agissent sur d'autres aspects de distinction : la gestion des hétérogénéités, l'importance de l'automatisation et le couplage.

L'objectif du service est l'indépendance avec les technologies d'implémentation. Un service doit être accessible et utilisable sans aucune hypothèse sur son implémentation, sur les utilisateurs potentiels ou sur la façon d'utiliser ce service. Cette problématique est bien connue en CBSE mais ne représente pas un enjeu aussi critique. En effet, il existe un très grand nombre de modèles de composant. Un architecte doit choisir un modèle particulier et n'utiliser que des composants respectant ce modèle car la collaboration entre modèles différents est très difficile. Le SOSE prône un unique modèle homogène de services qui doit être standardisé et utilisé par tous.

Une autre distinction consiste en l'importance accordée à l'automatisation qui a participé à la définition même du SOSE. En effet, la grande majorité des travaux cherche à automatiser les mécanismes du SOSE tels que la publication de services, les découvertes, les sélections, la composition, etc. Ce principe d'automatisation est poussé à son extrême par le concept d'auto-adaptation qui cherche à coordonner l'ensemble des mécanismes liés au service afin d'assurer des adaptations contextuelles. Bien que l'automatisation soit un élément clé de la recherche en CBSE elle ne fait pas partie de la définition d'un modèle de composant.

La gestion des hétérogénéités et l'automatisation participent à une autre distinction entre CBSE et SOSE : Le SOSE prône un couplage faible à tous les niveaux, du développement à l'exécution. D'une part, un architecte ne doit pas connaître, au moment de la spécification de son application, les services qui seront réellement utilisés. Ce couplage faible entre l'expression des besoins et les services concrets disponibles est supporté par les

principes de découvertes et sélections dynamiques. D'autre part, le SOSE cherche à réduire au maximum les dépendances entre services concrets en collaboration pour faciliter l'auto-adaptation.

Toutes les distinctions que nous avons identifiées sont liées à la notion de dynamicité. Le SOSE vient de besoins fonctionnels d'applications spécifiques en termes d'exigences sur l'adaptabilité et l'agilité alors que le CBSE a une vocation plus générale.

4. Développement par agents et systèmes multi agents

Le paradigme agent est un paradigme de programmation qui partage avec les paradigmes composants et celui à base de services, la propriété de proposer des abstractions pour la structuration du logiciel sous la forme d'une combinaison d'un ensemble éléments logiciels (des agents dans le cas de ce paradigme). Ce paradigme pousse très loin par rapport aux autres paradigmes à la fois l'abstraction et la flexibilité du couplage. La technologie des agents fait partie du domaine de l'intelligence artificielle distribuée dans laquelle l'intelligence et l'expertise sont distribuées sur un ensemble d'entités logicielles ou physiques autonomes, en interactions, qui travaillent en collaboration pour résoudre un problème ou attendre certains objectifs. Cette technologie a tiré profits d'autres disciplines comme la sociologie, la psychologie sociale et les sciences cognitives. Aujourd'hui la technologie d'agent est largement utilisée et les agents sont impliqués dans plusieurs applications informatiques et plus particulièrement celles caractérisées par la recherche d'atteindre certains buts, puisqu'un agent est une entité qui agit à la demande de quelqu'un pour accomplir quelque chose.

4.1 Définitions d'un agent

Le concept d'agent comme plusieurs autres concepts est définit selon plusieurs manières quoique toutes ces définitions de l'agent ont quelques points en commun. La communauté scientifique a accepté des définitions parmi lesquelles nous citons la définition de J Ferber considérée l'une des premières définitions où « *Un agent est une entité autonome, réelle ou abstraite, qui est capable d'agir sur elle-même et sur son environnement, qui, dans un univers multi-agents, peut communiquer avec d'autres agents, et dont le comportement est la conséquence de ses observations, de ses connaissances et des interactions avec les autres agents* » .

Une autre définition de l'agent proposée par Jennings est la suivante : *« Un agent est un système informatique qui est situé dans un certain environnement et qui est capable d'effectuer de manière autonome une action afin de répondre aux objectifs pour lesquels il a été conçu ».*

En synthétisant toutes les définitions proposées pour le concept d'agent, il en résulte qu'un agent est une entité physique ou virtuelle ayant les qualités suivantes: l'agent est capable d'agir dans un environnement ; il peut communiquer directement avec d'autres agents ; son comportement est contraint par des objectifs individuels ; Il possède des ressources propres ; Il est capable de percevoir son environnement et dispose d'une

représentation partielle de cet environnement ; l'agent possède des compétences et offre des services ; Il a la capacité de se reproduire ; le comportement d'un agent tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont il dispose, et en fonction de sa perception, de ses représentations de l'environnement ainsi que les communications qu'il effectue avec les autres entités.

4.2 Caractéristiques d'un agent

Les caractéristiques principales qu'un agent possède et qui sont pratiquement objet de consensus par toute la communauté agent sont :

- ***Situation*** **:** l'agent est une entité située, capable d'agir sur son environnement à partir des entrées sensorielles qu'il reçoit de ce même environnement ;
- ***Autonomie*** **:** l'agent est capable d'agir sans l'intervention d'un tiers (humain ou agent) et contrôle ses propres actions ainsi que son état interne ;
- ***Apprentissage*** **:** un agent est capable d'apprendre et d'évoluer. En fonction de cet apprentissage, il est aussi capable de changer le comportement en fonction des expériences passées ;
- ***Mobilité*** **:** la capacité d'un agent de se déplacer à travers un réseau d'une machine à une autre ;
- ***Flexibilité*** **:** cette caractéristique résume les propriétés suivantes :
  - ***Réactivité*** **:** un agent est capable de modifier son comportement lorsque les conditions environnementales changent. Il est capable aussi de percevoir son environnement et d'élaborer une réponse dans les temps requis ;
  - ***Pro-activité*** **:** les agents n'agissent pas seulement en réponse à leur environnement mais ils sont également capables d'avoir un comportement guidé par un but avec la possibilité de prendre l'initiative ;
  - ***Sociabilité*** **:** l'agent doit être capable d'interagir avec les autres agents (logiciels ou humains) et peut se trouver engagé dans des transactions sociales ;
  - ***Activité*** **:** un agent est toujours actif ; il s'exécute dans un thread ou un processus indépendant ;
  - ***Communication*** **:** c'est la possibilité d'échange de messages entre agents, selon l'un des schémas de communication : ***Now-type messaging*** **:** dans lequel l'émetteur du message bloque son exécution jusqu'à ce que le récepteur ait téléchargé le message et envoyé sa réponse ; ***Future-type messaging*** où l'émetteur ne bloque pas son exécution mais l'expéditeur retient une variable qui peut être utilisée pour obtenir le résultat et finalement ; le schéma ***One-way messaging*** qui présente un type de messagerie asynchrone qui ne bloque pas l'exécution courante. L'expéditeur ne retient pas une variable pour ce message et le récepteur ne va jamais répondre. Ce type est utile quand deux

agents engagent une conversation où l'agent expéditeur n'a pas besoin de la réponse de l'agent récepteur.

4.3 Typologie des agents

Principalement les agents peuvent être répertoriés en deux types reflétant chacun une école de pensée dans la communauté des agents: *l'école cognitive* et *l'école réactive.* Avec l'existence d'autres types d'agents, qualifiés d'hybrides, et qui utilisent ces deux types de comportements. Les agents réactifs sont de plus bas niveau, ils ne disposent que d'un protocole et d'un langage de communication réduits, ils n'ont pas une représentation globale de leurs environnement et ne sont pas capables de tenir compte de leur actions passées, leurs capacités répondent uniquement à la loi *stimulus/ action.* Quant aux agents cognitifs, ces derniers possèdent une représentation explicite de leur environnement et des autres agents ; Ils tiennent compte de leurs passés et disposent d'un but explicite; ils possèdent une base de connaissances qui contient toutes les connaissances concernant leurs fonctionnements et leurs environnements. Les agents cognitifs sont dotés de capacités de raisonnement sur leurs bases de connaissances et ils peuvent avoir un monde social d'organisation. Les agents cognitifs se trouvent en petit nombre, ainsi la granularité dans les systèmes composés de ces derniers est faible. Les agents BDI (**B**elief, **D**esire, **I**ntension) sont un bon représentant de ce type d'agents qui peuvent être *des agents intelligents*, *des agents collaborant*, *des agents interfaces* …etc.

Il en résulte de la combinaison des deux manières de pensée des architectures d'agents composées d'un ensemble de modules organisés dans une hiérarchie, chaque module étant soit une composante cognitive, soit une composante réactive. De cette manière, le comportement proactif de l'agent, dirigé par les buts, est combiné avec un comportement réactif afin d'obtenir les avantages des architectures cognitives et réactives, tout en éliminant ou réduisant leurs limitations. Les architectures hybrides souvent formées en couches dont trois suffisent généralement, la couche de plus bas niveau est une couche réactive qui prend des décisions en se basant sur des données brutes en provenance des senseurs, la couche intermédiaire qui fait abstraction des données brutes et fonctionne selon une vision qui se situe au niveau des connaissances de l'environnement et enfin, la couche supérieure qui se charge des aspects sociaux de l'environnement en tenant compte des autres agents.

4.4 Systèmes Multi Agents (SMAs)

La mise en œuvre d'un ensemble d'agents mettant en commun leurs compétences et connaissances s'avère plus intéressante que la manipulation d'agents en tant qu'entités individuelles formant ainsi ce qui est appelé un système multi agents. Un *système multi-agents* est un système distribué composé d'un ensemble d'agents. Contrairement aux systèmes d'IA, qui simulent dans une certaine mesure les capacités du raisonnement humain, les SMA sont conçus et implantés idéalement comme un ensemble d'agents interagissant, le plus souvent, selon des modes de *coopération*, de *concurrence* ou de *coexistence.*

Les systèmes multi agents sont généralement caractérisés par : la possession limitée des informations ou des capacités de résolution de problèmes de chaque agent ; ainsi que chaque agent a un point de vue partiel; il n'y a aucun contrôle global du système multi-agents; les donnés sont décentralisées; et finalement le calcul est asynchrone.

Les agents autonomes et les systèmes multi-agents représentent une approche pour l'analyse, la conception et l'implantation des systèmes informatiques complexes. La vision basée sur l'entité agent offre un puissant répertoire d'outils, de techniques, et de métaphores qui y ont le potentiel d'améliorer considérablement les systèmes logiciels. Les systèmes multi-agents sont présentés comme solution privilégiée pour analyser concevoir et construire les systèmes logiciels complexes puisque en partitionnant un système en N agents réduit sa complexité et la configuration résultante se trouve plus facile à développer, à tester et à maintenir. Les SMAs s'adaptent bien à la réalité dans la mesure où de nombreux problèmes sont de nature distribuée. L'utilisation de nombreux agents résous le problème de façons différentes et produit généralement des solutions de meilleures qualités en termes de compétence, complétude et de précision.

4.4.1 Types de systèmes multi agents

Les SMAs peuvent être catégorisés selon : le nombre d'agents, la nature et la complexité de ces agents, en :

- ***SMAs ouverts*** : les agents y entrent et en sortent librement.
- ***SMAs fermés*** : où l'ensemble d'agents restent le même.
- ***SMA homogène*** : dont tous les agents sont construits sur le même modèle.
- ***SMA hétérogène*** : dont les agents sont construits sur des modèles différents.

4.4.2 Interaction et Coopération entre agents

Une interaction est une mise en relation dynamique de deux ou plusieurs agents par le biais d'un ensemble d'actions réciproques. La coopération est la forme générale de l'interaction. Plusieurs études en IAD se sont attachées à la définir, mais l'ensemble de ces études est plutôt hétérogène et il règne une certaine confusion sur la signification et la mise en œuvre de la coopération. La coopération fait référence à plusieurs éléments : buts, stratégies, comportements, formes,…etc. la définition de la coopération est généralement subordonnée aux concepts de coordination, d'organisation, de communication, et de négociation.

En sociologie, les sociologues, considèrent que les être humains coopèrent à cause de leurs capacités limitées. Nous coopérons puisque nous ne pouvons pas accomplir une tâche individuellement ou puisque nous ne la réalisons pas rapidement et efficacement. La psychologie quant à elle, utilise deux conditions pour définir la coopération : « Les sujets

doivent avoir le même objectif et chaque sujet doit voir le produit final comme la somme, la composition ou la combinaison des produits partiels de son ou de ses actions et de celles des autres sujets ». La coopération est souvent traitée sous l'angle de l'intérêt qu'il y a ou non à coopérer avec autrui ou sur la manière de communiquer dans les groupes. Contrairement aux autres disciplines où l'accent est mis sur les structures du collectif et sur les relations affectives qui amènent les acteurs à coopérer et à travailler ensemble, l'approche IAD-SMA reformule le problème en mettant en avant les caractéristiques cognitives et comportementales nécessaires à la mise en œuvre d'un travail collectif ou de la coopération dont les objectifs sont généralement :

- Augmenter la vitesse de résolution des tâches par leur parallélisation ;
- Augmenter le nombre ou la portée des tâches réalisables par le partage de ressources ;
- Augmenter la probabilité d'achever des tâches par leur duplication et si possible par l'utilisation de méthodes différentes pour les réaliser ;
- Diminuer l'interférence entre tâches en évitant les interactions négatives.

4.4.3 Communication

La *communication* désigne l'ensemble des processus par lesquels s'effectue l'opération de mise en relation d'un émetteur avec un ou plusieurs récepteurs, dans l'intention d'atteindre certains objectifs. Elle est considérée comme une forme d'action particulière qui, au lieu de s'appliquer à la transformation de l'environnement, agit sur (modifie) les représentations mentales des agents (buts, croyances, etc..). D'où, la communication dans l'univers multi-agents n'est plus une simple tâche d'entrée-sortie, mais doit être modélisée comme un acte pouvant influer sur l'état des autres agents.

L'ensemble des processus matérialisant une communication regroupe des processus physiques qui désignent les mécanismes d'exécution des actions (l'envoi et la réception de messages), et des processus psychologiques se rapportent aux transformations opérées par les communications sur les buts et les croyances des agents.

La communication est un moyen ou une méthode de coopération (d'interaction), à côté de : la *collaboration* qui s'intéresse à la manière de répartir le travail et les ressources entre plusieurs agents ; la *coordination* d'actions qui analyse la manière dont les actions des différents agents doivent être organisées dans le temps et l'espace de manière à atteindre les objectifs ; la résolution de conflit par arbitrage et *négociation* en établissant par exemple des compromis, …, etc.

4.4.3.1 Pourquoi les agents doivent communiquer ?

Les agents communiquent et interagissent pour synchroniser leurs actions et pour résoudre des conflits, qui sont des conflits de ressources, de buts ou d'intérêts. Ils communiquent également pour s'aider mutuellement ou pour suppléer aux limites de leurs

champs de perception. En effet, un agent ne peut être en relation avec tous les autres, ni équipé de tous les capteurs nécessaires à la connaissance de l'environnement.

4.4.3.2 Quand et avec qui les agents communiquent ?

Pour répondre à cette question, il faut identifier les situations qui vont nécessiter la communication des agents. En général, les agents communiquent lorsqu'ils sont face à un problème qu'ils ne savent pas résoudre (soit par manque de compétences ou de ressources), lorsqu'il est nécessaire de coordonner leurs actions, ou encore lorsqu'il y a un conflit entre plusieurs agents et que le conflit ne peut pas être résolu de façon déterministe. Les communications peuvent être diffusées à l'ensemble des agents ou à des agents particuliers (des agents susceptibles d'être intéressés par le message).

4.4.3.3 Comment communiquer ?

Les procédures de communication pour véhiculer les messages (qui sont porteurs d'informations ou d'actions) entre agents sont principalement la communication par envoi de messages et la communication par partage d'informations.

4.4.4 Organisation des systèmes multi-agents

Les systèmes multi-agents sont composés d'un ensemble d'agents autonomes en interaction permanente, leurs comportements sont des émergences des comportements des agents qui les composent. Pour contraindre les agents à se comporter de sorte à satisfaire les objectifs globaux d'un système multi agents, les organisations présentent un bon moyen. Les recherches dans les SMA et les inspirations des organisations sociales dans les communautés d'individus dans les contextes sociologiques tout en respectant les contraintes d'opérationnalisation et de mise en œuvre ont permet d'avoir plusieurs modèles organisationnels pour les SMAs. L'étude de l'organisation opérationnalisable est à la fois l'étude du modèle décrivant les échanges potentiels entre agents, et également le schéma explicatif applicable sur le système.

L'organisation d'un SMA est un modèle permettant aux agents de coordonner leurs actions pour accomplir une ou plusieurs tâches, elle définit d'une part une structure exprimée en un ensemble de rôles à jouer par les agents ainsi que les communications entre rôles c'est à dire entre les agents jouant tel ou tel rôle. D'autre part l'organisation définit un aspect fonctionnel du SMA en termes de processus de coordination qui déterminent l'allocation des tâches aux agents ainsi que leurs décompositions en sous tâchent.

La situation de la connaissance organisationnelle présente un point crucial dans la spécification d'une organisation ; l'organisation peut exister que dans la connaissance des agents, nous parlons alors du *point de vue centré agent*, où la spécification de l'organisation revient à la spécification des agents eux-mêmes. Le *point de vue centré organisation* considère l'organisation comme une entité manipulable entant que telle, la spécification dans ce cas est indépendante des agents et de leurs architectures ou leurs choix d'implémentations. Dans ce

type, plusieurs modèles organisationnels – l'organisation est l'implémentation d'un modèle organisationnel - sont nombreux et on peut citer juste à titre d'exemples les modèles : Moïse, Moïse+, MOCA, AGR, AGRS,…etc. Ces modèles organisationnels explicites sont répertoriés en des modèles qui s'appuient sur les *plans globaux* et ceux qui se focalisent sur les *rôles* comme Moise, AGR et AGRS. .

4.4.5 La réorganisation des systèmes multi-agents

Les systèmes multi-agents s'appliquent particulièrement aux domaines ayant de très fortes dynamiques, où, il devient très difficile (voire impossible) de prévoir toutes les situations possibles dès la conception. Une situation imprévue peut donc remettre en cause tous le système. Il est donc nécessaire de trouver les moyens pour que les agents parviennent à gérer la dynamique de l'environnement de manière autonome. D'autre part, est-il possible d'établir une organisation idéale pour un problème ou un environnement donné ? C'est peu probable : l'environnement, le contexte des agents n'est jamais complètement maîtrisé, et l'adaptation du système est obligatoire.

La réorganisation est l'un des moyens pour qu'un SMA s'adapte aux changements de l'environnement, elle consiste à passer d'une organisation à une autre. Dans le cas où l'organisation s'avère n'est pas la plus appropriée le système procède à une réorganisation et passe à une nouvelle organisation plus adéquate qui améliorera la situation. Ce passage est réalisé par le SMA d'une manière autonome sans intervention externe (auto-organisation). La réorganisation peut être statique est spécifiée par le concepteur qui peut pour des situations spécifiques où l'organisation de départ n'est pas la plus appropriée spécifier d'autres organisations. Une fois ces situations détectées, le système se réorganise selon la nouvelle organisation définie à priori par le concepteur. La réorganisation peut être dynamique, les agents eux-mêmes décident quand et comment se réorganiser, aucune spécification n'est définie à priori, l'avantage de cette dernière par rapport à la réorganisation statique réside dans le fait qu'il est impossible de déterminer toutes les situations possibles et leur vérifier l'adéquaté de l'organisation de départ et en spécifier de nouvelles par la suite.

4.4.6 Méthodes de développement de systèmes multi-agents

Les systèmes multi-agents (SMA) ont montré leur pertinence pour la conception d'applications distribuées (logiquement ou physiquement), complexes et robustes. Le concept d'agent est aujourd'hui plus qu'une technologie efficace, il représente un nouveau paradigme pour le développement de logiciels dans lesquels l'agent est un logiciel autonome qui possède un objectif, évolue dans un environnement dynamique et interagit avec d'autres agents au moyen de langages et de protocoles. Souvent, l'agent est considéré comme un objet « intelligent » ou comme un niveau d'abstraction au-dessus des objets et des composants. Les méthodes de développement orientées objet – au vu des différences entre les objets et les agents – ne sont pas directement applicables au développement de SMA. Il est alors devenu nécessaire d'étendre ou de développer de nouveaux modèles, de nouvelles méthodes et de nouveaux outils adaptés au développement de systèmes multi-agents.

Le paradigme multi-agent permet, d'une part, de faciliter la compréhension et la modélisation des systèmes complexes. Le paradigme agent est un nouveau niveau d'abstraction qui permet d'exprimer une application en termes d'agents autonomes qui jouent des rôles et rendent des services dans une organisation. La conception multi-agent requiert donc des méthodes associées. Les premières méthodes ont vu le jour dans les années 1995 mais, c'est depuis les années 2000 que les recherches dans ce domaine ont été très nombreuses. On distingue trois grandes familles de méthodes : les plus nombreuses issues de l'ingénierie du logiciel et qui s'inspirent des méthodes orientées objet, celles qui sont issues de l'ingénierie des connaissances et les approches issues de plates-formes.

La plupart des méthodes de développement de systèmes multi-agents s'inspirent de l'ingénierie orientée objet, et il est donc naturel que les cycles de vie et processus utilisés dans la conception de tels systèmes soient de types classiquement utilisés en conception modulaire : cascades, itératifs, en V, et en spirales. De plus, les méthodes multi-agents intègrent, pour la grande majorité, les grandes phases de développement que sont l'analyse des besoins, la conception de l'architecture (ou analyse), la conception détaillée, le développement (ou implémentation) et le déploiement, même si elles ne sont pas clairement identifiées en tant que telles. Par exemple, les méthodes ADELFE et INGENIAS reposent sur le processus unifié (Unified Process, UP) et intègrent à ce processus de nouvelles étapes et définitions de travaux spécifiques au domaine multi-agent, alors que les méthodes PASSI et ASPECS reposent sur un processus itérable original. Certaines méthodes ne couvrent qu'une partie du processus, comme GAIA ou SODA qui se concentrent exclusivement sur l'analyse des besoins et la conception, ou Prometheus et MaSE qui n'abordent pas l'analyse des besoins. Généralement, les méthodes ne proposent pas de processus allant jusqu'au déploiement, s'appuyant sur le principe que les modèles multi-agents sont plus du niveau de l'analyse et de la conception, exception faite de PASSI ou INGENIAS qui couvrent tout le cycle de développement.

Quelles que soient les phases couvertes par ces méthodes, l'ingénieur y exprime ses réflexions en utilisant les concepts définis dans le langage de modélisation associé. Ainsi, pour des systèmes multi-agents, est-il important de définir la nature des agents et du système associés à la méthode.

4.4.6.1 Analyse des besoins

Au cours d'un processus de développement, la phase *d'analyse des besoins*, ou spécification fonctionnelle, doit exprimer les fonctionnalités du système à concevoir du point de vue de l'utilisateur. Il s'agit dans un premier temps d'établir un cahier des charges consensuel entre clients, utilisateurs et concepteurs sur ce que le système doit faire, ses limites et ses contraintes. Ces besoins, fonctionnels ou non, sont ensuite organisés et gérés de manière à définir plus précisément le système et son environnement. L'analyse exprime les besoins des utilisateurs en termes liés au domaine et à ce que le système doit faire. Dans

le cas des systèmes multi-agents, lorsque la tâche du système est clairement identifiée, cette phase doit mener à identifier les agents qui y interviendront ainsi que leurs interactions.

4.4.6.2 Conception

Après une phase d'analyse, pratiquement c'est la phase de *conception* qui suit dans un processus de développement. La conception s'attache à décrire le fonctionnement des agents identifiés lors de l'analyse (par le choix d'une architecture d'agent spécifique), l'architecture générale du système multi-agent (organisation) et les différents concepts spécifiques à la méthode utilisée. Ceci se traduit en règle générale, par la définition des classes d'agents et de leurs propriétés. Dans la méthode GAIA, par exemple, la conception architecturale aboutit au raffinage des modèles de rôle et d'interaction par l'analyse des structures organisationnelles. Lors de la conception détaillée, les modèles d'agent (détermination des types et instances d'agents) et de services sont spécifiés. Dans la méthode INGENIAS, la conception correspond à la définition de la société d'agents composant le système multi-agent, en décrivant les rôles et les protocoles de communication entre agents. Les principaux modèles manipulés lors de la conception multi-agent sont les mêmes que dans une conception objet : modèle de classes (modèle d'agents), modèle d'interaction et modèle de comportements. La plupart des méthodes commencent par une conception générale de l'architecture du système pour ensuite détailler les composants du système : agents et ressources. C'est lors de cette phase que la plupart des concepts multi-agents sont manipulés (rôle, groupe, organisation, tâche, etc.) et nécessitent parfois des notations spécifiques. Les interactions entre agents sont fréquemment spécifiées en utilisant les langages AUML, KQML ou ACL. La phase de conception résulte ainsi sur une définition claire de l'architecture du système multi-agent, décomposée en agents. Les comportements, les interactions entre les agents ainsi que les ressources nécessaires à l'atteinte de leurs buts sont également spécifiés de manière plus ou moins formelle en fonction des méthodes.

4.4.6.3 Implémentation

À partir de l'architecture définie à la phase de conception, la phase d'implémentation aboutit à la production du code testé. La mise en œuvre s'avère souvent difficile. Du point de vue de l'implantation, la manipulation de structures de données complexes, la distribution, les communications contribuent à cette difficulté. A cause de ces problèmes et aussi souvent des contraintes matérielles qu'ils imposaient, de nombreuses propositions de modèles sont malheureusement restées conceptuelles sans être étayées par des réalisations pratiques permettant de les valider. La facilité avec laquelle les applications pourront être développées est un facteur qui déterminera la rapidité de diffusion industrielle et commerciale des SMA. La généricité des modèles et des outils doit donc s'imposer afin d'évoluer vers la réutilisabilité. On rencontre donc aujourd'hui des environnements (plates forme) complets et génériques de développement de SMA comme : **Zeus**, **Jade**, **Madkit**,…etc. une plate-forme multi-agents consiste en un ensemble d'outils nécessaire à la construction et à la mise en service d'agents au sein d'un environnement spécifique. Une plate forme SMA peut servir

également à l'analyse et aux tests des systèmes ainsi créés. L'ensemble d'outils offerts par une plate forme peuvent être sous la forme d'environnement de programmation (API) et d'applications permettant d'aider et d'assister le développeur.

5 Composant, service et agent : analyse comparative

Pour mieux comparer les approches de développement à composant, à service et celles basées agent, il est intéressant de les situer dans le cadre d'analyse défini dans le chapitre III et qui présente un cadre d'évolution de la programmation. Ce cadre est un repère dont la première dimension représente le niveau d'abstraction alors que la seconde représente le couplage entre différentes entités. La projection sur la dimension du niveau d'abstraction reflète le besoin progressif d'abstraction qui n'a pas cessé d'augmenter. Ce besoin est témoigné par l'évolution de la programmation et du développement du logiciel allant des procédures et des structures de données, passant par les objets, les acteurs, les composants, les services, les modèles, les ontologies, les agents, les connaissances jusqu'à l'arrivée aux organisations et aux sociétés artificielles. La deuxième dimension représente les aspects liés aux liaisons entre différentes entités logicielles mises en jeu. Cette dimension reflète le besoin croissant en matière de description des liaisons et des relations indépendamment des entités qu'elles relient. Cette dimension exprime également la flexibilité des couplages signifiant la capacité de mettre des relations entre différentes entités logicielles. Le troisième aspect exprimé à travers cette dimension est le besoin de repousser le plus tard possible la décision de choisir l'action à exécuter et qu'elle est l'entité responsable de son exécution.

5.1 Apports des composants, services et agents sur le plan : abstraction

Sur le plan de l'abstraction trois éléments semblent essentiels pour définir l'espace dans lequel nous pouvons comparer les concepts et les technologies ayant un apport dans la programmation et le développement de logiciel d'une manière générale. Le premier élément représente les possibilités offertes pour identifier des abstractions de plus en plus de haut niveau ; le deuxième élément de définition résume l'évolution dans la représentation et l'explicitation des données et des connaissances manipulées ; le troisième élément concerne l'évolution en termes de représentations et d'explicitations des données et des connaissances échangées et communiquées.

Les composants, les service, les agents et les systèmes multi agents s'inscrivent dans le courant d'approches de développement de logiciel qui ciblent d'avoir de plus en plus des niveaux d'abstraction élevés. En particulier, les agents et les systèmes multi agents représentent des avancées par rapport aux composants et aux services dans le sens d'avoir plus de manipulations et d'explicitations des connaissances plutôt que de données.

La figure 3.3 du chapitre précédent illustre les positions des composants, des services, des agents et des systèmes multi agents dans un axe présentant l'évolution du niveau d'abstraction en programmation. D'une manière générale la programmation est passée de la manipulation des données à la manipulation des concepts. Les objets représente plus que des

structures de données, ils représentent des objets du monde réel, les composants et les services se distinguent d'une granularité plus importante et par la composition. Les agents repoussent encore plus loin le niveau d'abstraction. Les agents et les systèmes multi agents représentent des avancées technologiques par rapport aux composants et aux services, les agents et les systèmes multi agents prolongent l'évolution de l'abstraction à travers l'explicitation des connaissances et l'introduction des notions de croyance, but,…, et d'état mental dans les agents cognitifs. En plus que ces notions et ces concepts sont représentés explicitement généralement d'une manière symbolique, ils sont aussi communiqués et échangés entre les agents à fin que ces derniers puissent apprendre ou se coordonner entre eux.

Les langages de communication d'agents comme KQML et ACL apportent encore plus d'explicité et d'abstraction du fait que les informations comme la logique de coordination et qui sont implicites dans les approches à base de composants deviennent explicites. Ces langages peuvent être utilisés pour spécifier le contenu des messages échangés entre les agents. À titre d'exemple une communication ACL peut spécifier une désignation symbolique de l'intention de la communication; le langage de description du contenu utilisé pour décrire le contenu où il peut s'agir d'un langage de programmation comme Java ou un langage de représentation de connaissances adapté comme SL de FIPA ; et l'ontologie des concepts du message.

La conception des systèmes multi agents selon un point de vue social repousse encore plus loin l'abstraction. Cette conception guidée par l'organisation fait éloigner et rend plus tardives les questions de mise en œuvre. Généralement des concepts de très haut niveau d'abstraction tel que les rôles, les groupes d'agents, les mécanismes de réorganisation et l'auto organisation sont proposés et manipulés dans l'organisation des SMAs.

5.2 Apports des composants, services et agents sur le plan : liaisons

Les technologies de composants, de services et d'agents ont des apports concernant les aspects liés aux liaisons (description des liaisons et des relations, flexibilité des couplages et le besoin de repousser le plus tard possible la décision de choisir l'action à exécuter et qu'elle est l'entité responsable de son exécution.) . Si nous voyons les trois technologies dans un cadre de l'évolution de la programmation nous pouvons distinguer l'apport de chacune des technologies.

La programmation au début de son histoire était simple. Les différentes instructions sont identifiées par l'intermédiaire de leurs numéros de lignes, ce que signifie que la sélection de l'action à exécuter est exprimée globalement d'une manière statique. La situation est améliorée légèrement avec la programmation structurée où la modularité et l'encapsulation du code ont un impact sur la sélection de l'action à exécuter dans le sens où l'action à exécuter est exprimée via un nom symbolique et non par un numéro de ligne. Les langages de programmation orientés objet apportent une innovation importante avec la réunion des données et des procédures appelées méthodes. La liaison tardive était une avancée

déterminante, c'est-à-dire, la méthode à invoquer sera déterminée en fonction de la classe de l'objet effectivement invoqué et non pas en fonction de la déclaration. Nous pouvons remarquer que la sélection de l'action à exécuter est repoussée à l'exécution et non résolue statiquement. Les composants ont apporté les notions du prêt à porter, prêt à déployer et prêt à utiliser. Et là les composants se distinguent des objets qui nécessitent les codes écrits dans leurs classes et leurs superclasses. Les composants contiennent tous leurs codes et toutes leurs documentations. Les services sont prêts à utiliser sans que le développeur d'une application ait besoins de les déployer. Le service à utiliser n'est pas connu qu'au moment de l'exécution en faisant abstraction totale sur l'entité responsable de son exécution. Quant aux agents, la sélection de l'action à exécuter prend tous son sens. L'apport de ces derniers résident dans l'autonomie interne de la prise de décision, ce que signifie que la sélection de l'action à exécuter ne dépend pas totalement de l'entité externe qui à envoyer la requête à l'agent. La sélection de l'action à exécuter dépend de l'état interne de l'agent, de ses connaissances, et de ses objectifs. Les agents à travers l'introduction d'un ensemble de concepts avancés repoussent encore vers l'avant la sélection de l'action à exécuter. A titre d'exemple, la notion d'action persistante structurée dans laquelle l'agent tente d'une manière persistante d'accomplir une mission indépendamment de la manière selon laquelle l'agent est programmé. Dans ce mécanisme d'action le concepteur fournit une description de l'objectif sans programmer explicitement les tentatives. Il à noter que l'action persistante structurée n'est pas l'unique concept influant sur la sélection de l'action à exécuter, la négociation, la réorganisation et d'autres concepts avancés ont un impact certain sur la sélection de l'action à exécuter.

L'aspect couvert par le deuxième volet de la flexibilité du couplage concerne les concepts architecturaux de mise en relation structurelle entre entités et les modes de communication entre celle-ci. Les modes de communication sont caractérisés par les modes de désignation du receveur, les modes de transfert de données et les modes de couplage temporel.

Les composants logiciels apportent une amélioration pour le couplage structurel par l'externalisation des références et par leurs descriptions explicites sous la forme d'interfaces fournies (de sortie) et d'interfaces requises (d'entrée). Les services aussi apportent une telle amélioration à travers les descriptions de services. La manipulation du couplage devient ainsi explicite et externe aux entités logicielles. Le couplage est explicité dans le cas des composants par des connecteurs, qui vont relier les interfaces des composants. Les identifiants de ces interfaces sont appelés des « ports », d'entrées ou de sorties. Dans les technologies précédentes les références n'étaient pas externe, dans le paradigme objet par exemple un objet référence un autre par avoir un attribut dont la valeur est l'identifiant de l'objet référencé ce que implique des modifications à l'intérieur de l'objet en cas de besoin de modification de liaisons. Pour les composants la reconfiguration souhaitée s'opère par un simple ajout de connecteur. Un composant peut posséder plusieurs interfaces d'entrée, et de même pour les interfaces de sortie. C'est une différence importante avec un objet qui n'a lui

qu'un identifiant et qu'un seul point d'entrée. Une conséquence intéressante est que les composants sont compositionnels (les services aussi sont compositionnels), c'est-à-dire qu'une composition de plusieurs composants est équivalente à un composant qui posséderait la même union d'ensembles d'interfaces d'entrée et d'interfaces de sorties. Les objets ne sont pas directement compositionnels : une composition de plusieurs objets n'est pas immédiatement équivalente à un objet, car elle a plusieurs points d'entrée.

La vision architecturale explicite apportée par les composants et (les services) est encore un gain en terme d'abstraction de couplage puisque l'accent est mis sur la logique du couplage entre les composants (services) indépendamment de leur implantation interne en se servant des langages de description d'architecture dédiés à la spécification de l'architecture d'une application. Les informations de typage des interfaces des composants sont utilisées pour vérifier la correction de l'assemblage, c'est-à-dire la conformité entre les interfaces mises en relation. Il existe plusieurs types de connecteurs où chaque type s'adapte avec un ou plusieurs styles architecturaux ainsi que les protocoles de communication associés. Les approches à base de composant ne se limitent pas à exprimer des indications sur les types de données (typage) mais également sur les comportements des composants à travers les contrats qui peuvent contenir des indications : syntaxiques, comportementales, de synchronisation, et de qualité de service. Suivant les cas, ils peuvent être garantis, vérifiés ou négociés.

L'organisation du couplage et la coopération entre différentes entités logicielles sont poussées encore plus loin avec les systèmes multi-agents grâce à des mécanismes de coordination, décomposition, négociation et autres mécanismes permettant la collaboration entre plusieurs entités, et à l'aide aussi de la manipulation des connaissances comme l'organisation, les tâches et les plans. Le plus apporté par les systèmes multi-agents réside dans le fait qu'ils proposent un couplage sémantique guidé par les connaissances et par une organisation sociale du travail contrairement aux composants et les services où le couplage est principalement syntaxique (discipline des types) entre les composants. Le concept d'organisation est plus axé connaissances que le concept d'architecture logicielle.

Le plus souvent et dans plusieurs modèles organisationnels, l'organisation d'un système multi-agent est décrite sous la forme d'un ensemble de rôles et un ensemble de liens entre ces rôles. Les rôles sont joués par les agents. La manière selon laquelle les rôles sont attribués aux agents diffère d'un modèle à un autre. Cette abstraction de l'agent vers le rôle permet une description plus générique de l'architecture de l'application ainsi que les rapports et les interactions entre les agents. En jouant un rôle l'agent référence d'une manière indirecte et implicite l'ensemble des agents remplissant au moment de l'interaction ce rôle. De cette manière, le couplage structurel est externe au même titre que pour les composants et les services, mais avec un degré d'implicite de plus par rapport aux connexions explicites entre composants. Les systèmes multi-agents se distinguent aussi des

composants par l'utilisation de divers mécanismes de mise en relation dynamiques et indirects, par exemple, par l'intermédiaire de la consultation d'agents intermédiaires ou d'agents annuaires. La sélection et la contractualisation de partenaires dans l'objectif d'effectuer et de traiter des tâches peut s'effectuer par des mécanismes d'appels d'offre dont le protocole *contract net* est un bon exemple. Nous concluions par dire que les relations entre les agents sont très dynamiques et gérées en partie par les agents mêmes ou via les organisations. Cela aura pour effet d'apporter en plus de la dynamicité, les capacités d'autonomie ou d'auto réorganisation de l'organisation, qui peuvent évoluer en fonction des besoins soient guidées par un niveau plus haut, soit à l'initiative des agents eux-mêmes. Cela représente des avancées importantes par rapport aux approches basées composants malgré que la communauté des architectures logicielles et des composants aborde également ces enjeux de dynamicité et des capacités de reconfiguration automatique de l'architecture. Les SMAs répondent mieux aux besoins croissants de flexibilité et donc de délégation de l'initiative, ce qui augmente d'un autre coté les besoins pour l'assurance de certaines garanties sur le fonctionnement du système car l'approche multi-agent malgré quelle est plus ambitieuse, elle reste plus difficile à vérifier.

En termes de communication, les technologies d'agent et de composant ont leurs apports importants. Les composants introduisent une communication multipoints, du fait qu'une sortie d'un composant peut être connectée à plus d'un composant. Cela représentait une avancée par rapport aux objets où le mode de communication entre les objets n'était fondamentalement que point à point avec la désignation explicite du receveur du message. Plusieurs variantes de connecteurs entre composants ont été proposées offrant chacune de nouvelles possibilités pour la gestion indirecte et dynamique des connexions entre composants en introduisant des mécanismes de désignation de receveurs totalement implicites. De l'autre coté, les systèmes multi-agents généralisent le mécanisme de désignation indirecte et dynamique par l'intermédiaire de la consultation d'agents intermédiaires ou annuaires. Un agent peut sélectionner dynamiquement lui-même son interlocuteur d'une manière qui peut être aussi automatique et implicite. Les modèles SMA organisationnels basés sur la notion de rôle et les travaux visant à promouvoir l'environnement d'un système multi-agent comme une abstraction privilégiée peuvent servir d'exemples. Dans les modèles organisationnels à base de rôle, un agent peut s'adresser à un rôle, par conséquence à l'ensemble des agents remplissant ce rôle à l' instant de la communication. Le deuxième exemple représente les systèmes multi-agents dans lesquels l'environnement est modélisé explicitement, les agents peuvent communiquer via l'environnement, en laissant des données spécifiques. Les approches à services favorisent aussi le mécanisme de désignation indirecte et dynamique par l'intermédiaire de la consultation des annuaires et des courtiers.

La mobilité des agents signifiant la capacité d'un agent de se déplacer à travers un réseau pour accéder à des ressources distantes ou pour rencontrer d'autres agents n'apporte pas de nouveau ni en termes de sélection d'action à exécuter ni en termes de modes de

transfert de données mais elle a des influences sur la qualité du fonctionnement par rapprocher les clients et les serveurs et donc réduire le nombre et le volume des interactions distantes (en les remplaçants par des interactions locales) ce que permet de réduire le trafic sur le réseau.

6. Conclusion

Les approches à composants, à services et agents présentent trois approches de développement de logiciels ayants un impact important sur la programmation et l'industrie du logiciel. Toutes les trois proposent des abstractions pour la structuration et l'organisation du logiciel comme une combinaison de plusieurs entités dans le but de faciliter et de mieux maitriser son développement, sa maintenance et pour mieux gérer ses évolutions. Le prochain chapitre présente un autre paradigme de programmation. Il s'agit du paradigme aspect qui est très différent des paradigmes impératif, orienté objet, composants, services et agents vus jusqu'ici et qui présentent une évolution naturelle de la programmation en visant la recherche de niveaux d'abstraction de plus en plus élevés et une flexibilité de plus en plus grande du couplage entre les différentes entités constituant un logiciel.

Chapitre V :

La programmation orientée aspect

1. Introduction

Parmi les approches les plus utilisées pour le développement de logiciels figurent certainement les approches objet et composant et à un degré moindre figurent les services, vu les avantages importants que ces approches proposent pour la modularité, la lisibilité et la compréhension des programmes, la possibilité de les réutiliser, et de les faire évoluer facilement et de manière fiable. Ces approches, malgré leurs impacts importants sur la qualité du développement des logiciels, ont aussi leurs lots d'inconvénients et présentent un certain nombre de lacunes et problèmes qui les rendent, aujourd'hui, insuffisants afin de prendre en compte la complexité croissante des nouvelles applications. Cette complexité dérive principalement du besoin d'évolution des applications et qui rend leurs développements plus difficile, plus couteux et moins fiable. Afin de faciliter l'évolution et favoriser la réutilisation des systèmes logiciels produits, de nouvelles voies de développement ont été explorées donnant naissances à de nouvelles approches de développement basées sur la notion d'Aspect, l'Ingénierie Dirigée par les Modèles (IDM ou MDA pour Model Driven Architecture), ou les méthodes agiles.

Dans ce chapitre nous allons étudier l'approche de développement basée sur la notion d'aspect dont la colonne vertébrale est le principe de séparation des préoccupations.

2. Séparation des préoccupations

Pour faire face à la complexité du développement des Systèmes logiciels, la séparation des préoccupations, lors du cycle de développement des applications, s'est avérée être une solution prometteuse. La séparation des préoccupations (SoC, Separation of Concerns) est un concept présent depuis très longtemps dans l'ingénierie des logiciels. Les différentes préoccupations des concepteurs apparaissent comme les motivations premières pour

décomposer une application en un ensemble d'éléments manipulables les uns indépendamment des autres et dont la compréhension est plus facile. La séparation en préoccupations apparaît dans les différentes phases du cycle de vie du logiciel et par conséquence elles sont de différentes natures. Les préoccupations peuvent être d'ordre fonctionnel, technique ou encore liées aux rôles des acteurs du processus logiciel. Par ces séparations, le logiciel n'est plus abordé dans sa globalité, mais par parties.

Une préoccupation peut être définie comme étant l'abstraction d'un but particulier ou une unité modulaire d'intérêt. Pour chaque système il existe essentiellement deux catégories de préoccupations : des préoccupations fonctionnelles et des préoccupations non fonctionnelles. Lors du développement d'un système logiciel, différentes préoccupations apparaissent, par exemple : des préoccupations liées aux données, à leur persistance et à leur contrôle d'accès, ou encore, des préoccupations architecturales telles que l'interopérabilité entre applications ou des préoccupations de gestion organisationnelle. A titre d'illustration, dans un système de gestion de cartes de crédit, le processus de paiement est une préoccupation fonctionnelle ; l'intégrité de la transaction de paiement et la gestion de l'authentification lors de la connexion sont des préoccupations non fonctionnelles.

L'objectif principal de la séparation des préoccupations est de représenter de manière abstraite et explicite, à tous les niveaux de développement, les différentes préoccupations d'un système, et de les approcher indépendamment les une des autres tout en identifiant leurs interrelations afin de les composer. La séparation des préoccupations est un principe clé pour la réduction de la complexité du développement de grands systèmes. Si le code relatif à chaque préoccupation est isolé, cela permet de comprendre comment la préoccupation est adressée dans le code qui devient facile à étendre, à modifier et à réutiliser.

2.1 Les préoccupations transversales

Le principe de séparation des préoccupations n'indique pas comment arriver par un processus précis de développement à implémenter des préoccupations. Plusieurs modèles et langages de programmation proposent des supports offrant différents mécanismes et concepts qui permettent une meilleure organisation des programmes selon ce principe. Il s'agit en général d'organiser les programmes en unités modulaires séparées (procédures, fonctions, objets, etc.) concernant chacune une préoccupation particulière. Les méthodes classiques assurent un traitement pour la séparation et la composition de préoccupations fonctionnelles. Elles ne garantissent pas en revanche, une représentation modulaire et séparée des préoccupations, qui affectent à la fois plus d'une unité modulaire fonctionnelle. Le code de telles préoccupations, dites transversales (Crosscutting Concerns), se trouve ainsi dispersé et enchevêtré dans l'ensemble du code des préoccupations fonctionnelles déjà implémentées. Ceci présente une source de plusieurs problèmes liés à la compréhension, la maintenance, l'évolution et la réutilisation du code relatif à ces préoccupations transversales, et de celui relatif à l'application.

Les préoccupations transversales peuvent être de différentes natures, et on peut distinguer celles qui sont liées aux données (persistance de données, contrôle d'accès, etc.) ; des préoccupations de logging/tracing, de gestion d'exceptions, de debugging et de tests ; des préoccupations de gestion organisationnelle ; des préoccupations techniques et architecturales, telles que la configuration, la distribution, la synchronisation, l'interopérabilité entre applications hétérogènes, la sécurité, la gestion de performance, etc.

Une analyse orientée objet d'une application conduit à organiser cette dernière sous la forme d'un ensemble de classes dont les instances peuvent être corrélées. Par exemple, dans une application qui contient une classe *Client* et une autre classe *Commande*, les objets clients peuvent être corrélés avec des objets de type commande d'où il faut s'assurer avant de supprimer un client qu'il n'a aucune Commande non satisfaite dans la classe Commande. De l'autre côté, la suppression d'une commande peut engendrer la perte des coordonnées du client qui a passé cette commande. Comme solution à ce problème, on peut proposer la modification de la méthode de suppression de la classe *Client*, de sorte à intégrer dans cette dernière la vérification au préalable de l'absence de *Commande* non satisfaites relevant du *Client* à supprimer. Une telle solution qui semble bonne est pratique présente cependant quelques inconvénients : (*i*) la vérification des commandes non satisfaites ne fait pas partie de la logique de gestion d'un client. Alors que la classe des clients devrait gérer les fonctionnalités qui relèvent de la propre logique du client. (*ii*) Une classe n'est pas censée connaître toutes les contraintes d'intégrité imposées par les autres classes de l'application.(*iii*) Si une classe implémente des fonctionnalités liées à d'autres classes (la classe *Client* dans cet exemple implémente une fonctionnalité de la classe *Commande*), ceci limite les possibilités de sa réutilisation indépendamment des classes liées. D'un autre côté, la classe *Commande* n'est pas aussi adaptée à l'implémentation de cette contrainte : si nous voulons supprimer un *Client*, il n'y a aucune raison pour vérifier si la classe Commande le permet.

Il est claire que ni la classe *Client* ni la classe *commande* présentent le meilleur emplacement pour implémenter cette contrainte d'intégrité référentielle. Cette contrainte n'est pas propre ni à la classe *Client* ni de la classe *Commande*, elle est transversale aux deux classes. En présence de fonctionnalités transversales, les classes se trouvent couplées et donc sont dépendantes les une des autres. La programmation orientée objet ne propose aucune solution pour prendre en compte les fonctionnalités transverses. Dans ce paradigme de programmation, l'implémentation d'une méthode est localisée dans une classe, et\ ou son utilisation se trouve dispersée. Cette manière de situer et utiliser les méthodes est un réel obstacle pour la maintenance et l'évolution des applications orientées objet.

2.2Effets produits par la prise en compte de la séparation des préoccupations

La séparation des préoccupations vise de séparer les fonctionnalités qu'un logiciel doit réaliser et les préoccupations qui contraignent le déroulement de ces fonctionnalités. Cette séparation est à l'origine de plusieurs avantages :

- *Découplage et élimination des dépendances* : La séparation des préoccupations permet éliminer l'enchevêtrement des préoccupations avec les fonctionnalités et les autres préoccupations. Par conséquence, les changements qui peuvent toucher certaines préoccupations dans un but d'évolution ou d'une maintenance auront un effet réduit et bien cerné sur les autres.
- *Séparation des services communs* : en plus des fonctionnalités qu'une application doit assurer, cette dernière contient aussi généralement des services qui sont constitués par des préoccupations. Par exemple, un service de sécurité ou un service de synchronisation sont du genre de services constitués par des préoccupations. Une séparation des préoccupations permet à ce genre de services d'être séparés et de subir à des changements sans modifier le reste de l'application.
- *Doter le système de moyens d'adaptation dynamique :* un développement avec une séparation des préoccupations ouvre la voie pour plusieurs possibilités d'adaptation statique et dynamique (en cours d'exécution). Ces possibilités présentent les facultés d'ajouter et de retirer de manière statique ou dynamique des aspects sans conséquences sur le reste de l'application.

2.2.3 La séparation des préoccupations et la programmation impérative, orientée objet et par composant.

La programmation impérative met l'accent sur les fonctionnalités attendues du système concret. L'ensemble des constructions du programme partagent les données du système. La programmation procédurale atteint ses limites avec l'augmentation de la complexité des systèmes informatiques. Ainsi, la réutilisation, l'évolution ainsi que la maintenance des applications à base de procédures et fonctions deviennent difficiles. Les concepts de modularité, d'encapsulation et d'abstraction des données introduits dans les langages de programmation impératifs les plus modernes comme Modula et ADA, ne garantissent pas l'application du principe de la séparation des préoccupations, à cause des différentes interactions qui se trouvent non encapsulées et dispersées dans les différents modules qui composent un programme.

Le découpage d'une application en termes de classes dans la programmation orientée objet n'est pas en mesure d'assurer une réelle solution aux problèmes de dispersion et d'enchevêtrement du code d'une application comme il est illustré dans la section 2.1.

Les modèles à base de composants bien qu'ils permettent une mise en œuvre de manière transparente et flexible de certaines préoccupations techniques (telles que la gestion de la distribution, de la synchronisation, des transactions ou encore de la persistance), ils ne prennent en compte qu'un nombre limité de préoccupations.

3. La séparation des préoccupations et le paradigme Aspect

La programmation orientée aspect (POA) est le produit de recherches poussées par les limitations dont souffre la programmation orientée objet dans l'objectif de préconiser une réelle séparation des préoccupations qui assure, entre autres, une meilleure modularité et réutilisabilité du code source. La programmation orientée aspect propose d'encapsuler dans des aspects les préoccupations incompatibles avec les logiques métiers des objets. Le code nécessaire à ces préoccupations est représenté par des aspects qui injectent ensuite le code nécessaire à ces préoccupations.

L'idée principale de la programmation orientée aspect était de rendre le code source du programme plus lisible et beaucoup plus réutilisable, en implémentant de manière séparée les préoccupations transversales.

La plupart des logiciels développés répondent à des exigences fonctionnelles décrivant leurs comportements et définissent les fonctions ou les services que ces derniers doivent remplir. D'un autre côté, ces logiciels répondent aussi à des exigences non fonctionnelles qui expriment souvent les qualités ou les contraintes imposées sur la manière de satisfaire les exigences fonctionnelles. Le paradigme objet et celui de la programmation impérative posent problème avec l'implémentation des exigences non fonctionnelles qui se trouvent dispersées dans les différentes entités fonctionnelles d'un système, du fait qu'il est difficile d'en limiter la portée dans un domaine bien circonscrit. En réponse à ces problèmes, plusieurs modèles permettant une meilleure séparation et composition de tout type de préoccupations ainsi que des langages de programmation associés sont proposés. Ces modèles sont connus sous la désignation d'approche Aspect. Ils prônent une décomposition des programmes non seulement en unités modulaires représentant les préoccupations fonctionnelles de base, mais aussi en unités modulaires dédiées à la représentation des préoccupations transversales. Ils offrent en plus des solutions adéquates de composition de préoccupations dans le but de construire des systèmes efficaces.

3.1 Modèles de programmation introduits dans le cadre de l'approche Aspect

Les modèles et les langages de programmation, introduits dans le cadre de l'approche Aspect, sont définis le plus souvent comme étant des extensions des modèles et langages de programmation déjà existants (procéduraux et fonctionnels, Objets, Composants, etc.). Les principaux modèles de programmation introduits dans le cadre de l'approche Aspect sont : la programmation adaptative ; La programmation par rôles ou par points de vue ; La programmation par sujet ainsi que plusieurs autres approches.

La programmation adaptative dite aussi programmation orientée patron tente de résoudre les problèmes de dépendances entre comportements et structures d'objets d'une application afin d'assurer une meilleure séparation des préoccupations. La programmation adaptative modélise les préoccupations transversales dans des *patrons* qui sont classés en différentes catégories telles que les patrons de propagation d'opérations sur les données et

les patrons de synchronisation d'accès concurrents afin d'apporter des solutions à des problèmes dont la source est l'augmentation considérable de dépendances entre objets utilisés dans toute collaboration à base d'objets assurée par un ensemble de méthodes, qui interagissent entre elles, en mettant en jeu plusieurs propriétés structurelles.

La programmation par rôles ou par points de vue est aussi l'une des sortes de programmation introduites dans le cadre de l'approche Aspect. Dans ce genre de programmation, les programmes sont structurés en groupes d'objets qui représentent les différents points de vue du système à développer. A la différence de l'approche objet où une entité du monde réel est représentée par un objet, en programmation par rôles une entité du monde réel est représentée par un ou plusieurs objets. Selon le point de vue considéré, une entité peut jouer plusieurs rôles. Chaque objet modélise un rôle particulier que peut jouer une entité. L'ensemble des préoccupations de l'application représentent les différentes vues subjectives du système. Chaque préoccupation est représentée par les différents rôles et leurs interactions. Ce genre de programmation présente plusieurs limites liées à la séparation des préoccupations à cause des problèmes de partage de données et de recouvrement de diverses définitions entre préoccupations distinctes. Une autre limite consiste en la difficulté de mise en œuvre de l'intégration de propriétés d'objets représentant une même entité du monde réel mais appartenant à des groupes d'objets différents.

Une autre technique pour la séparation des préoccupations fut introduite par la programmation par sujets qui consiste à structurer les programmes en sujets représentant les différents développeurs, utilisateurs du système, et les contextes d'utilisation. Dans cette approche, chaque sujet représente une préoccupation particulière. Un sujet est défini comme étant une collection d'états et de comportements d'un groupe d'objets/classes ou fragments de classes liés entre eux grâce à l'héritage ou aux autres relations. Ces objets reflètent une perception du monde réel. Dans ce contexte, la composition de l'ensemble des sujets considérés produit le système final.

4. Les concepts de base de la programmation orientée Aspect

Chaque paradigme de programmation se base sur un ensemble de concepts clés. Les concepts fondamentaux de la programmation orientée aspect sont les concepts : d'aspect, de point de jonction, de coupe et de greffon.

La programmation par aspect s'applique généralement aux langages orientés objet. Il existe des implémentations orientées Aspect spécifiques à certains langages orientés objet. Par exemple, les langages AspectC++ ou C# pour le langage C++ . AspectJ est l'une des implémentations du paradigme orienté Aspect appliquée au langage Java.

4.1 Le concept d'aspect

Le concept d'aspect est proposé pour capturer les différentes préoccupations transversales d'un système. Un aspect peut être défini comme une entité logicielle qui capture une fonctionnalité transversale.

Dans le paradigme aspect, une application comporte des entités logicielles (classes, modules,…) et des aspects. Un aspect implémente une fonctionnalité transversale à l'application, qui se retrouve dispersée dans des classes en programmation orientée objet, et dans des modules, dans la programmation procédurale comme il est illustré dans la figure 5.1.

4.2 Point de jonction

La programmation par aspect permet, par le biais de ces mécanismes, d'intervenir au niveau d'un programme orienté objet dans un point quelconque de l'exécution pour injecter le comportement nécessaire à la réalisation de la préoccupation transversale. Ces points sont dits : points de jonction.

Un point de jonction (*pointcut*) se définit comme un point dans l'exécution d'un programme autour duquel un ou plusieurs aspects peuvent être définis. Pour désigner de manière concrète cette notion de point de jonction dans un code d'aspect, le concept de coupe est utilisé.

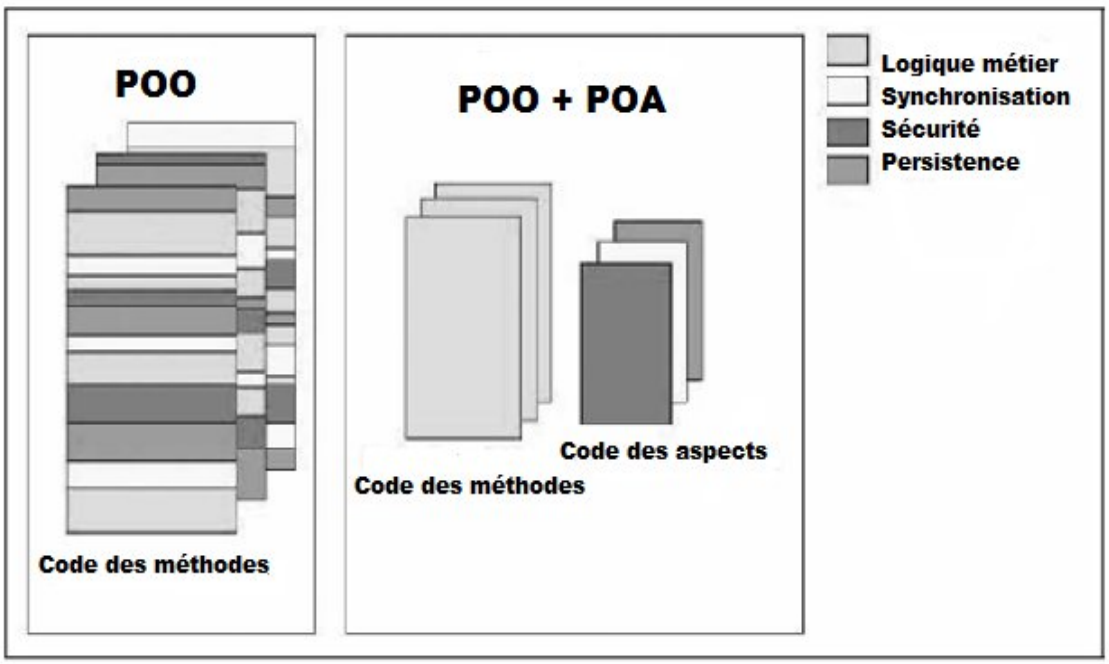


Figure 5.1 : *L'encapsulation des préoccupations transversales*

Le point de jonction est le concept qui permet de définir un point précis dans l'exécution d'un programme pour injecter une action. Le langage AspectJ par exemple permet de spécifier plusieurs types de points de jonction. On distingue : appel et exécution de

méthode ; appels et exécutions de constructeurs ; accès en lecture ou en écriture à un attribut ; exécution d'un gestionnaire d'exception et initialisation d'une classe ou d'un objet.

4.3 La notion de coupe

Une coupe désigne un ensemble de points de jonctions. Une coupe est définie à l'intérieur d'un aspect. Une seule coupe peut suffire pour définir la structure transversale d'un aspect dans les cas simples. Dans les cas plus complexes, un aspect est associé à plusieurs coupes. Les notions de coupe et de point de jonction sont liées par leur définition. Une coupe est un choix de point de jonction défini dans un aspect, alors qu'un point de jonction est un emplacement dans l'exécution d'un programme. De ce fait un point de jonction peut appartenir à plusieurs coupes dans un même aspect ou dans des aspects différents.

En AspectJ, la déclaration d'un point de coupure se fait avec le mot clé *Pointcut*. Le Listing 5.1 contient un extrait qui montre la déclaration d'un point de coupure, appelé *Pointmoved*. Ce point de coupure représente un appel à une méthode préfixée par *set* de la classe Point. De plus cette méthode est de type public et retourne la valeur void. Il est mentionné que le point de coupure fait référence à un appel de méthode grâce au prédicat *call*.

```
Pointcut pointmoved () : call(public void Point.set*(int))
```

Listing 5.1 : Déclaration d'un point de coupure.

4.4 La notion de greffon

Un greffon (advice) définit ce que l'aspect greffe dans l'application, c'est le code ajouté par l'aspect. Un aspect définit un ou plusieurs greffons, que chacun réalise un comportement particulier de son aspect. A la différence d'une méthode en programmation orientée objet, un greffon est associé à une coupe, et donc à des points de jonction. Un greffon a un type qui détermine le moment de son exécution (before, after, arround).

En AspectJ, les mots clés : `After`, `Before` et `Around` sont utilisés. After indique que le greffon est exécuté après le point de coupure associé. Before indique que le greffon est exécuté avant le point de coupure associé. Around le greffon est exécuté à la place du point de coupure associé.

5. Exemple de programme en AspectJ

En AspectJ, les aspects sont des unités modulaires permettant d'implémenter des préoccupations transversales. Les Aspects sont déclarés par le mot clé `aspect` et peuvent contenir des méthodes, des attributs, des points de coupure, des greffons, et des mécanismes d'introduction.

Le tissage (Weaving) est l'étape concrète qui permet d'injecter le comportement décrit par les Aspects dans un programme orienté objet. Le tissage peut se faire de manière statique comme le fait AspectJ grâce à son compilateur ajc ou de manière dynamique à l'exécution du programme.

Le listing 5.2 contient un extrait d'un programme écrit en AspectJ. C'est un programme de gestion de commandes d'articles par des clients. Le programme permet à un client d'ajouter des articles à une commande et de calculer le montant en utilisant un catalogue d'articles dans lequel sont mentionnés les prix des différents articles. L'application est constituée de trois classes : Client, Ordre et Catalogue. La classe Client présente le point d'entrée de l'application.

La méthode `main()` crée un objet Client et invoque la méthode `run` qui crée une commande (objet m_Ordre), appelle deux fois la méthode addItem avec comme paramètres une référence d'un article et une quantité, et puis calcule et affiche le montant de la commande.

```
package aop.aspectJ;
public class Client{
public void run(){
Ordre m_Ordre= new Ordre();
m_Ordre.addItem("CD", 2);
m_Ordre.addItem("DVD", 1);
double montant = m_Ordre.computeAmount();
System.out.println("Montant de la commande: "+montant+"DA");
}
public static void main(String[] args)
new Client().run();
}}
```

Listing 5.2 : Un exemple de programme en AspectJ

L'affichage de l'ensemble des articles commandés est attribué à un Aspect qui affiche un message avant et après les appels de la méthode `addItem` de la classe Ordre comme le montre l'extrait du programme du listing 5.3. Cet aspect contient un point de coupure `ToBeTaced` composé d'un seul point de jonction (`call (public void Ordre.addItem(string,int))`) qui capture l'appel de la méthode `addItem` de la classe Ordre. Le greffon `((void around(): ToBeTraced)` utilisé est de type `around` attaché au point de coupure `ToBeTaced`.

```
package aop.aspectJ;
public aspect traceAspect{
pointcut: ToBeTraced():
call (public void Ordre.addItem(string,int));
void around(): ToBeTraced {
System.out.println("Avant Appel addItem");
proceed();
System.out.println("Après Appel addItem");} }
```

Listing 5.3 : Programmation d'un aspect

6. Conclusion

Ce chapitre a mis l'accent sur le principe de la séparation des préoccupations, ainsi que le concept des préoccupations transversales. Dans un second temps, ce 5ème chapitre présente le paradigme Aspect et ses concepts fondamentaux tout en montrant comment ces concepts sont supportés par le langage de programmation orienté aspect AspectJ, considéré comme le plus connu dans la catégorie des langages de programmation orientée aspect. Un exemple est donné en fin de chapitre, pour illustrer d'une manière très brève et concise les concepts de la POA sous AspectJ. Le chapitre suivant aborde un paradigme de programmation différent de tous les paradigmes vus jusqu'ici et que sont qualifiés de paradigmes applicatifs. Il s'agit de la programmation fonctionnelle qui est regroupée avec la programmation logique sous la rubrique de : *programmation déclarative.*

Chapitre VI :

La programmation fonctionnelle.

1. Introduction

A ses débuts, la programmation dite évoluée ou de haut niveau était principalement impérative. Dans le style impératif, un programme consiste en un ensemble de variables et un ensemble d'instructions qui inspectent et mettent à jour ces variables. Cette manière de programmer est une abstraction de l'architecture de base des ordinateurs et du processus d'exécution d'instructions sur un ordinateur. Les langages de programmation impératifs se prêtent mal à l'exécution simultanée de différentes parties d'un programme, parce que toute commande peut dépendre des changements aux variables causés par les commandes précédentes.

Par contraste avec la programmation impérative, la programmation fonctionnelle présente une autre voie pour écrire des programmes sans utilisation de variables et d'instructions d'affectation ni encore de structures de contrôle itératives. Historiquement, le premier langage de programmation fonctionnel est le langage LISP (LISt Processor), créé par John McCarthy vers 1958, un peu après FORTRAN (FORmula TRANslator, 1955). Alors que FORTRAN naquit des besoins numériques des militaires et des gestionnaires, LISP vit le jour au MIT [Massachusetts Institute of Technology] pour résoudre des problèmes plus "symboliques" liés à l'Intelligence Artificielle. La programmation fonctionnelle est basée sur le concept de fonctions mathématiques, qui sont souvent définies par la séparation entre différents cas qui présentent des fonctions à leurs tours. Ainsi, un programme fonctionnel peut être vu comme une fonction définie par d'autres fonctions.

2. Concepts clés de la programmation fonctionnelle

En programmation fonctionnelle, les calculs sont considérés comme des évaluations de fonctions mathématiques où le modèle de computation est d'appliquer des fonctions sur des

arguments. Les concepts clés pour ce paradigme de programmation sont : les expressions, les fonctions et le polymorphisme. D'autres concepts caractérisent quelques langages fonctionnels ; il s'agit de l'abstraction de données et de l'évaluation tardive.

2.1 Les expressions

Une expression permet de produire une nouvelle valeur à partir de valeurs existantes. En programmation fonctionnelle, contrairement à la programmation impérative, les expressions et les instructions ne sont pas distinctes. En C par exemple, il n'est pas possible d'écrire une instruction telle que : `1+ (if (x !=0) f() else h()).` Par contre, dans un langage fonctionnel comme OCaml où il n'y a que des expressions, on peut écrire par exemple : `1+(if(x=0) then 5 else 2)` car la construction `if-then-else` est une expression qui s'évalue de la manière suivante : son premier opérande est évalué et si le résultat vaut *true* le deuxième opérande est évalué et son résultat est celui de toute l'expression if ; sinon c'est le troisième opérande qui est évalué et donne le résultat de toute l'expression.

2.2 Les fonctions

La fonction est un concept clé pour la programmation fonctionnelle. Elle fait une abstraction sur les expressions. La fonction est le type de base manipulé dans la programmation fonctionnelle, elle peut être passée comme argument à une autre fonction, retournée comme résultat par une fonction, intégrées dans des valeurs composites,…etc.

Le terme *fonction* est à prendre dans ce paradigme de programmation au sens mathématique du terme, c'est-à-dire une relation entre deux ensembles A et B telle que tout élément de l'ensemble A soit en relation avec au plus un élément de l'ensemble B. Si on nomme f cette fonction, et x un élément de A on note habituellement $f(x)$ l'élément de B associé à x et on dit que $f(x)$ est l'image de x par f si cette image est définie.

Une fonction qui prend une autre fonction comme argument ou qui produit une fonction en retour est dite une fonction d'ordre supérieur. Les fonctions d'ordre supérieur permettent de définir ce que les choses sont plutôt que des étapes qui changent un état et bouclent. Elles sont un moyen très puissant pour résoudre des problèmes et pour faire des réflexions sur les programmes. Le listing 6.1 contient un code Haskell dans lequel est définie une fonction d'ordre supérieur appelée *map* applique une fonction à tous les éléments d'une liste :

```
map :: (a -> b) -> [a] -> [b]

Par exemple :

> map (+1) [1 ,3 ,5 ,7]
[2 ,4 ,6 ,8]
```

Listing 6.1 : Exemple d'une fonction d'ordre supérieur.

2.3 Polymorphisme paramétrique

Le polymorphisme paramétrique est un concept clé de la programmation fonctionnelle. C'est une forme de polymorphisme où les types de fonction sont paramétrables. Le polymorphisme paramétrique permet à une fonction d'opérer sur des valeurs d'un ensemble de types (plutôt que seulement un seul type). Examinons le code Haskell dans le Listing 6.2. Le code contient une définition d'une fonction identité. Nous remarquons qu'il y a plusieurs versions de cette fonction (une version par type) même si elles ont la même implémentation. Le code Haskell du listing 6.3 fait la même chose en utilisant un type paramétrique exprimé avec une lettre minuscule pour dire que cela fonctionne sur tout type.

```
idInts :: Int -> Int
idInt x = x

idString :: String -> String
idString x = x
```

Listing 6.2 : Plusieurs versions d'une fonction d'identité

```
id :: a -> a
id x = x
```

Listing 6.3 : Une fonction d'identité avec polymorphisme paramétrique

2.4 Abstraction de données

L'abstraction de données n'est pas propre à la programmation fonctionnelle. C'est un concept clé dans les langages fonctionnels les plus modernes tels que ML et Haskell. L'abstraction de données consiste à manipuler des données définies de manière abstraite indépendamment de la façon dont elles sont implantées soutenant ainsi la séparation des préoccupations. Ce qui est essentiel pour la conception et la mise en œuvre de programmes de grande envergure. Dans un langage fonctionnel, toutes les opérations d'un type abstrait sont des constantes et des fonctions. Ainsi, un type abstrait peut être équipé d'opérations pour calculer de nouvelles valeurs du type à partir d'ancienne. Donc Il n'est pas possible pour un type abstrait d'être équipé d'une opération qui met à jour sélectivement une variable du type, comme dans un langage impératif.

2.5 L'évaluation tardive

L'évaluation tardive ou (*lazy evaluation*) en anglais, est l'un des principes de la programmation fonctionnelle que l'on trouve que dans les langages fonctionnels les plus modernes. Selon ce principe, l'expression dont la valeur n'est jamais utilisée ne doit jamais

être évaluée. Cela signifie que les expressions ne sont pas évaluées quand ils sont liés à des variables, mais leurs évaluations sont reportées jusqu'à ce que leurs résultats soient nécessaires par d'autres calculs. En conséquence, les arguments ne sont pas évalués avant qu'ils ne soient transférés à une fonction, mais seulement lorsque leurs valeurs sont effectivement utilisées.

Dans les langages qui ne supportent pas ce principe, les paramètres réels d'une fonction sont évalués, soit : (i) une fois, au point de l'appel. En effet, les valeurs des arguments sont substituées pour chaque occurrence de chaque paramètre formel. Soit (ii), lorsque les arguments sont réellement nécessaires. En effet, il s'agit d'une substitution des paramètres réels eux-mêmes (expressions non évaluées) pour chaque occurrence de chaque paramètre formel.

Pour illustrer la situation, considérons la fonction *f* définie comme *f n = n * n* où *n* est le paramètre formel de *f*. Considérons l'appel *f(a + 1)* où la valeur de *a* est 5. Dans le premier cas, le cas (i), *a+1* est calculée d'abord pour produire la valeur 6. La valeur 6 substitue toutes les occurrences de *n*. Par conséquence, l'évaluation de *f* revient à calculer 6*6 pour produire 36. Dans le deuxième cas, cas (ii), chaque occurrence du paramètre formel *n* est substituée par (5+1) son évaluation. Par conséquence, l'évaluation de *f* revient à calculer (5+1)*(5+1) pour produire 36 aussi. Nous remarquons que selon les deux modes d'évaluation, la fonction *f* produit le même résultat. Ceci n'est pas toujours valable pour toutes les fonctions. Il existe des cas où l'ordre d'évaluation des paramètres influence le résultat produit par une fonction. Considérons la fonction *cond* définie de la manière suivante : *cand c1 c2 =* ***if*** *c1* ***then*** *c2* ***else*** *false*. *c1* et *c2* présentent les paramètres formels de la fonction *cond*. Considérons aussi un appel de *cond* avec les deux paramètres effectifs (*n> 1*) et (*t/n> 30*) où la valeur de *n* est 0 et la valeur de *t* est 120. Pour le cas (i), (*n> 1*) est évaluée à *false* et (*t/n> 30*) provoque un déroutement causé par une division sur zéro. D'où l'échec de l'exécution de *cond*. Pour le cas (ii), on remplace les paramètres formels c1 et c2 par les paramètres effectifs (*n> 1*) et (*t/n> 30*). L'évaluation de la fonction *cond* revient à évaluer l'expression "***if*** *(n> 1)* ***then*** (*t/n> 30*) ***else*** *false*" qui produit *false* puisque (*n> 1*) s'évalue à *false* d'où la section ***else*** du ***if*** est actionnée pour retournée *false*.

Pratiquement, le deuxième mode d'évaluation dit l'évaluation d'ordre normale est trop inefficace pour être utilisé en programmation. Un paramètre effectif pourrait être évalué à plusieurs reprises, en donnant toujours le même argument. Pour éviter cela, les langages fonctionnels modernes proposent un mécanisme d'évaluation tardive dans lequel un paramètre est calculé à la première fois quand il est nécessaire, la valeur de l'argument est stockée pour être utilisée à chaque fois que l'argument est nécessaire.

Si dans un langage la propriété dite propriété de *Church–Rosser* est respectée ; l'évaluation tardive produit les mêmes résultats qu'une évaluation d'ordre normale. La propriété de *Church–Rosser* stipule que « *si une expression peut être complètement évaluée, elle doit être apte à être évaluée dans l'ordre normale. Si une expression peut être évaluée dans les trois*

ordres d'évaluation, le résultat produit par l'expression doit être le même quelque soit l'ordre d'évaluation »

3. Caractéristiques de la programmation fonctionnelle

La programmation fonctionnelle est un paradigme de programmation qui considère le calcul en tant qu'évaluations de fonctions et rejette le changement d'état et la mutation des données. Elle souligne l'application des fonctions, contrairement au modèle de programmation impérative qui met en avant les changements d'état. La programmation fonctionnelle s'affranchit de façon radicale des effets secondaires en interdisant toute opération d'affectation (principalement l'effet de bord). Un avantage important des fonctions sans effet de bord est la facilité que l'on a à les tester unitairement. En pratique, pour des raisons d'efficacité, et du fait que certains algorithmes s'expriment aisément avec une machine d'états. Certains langages fonctionnels autorisent la programmation impérative en permettant de spécifier que certaines variables sont assignables et donc la possibilité d'introduire localement des effets de bord. Ces langages sont regroupés sous le nom de langages fonctionnels impurs.

La programmation fonctionnelle n'utilise pas de machine d'états pour décrire un programme, mais un emboîtement de fonctions. Un programme est donc une application, au sens mathématique, qui ne donne qu'un seul résultat pour chaque ensemble de valeurs en entrée. Cette façon de penser, qui est très différente de la pensée habituelle en programmation impérative est l'une des causes principales de la difficulté que trouvent les programmeurs formés aux langages impératifs pour aborder la programmation fonctionnelle. Cependant, elle ne pose généralement pas de difficultés particulières aux débutants qui n'ont jamais été exposés à des langages impératifs.

L'implémentation des langages fonctionnels fait un usage sophistiqué de la structure pile. Afin de s'affranchir de la nécessité de stocker des données temporaires dans des tableaux; les langages fonctionnel font largement appel à la récursivité (fait d'inclure l'appel d'une fonction dans sa propre définition).

Les langages fonctionnels sont caractérisés par la propriété de *transparence référentielle.* Principe selon lequel le résultat d'un programme ne change pas si on remplace une expression par une expression de valeur égale. Ce principe facilite les transformations de programme par exemple pour en améliorer les performances. Par exemple, si nous voulons remplacer $f(x)+f(x)$ par $2*f(x)$ dans un programme. Dans le cas ou la fonction f est à effets de bord, on ne peut pas remplacer $f(x)+f(x)$ par $2*f(x)$ car f ne dépend pas uniquement de ses arguments d'entrée; et elle ne se comporte pas forcément de façon identique à deux instants donnés lors de l'exécution du programme.

Pour bien illustrer la propriété de transparence référentielle, examinons le code *C* dans le listing 6.4 où n désigne une variable globale visible par tout le programme contenant un entier à incrémenter et $inc(k)$ est une fonction qui augmente la valeur de n de la quantité k.

```
int n = 2;
int inc(int k) { n = n + k; return n; }
/* incrémentation par effet de bord */
……
Somme= inc(1) + inc(1);
```

Listing 6.4 : Une fonction C à effet de bord

La fonction *inc* ne retourne pas la même valeur lors des deux appels malgré que la même valeur « 1 » est passée comme argument effectif à la fonction lors des deux appels. Dans l'instruction `Somme= inc(1) + inc(1),` l'un des opérandes de l'addition vaudra (2 + 1 = 3) et l'autre (3 + 1 = 4). Il s'avère donc impossible de remplacer *inc(i)* + *inc(i)* par *2 * inc(i)* car la valeur de *inc(i)* diffère à chaque appel. À l'échelle d'un programme important, cela signifie que le remplacement d'une fonction par une autre peut modifier son comportement global. À l'inverse, la propriété de transparence référentielle permet d'assurer que le remplacement d'une fonction par une autre équivalente ne risque pas de modifier le comportement global du programme. Cette propriété facilite la maintenance ainsi que l'évolution des logicielles.

En programmation fonctionnelle, la valeur d'une expression dépend seulement des valeurs de ses sous expressions (exemple : *E1+E2*). L'ordre d'exécution a beaucoup moins d'importance qu'en programmation impérative. *E1* peut être évaluée avant, après ou pendant l'évaluation de *E2*)

Le modèle de calcul en programmation fonctionnelle est plus proche des mathématiques, ce que permet d'appliquer les mêmes techniques de preuves et de raisonnement.

La programmation fonctionnelle reste peu utilisée en dehors du milieu universitaire.

Le développement des langages fonctionnels est très limité comparé au développement des langages impératifs et des langages orientés objet.

Les langages fonctionnels souffrent encore d'une réputation de lenteur aujourd'hui complètement injustifiée. Les langages comme Objective Caml ou encore Haskell produisent des exécutables dont les performances moyennes sont comparables à celles des codes produits par les compilateurs C ou C++.

4. Le langage Scheme, un bref aperçu

Un langage fonctionnel est un langage de programmation dont la syntaxe et les caractéristiques encouragent la programmation fonctionnelle. Le langage fonctionnel le plus ancien est Lisp, créé en 1958 par McCarthy. Lisp a donné naissance à des variantes telles que Scheme (1975) et Common Lisp (1984) qui, comme Lisp, ne sont pas typées. Des langages

fonctionnels plus récents tels ML (1973), Haskell (1987), OCaml, Erlang, Clean et Oz, CDuce, ou Scala sont fortement typés.

Le langage Scheme est un dialecte de LISP conçu au MIT en 1975, principalement pour l'éducation. Initialement le langage était petit, mais aujourd'hui c'est un langage complet. Généralement Scheme est interprété, mais il peut aussi être compilé afin de produire des exécutables efficacement exécutés. La catégorie syntaxique principale dans Scheme c'est l'expression.

4.1Notions de base

Comme tout dialecte de Lisp, Scheme repose sur des règles syntaxiques simples: notation préfixée et entièrement parenthésée pour les programmes et les données. La fonction précède ses arguments, et le tout est encadré par un couple de parenthèses. La forme générale d'une expression est : **(<opération> <arguments>)** où :

<opération> : est un mot clé, une règles spéciales sinon un appel de fonction.

<arguments>: sont les paramètres actuels passés par valeur.

Le listing 6.5 contient quelques exemples d'expressions. La première expression calcule la somme de deux nombres. La seconde expression calcule le produit de cinq nombres. La troisième calcule le produit du nombre 73 par le nombre 78. Enfin, la dernière expression se réduit à un seul nombre. Sa valeur est 20.

```
(+ 2 3)
(* 2 3 5 7 8)
(* 73 78)
20
```

Listing 6.5 : Quelques exemples d'expressions

Tout programme Scheme est représenté par des expressions symboliques. Une expression symbolique est une structure arborescente de taille arbitraire, composée d'atomes. Les atomes sont les données élémentaires du langage. Les atomes se décomposent typiquement en un petit nombre de catégories :

- les *symboles*, ou *identificateurs* : un symbole est une suite de caractères ne représentant ni un nombre, ni une chaîne de caractères. La plupart des caractères du code ASCII sont admis à l'intérieur d'un identificateur. Voici quelques identificateurs : `symbol ? 2@5 ?66 get* set-car!`
- les *booléens*, représentent les valeurs logiques vrai et faux. Ils sont dénotés par les noms `#t et #f.`
- les *nombres* : suites de chiffres, débutant éventuellement par un signe + ou un signe -, pouvant comporter un point décimal ou un exposant. 2 -45 12.34 e-5

- les caractères ; un caractère s'écrit sous la forme d'un dièse, #, suivi d'un backslash, \, suivi du caractère ou du nom du caractère. Voici quelques caractères : `#\b  #\+  #\8`
- les chaînes de caractères ; une chaîne de caractères est une suite arbitraire de caractères placée entre doubles quotes ". Exemple : "Bonjour"

Le tableau 6.1 contient une liste non exhaustive des caractères spéciaux du langage Scheme.

Caractère	Rôle
Espace	sert de séparateur
Le point .	utilisé seul, le point sert de séparateur dans une paire pointée. Le point sert également à séparer la partie entière de la partie décimale d'un nombre. Il peut être utilisé à l'intérieur d'un identificateur.
Le point-virgule **;**	introduit un commentaire. Le restant de la ligne est ignoré, et le tout équivaut à un blanc.
La double quote "	introduit une chaîne de caractères.
Les parenthèses( )	servent à délimiter listes et paires pointées.
Le dièse #	est un macro-caractère. Il précède certaines constructions particulières du langage ; par exemple, #x2fc est une notation hexadécimale du nombre entier 764
L'apostrophe '	ou quote, introduit des données symboliques.
La contre apostrophe ‘	ou back-quote, joue un rôle voisin de celui de la quote.
les { } et les [ ]	ne sont pas utilisés dans le langage, mais qui sont « réservés » pour de futures extensions.

Tableau 6.1 Ponctuation en Scheme

4.2 La fonctions define

Pour nommer une valeur par un identificateur, on dispose de la forme : `(define <ident> <exp>)`. Exemples : `(define pi 3.14)` **ou** `(define rac2 (sqrt 2))`.

Si on veut nommer une expression contenant des paramètres, il s'agit d'une fonction: `(define (nom-fonction x1 x2 … xk) corps de la fonction )`. Par exemple : `(define (carre x (* x x))`

define crée la variable globale <ident> et l'initialise à la valeur de <expr>

4.3 La fonctions quote

La fonction quote ou ' est utilisé lorsqu'on ne veut pas évaluer immédiatement une expression. Par exemple :

```
(define x (+ 1 2))  ; liaison de x à 3

(define y '(+ 1 2)) ; liaison de y à (+ 1 2)

(quote (1 2 3))      ; (1 2 3)
```

4.4 Les conditions

Une condition est exprimée par `(if <expr1> <expr2> <expr3>)` dont l'évaluation se fait de la manière suivante : évaluer <expr1> : si le résultat n'est pas faux «retourner» la valeur de <expr2> ; sinon «retourner» la valeur de <expr3>.

Une autre manière pour exprimer les conditions consiste en utiliser la fonction `cond` dont la syntaxe est : *(**cond** <clause 1> <clause 2> … <clause n> <clause else>)* avec *<clause> = (condition <corps>).*

Les clauses sont évaluées dans l'ordre où elles apparaissent, de la façon suivante :

- évaluation de la condition ;
- si elle est fausse, on passe à la clause suivante (en l'absence de clause le résultat général du cond est faux) ;
- sinon évaluation en séquence du corps, composé d'une suite d'expression, le résultat étant celui de la dernière.
- Si aucune clause n'est vraie, on peut facultativement ajouter une clause else = (else <corps>), qui sera évalué en dernier recours.

4.5 Les Prédicats

Les prédicats sont des fonctions à valeurs booléennes. On donne ci-dessous la liste des plus courants :

`number? odd? real? even? integer? null? string? boolean? zero? exact?`

4.6 Les listes

Pour construire une liste, on peut utiliser la fonction `list`, ou tout simplement, en définissant une variable. Par exemples:

`(list 10 12 14 16)` ; répond `(10 12 14 16)`

`(list (> 10 12) (not #f))` ; répond #f #t.

`(define L '(1 2 3 4 5))` ; répond `(1 2 3 4 5)`

On dispose de trois fonctions de base pour la manipulation récursive des listes : `cons` , `car` , `cdr`.

La fonction **`cons`** permet la Construction d'une liste dont le premier élément est le résultat de l'évaluation de son premier argument, et dont la suite est son deuxième argument. Le deuxième argument doit être une liste. Par exemples :

```
(cons 10 (list 20 30)) ; (10 20 30)
(cons '( ) (1 2 3)); (1 2 3)
```

La fonction `car` renvoie le premier élément de son argument qui est une liste. Sa syntaxe est : `(car <liste>.` Par exemple :

`(car '(1 2 3))` donnera 1 en retour.

La fonction cdr Renvoie la liste constituée de son argument privée de son premier élément. Sa syntaxe est: (cdr <liste>). Par exemples :

```
(cdr '(1 2 3)) donnera (2 3) en retour
(cdr '(1)) donnera ( ) en retour
```

Il existe éventuellement d'autres fonctions pour la manipulation des listes, on a : la fonction null? ; `append ; reverse ; list-ref ; list-tail et member.`

La fonction null? retourne vrai si la liste est vide, sinon elle retourne faux. Par Exemple: `(null? '( )) ; #t`

La fonction append permet de construire une nouvelle liste en concaténant d'autres. Par exemple : `(append '(1 2 3) '(4 5 6))` produit `(1 2 3 4 5 6).`

La fonction reverse agit sur un argument liste est retourne une liste dont les éléments sont les même que son argument dans l'ordre inverse. Par exemple :

`(reverse '(1 2 3))` retourne `(3 2 1).`

La fonction list-ref : retourne le ième élément, à compter de zéro. Par exemple :

`(list-ref '(a b c d) 2)` retourne `c.`

La fonction list-tail : retire les i premiers éléments de la liste. Par exemple :

`(list-tail '(a b c d e) 3)` retourne `(d e)`

La fonction `member` vérifie si un élément fait partie d'une liste. Par exemple :

`(member 'd '(a b c))` retourne `#f`

4.7 La Récursivité

Sans affectation, on utilise la récursivité pour programmer des boucles. Le code ci-dessous présente une définition récursive de la factorielle :

```
(define (fact n)
(if (zero? n)
1
(* n (fact (- n 1)))))
```

5. Conclusion

En conclusion, il est important de signaler les empreintes des la programmation fonctionnelle principalement par rapport à la programmation impérative. La programmation fonctionnelle est une façon de penser qui peut se traduire même dans des langages hautement impératifs. Cependant, c'est plus facile avec un langage fonctionnel. La

programmation fonctionnelle n'encourage pas les effets de bord. La programmation fonctionnelle encourage la composabilité. L'impératif est plus proche du fonctionnement de la machine: une variable est une partie de la mémoire, sa valeur change au fur et à mesure du déroulement du programme. En fonctionnel on privilégie l'usage des valeurs ou de variables au sens mathématique. En impératif les étapes du calcul sont détaillées et l'ordre d'exécution est important. Par contre en fonctionnel on précise à la machine ce qu'est une opération et on la laisse décider comment la réaliser. C'est pour ce dernier point que la programmation fonctionnelle appartient à la classe de la programmation dite déclarative tout comme la programmation logique qui fera l'objet du prochain chapitre.

Chapitre VII :

La programmation logique.

1. Introduction

La programmation logique est une manière de programmer dans laquelle un programme n'est pas vu comme une application qui transforme des données en entrée en données en sorties comme en programmation impérative et fonctionnelle. En programmation logique, un programme implémente des relations de la forme *r(x,y)* entres des valeurs. Considérant deux ensembles de valeurs X et Y, *r* est une relation si pour tout *x* appartenant à X et pour tout *y* appartenant à Y, *r(x,y)* est soit vraie soit fausse.

La programmation logique et la programmation fonctionnelle sont souvent regroupées sous la rubrique « programmation déclarative». Idéalement, un programme déclaratif spécifie simplement le problème - ce que nous voulons - et l'ordinateur réalise le comment faire. Bien sûr, cela est une simplification excessive. Pour les langages déclaratifs qui existent actuellement, la description du problème est vraiment un programme destiné à tourner sur une machine abstraite. Un programme fonctionnel définit un système de règles de réécriture qui peut évaluer une fonction. Un programme de la logique définit un espace de recherche pour des problèmes de réduction qui peuvent résoudre toutes les instances d'un but visé. Un programme déclaratif exprime l'algorithme d'une manière plus abstraite que, disons, un programme Pascal ou C, mais les moyens d'expression sont restrictifs.

2. Concepts clés de la programmation logique

Pour comprendre la programmation logique, nous examinons d'abord la différence entre le calcul et la déduction. Pour calculer nous partons d'une expression donnée et, selon un ensemble de règles fixes (le programme) générer un résultat. Pour déduire nous partons d'une hypothèse et, selon un ensemble de règles fixes (des axiomes et des règles d'inférence),

essayer de construire une preuve d'une conjecture. Donc le calcul est mécanique et ne nécessite pas d'ingéniosité, tandis que la déduction est un processus créatif.

Unifier le calcul et la déduction est derrière le paradigme de programmation logique. Le calcul peut être considéré comme une forme limitée de déduction parce qu'il établit des théorèmes. Par exemple, 14 + 16 = 30 est à la fois le résultat d'un calcul, et d'un théorème d'arithmétique. Inversement, la déduction peut être considérée comme une forme de calcul dans laquelle nous fixons une stratégie de recherche de preuve.

Contrairement aux autres modèles computationnels, Un programme de logique se compose d'une série d'assertions écrites dans le langage formel de logique. Les résultats sont issus des programmes logiques par raisonnements symboliques. Les Systèmes de programmation logique résolvent les buts par chercher systématiquement un moyen pour obtenir la réponse du programme.

Un langage de programmation logique consiste en trois langages comme il est illustré dans la figure 7.1 :

- Un langage pour décrire les données : Ψ ;
- Un langage $\mathcal{F}$ pour décrire les formules logiques représentant les relations entre les objets de l'univers de discoure et/ou du problème à résoudre ;
- Un langage pour décrire les buts et les questions cherchées : $\mathbb{G}$.

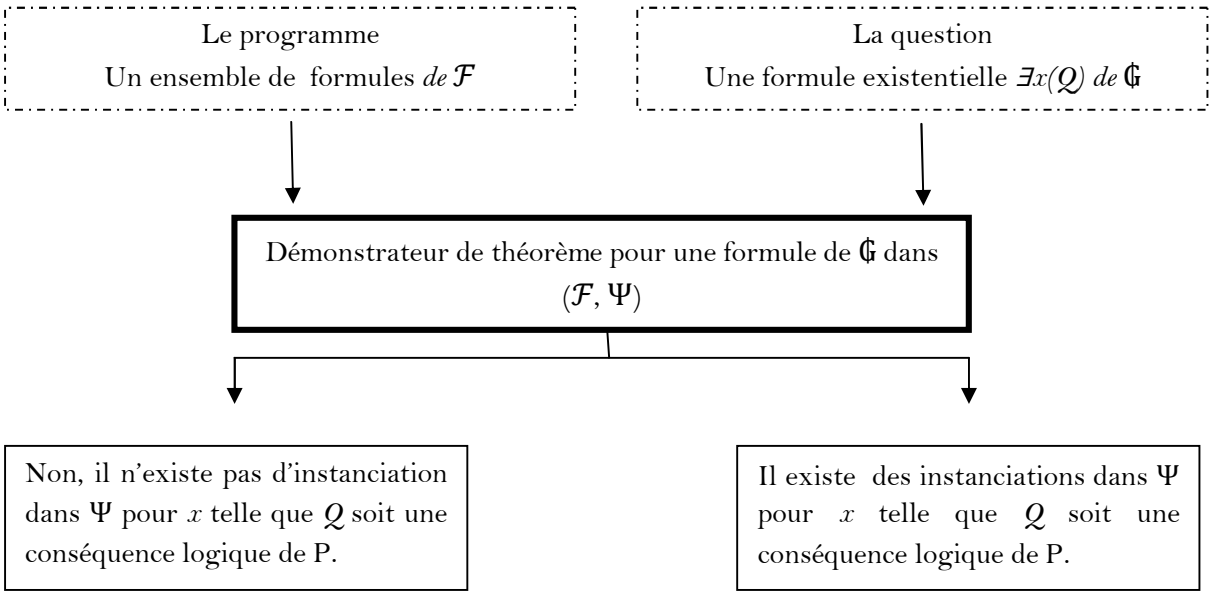


Figure 7.1 : *Schématisation d'un langage de programmation logique.*

2.1 La programmation logique en clauses de Horn

Une programmation logique en clause de Horn est obtenue en spécialisant les trois langages constituant un langage de programmation logique, de sorte que le langage des donnés Ψ est le langage des termes ; Le langage $\mathcal{F}$ est le langage des *clauses définies* ; et finalement, le langage ₲ est celui des conjonctions d'atomes.

Un *terme* est défini sur un ensemble de symboles de fonctions et de constantes de la manière suivante :

- si X est une variable alors X est un terme ;
- si c est une constante alors c est un terme ;
- si *f* est un symbole de fonction et *t1*, . . ., *tn* sont des termes alors *f(t1*,. . .,*tn)* est un terme.

Un *atome* est défini sur un ensemble de symboles de prédicats : si P est un symbole de prédicats et *t1*, . . ., *tn* sont des termes alors P*(t1*,. . .,*tn)* est un atome.

Une *clause définie* est constituée dans cet ordre : (1) d'une tête de clause (un atome) ; (2) du symbole " :-" ("si") ; (3) d'un corps de clauses, conjonction d'atomes séparés par le symbole "," ; (4) et finalement le symbole ".".

Par exemple, « *Amir a la rougeole* si il a *de la fièvre et des points rouges au fond de la gorge*» s'écrit comme :

avoir_rougeole(Amir) :- avoir_fièvre(Amir), avoir_pts_rouges(Amir).

La tête — Le corps

Si le corps est absent, une clause définie est un fait et est constituée : (1) d'une tête de clause (un atome) et (2) le symbole ".". Par exemple : « Amir est un homme », s'écrit de la forme: homme (Amir).

Un *programme logique* est un ensemble (une conjonction) fini de définitions de prédicats. Sachant qu'un prédicat est défini par l'ensemble fini de clauses définies ayant le même nom de prédicat dans la tête.

Une *variable logique* représente une donnée quelconque mais unique. Par exemple : Etudiant(X) signifie que l'entité X est un étudiant.

Une *substitution* des variables par les termes (un ensemble de couples Xi ← ti) est une fonction des variables dans les termes notée ⁅X1 ← t1, . . . ,Xn ← tn⁆ (Xi distincte de ti et Xi toutes distinctes entre-elles).

La substitution σ est étendue aux termes (et aux atomes) :

- si (X ← t) ∈ σ alors σ(X) = t sinon σ(X) = X ;
- σ(f (t1, . . . ,tn)) = f (σ(t1), . . . ,σ(tn)).

Le résultat de l'application d'une substitution σ (constituée de couples Xi ← ti) à un terme (ou un atome) t, nommée instance de t et notée σ(t), est le terme t dont toutes les occurrences des Xi ont été remplacées simultanément par les ti.

Dans une clause (ou un fait), Une variable logique exprime que la clause (ou le fait) est vrai pour n'importe quelle donnée. Par exemple : mortel(X) :- homme(X).

L'*unification* calcule une substitution qui rend des termes égaux. Soient E = {t1, . . . ,tn} un ensemble de termes et σ une substitution. σ unifie E si par définition σ(t1) = . . . = σ(tn) (σ est un unificateur de E et E est unifiable).

L'unificateur σ d'un ensemble de termes E est l'unificateur le plus général de E si quelque soit σ′ un unificateur de E il existe une substitution η telle que σ′ = ση.

2.2 Aspects Sémantiques

Les aspects sémantiques derrières la programmation logique, notamment la sémantique opérationnelle, sont les différents raisonnements symboliques sur de valeurs de vérités des énoncés logiques. Par le terme symbolique nous cherchons à exclure les techniques directes de calculs de valeurs de vérités des énoncés par les tables de vérités qui sont inefficaces dans le cas où les énoncés sur lesquels s'effectuent les raisonnements inclurent un nombre important de variables logiques. .

Les principaux mécanismes de raisonnements sur les clauses sont le calcul de résolvantes selon le principe de résolution de Robinson et la réfutation qui permet aussi de déterminer si un énoncé est la conséquence logique d'un autre ou d'une conjonction d'énoncés. En pratique la programmation logique et particulièrement le langage Prolog se base sur la méthode SLD-résolution (SLD : Sélection, Linéaire, Définie) qui est un algorithme servant à prouver une formule de logique du premier ordre à partir d'un ensemble de clauses de Horn. Elle est basée sur une résolution linéaire, avec une fonction de sélection sur les clauses définies.

Étant donné un ensemble de clauses de Horn (un programme logique) et une requête (une conjonction de littéraux positif), la SLD-résolution fonctionne globalement comme d'autres formes de résolution, en tentant d'unifier des clauses de Horn pour en créer de nouvelles, jusqu'à ce qu'une contradiction soit atteinte, ou qu'aucune nouvelle unification ne puisse être faite. Plus précisément, la SLD-résolution commence par construire la négation de la requête qui donne une clause de Horn négative, puis unifie le premier élément de cette négation avec une clause du programme. La clause en question est recherchée de manière déterministe, en recherchant la première clause du programme dont la tête est la contrepartie positive de la première négation de la liste de buts. Si une contradiction est

trouvée, alors la négation de la requête est inconsistante avec le programme, la requête termine donc avec succès. Si aucune unification ne peut être trouvée pour le premier terme négatif de la liste de buts, alors la requête échoue. Si une unification sans contradiction existe, alors la nouvelle liste de buts devient la conjonction de l'ancienne avec la clause sélectionnée, et la SLD-résolution est relancée (de manière récursive) sur la nouvelle liste de buts. Si une sous-requête échoue, alors l'unification de niveau supérieur dont elle provient échoue, et l'algorithme cherche une autre unification (dans la suite du programme) pour le premier terme de la liste de buts. Cette étape s'appelle le retour sur trace.

Selon la terminaison de l'algorithme de la SLD-résolution, on a les propriétés suivantes :

- Pour une dérivation SLD d'une résolvante initiale , la question, est une suite finie ou infinie ≥0 de résolvantes tel que : ——— ,
- Un succès ou réfutation SLD d'une résolvante est une dérivation finie $0 \leq \leq$, tel que la dernière résolvante est vide.
- Une dérivation telle que la dernière résolvante ne peut plus inférer de nouvelle résolvante est une dérivation échec.

Pour bien illustrer le fonctionnement du principe de la SLD-révolution, examinons l'exemple connu sous le nom du programme de la mort de Socrate figurant dans 7.2 et 7.3. Le programme de la mort de Socrate est constitué des clauses suivantes :

animal(X) :- homme(X).

mortel(X) :- animal(X).

meurt(X) :- mortel(X), empoisonne(X).

empoisonne(X) :- boit(X,Y), poison(Y).

homme(socrate).

homme(platon).

boit(socrate, cigue).

poison(cigue).

$$\frac{\overbrace{meurt(X)}^{1}}{mortel(X_1), empoisonne(X_1)} \quad C_0,1\in\{1\}$$

avec $C_0 = meurt(X) : -mortel(X), empoisonne(X).$
et $\theta_0 = [X \leftarrow X_1]$ et $\theta_0(meurt(X)) = meurt(X_1)$
et donc $\sigma_0 = upg(meurt(X_1), meurt(X)) = [X \leftarrow X_1]$
$\sigma_0(\theta_0(mortel(X)), \theta_0(empoisonne(X))) = mortel(X_1), empoisonne(X_1).$

$$\frac{\overbrace{mortel(X_1)}^{1}, \overbrace{empoisonne(X_1)}^{2}}{mortel(X_2), boit(X_2, Y_2), poison(Y_2)} \quad C_1,2\in\{1,2\}$$

avec $C_1 = empoisonne(X) : -boit(X, Y), poison(Y).$
$\theta_1 = [X \leftarrow X_2][Y \leftarrow Y_2]$ et $\theta_1(empoisonne(X)) = empoisonne(X_2)$
et donc $\sigma_1 = upg(empoisonne(X_2), empoisonne(X_1)) = [X_1 \leftarrow X_2]$
$\sigma_1(mortel(X_1), \theta_1(boit(X, Y)), \theta_1(poison(Y))) =$
$mortel(X_2), boit(X_2, Y_2), poison(Y_2).$

$$\frac{\overbrace{mortel(X_2)}^{1}, \overbrace{boit(X_2, Y_2)}^{2}, \overbrace{poison(Y_2)}^{3}}{animal(X_3), boit(X_3, Y_2), poison(Y_2)} \quad C_2,1\in\{1,2,3\}$$

avec $C_2 = mortel(X) : -animal(X).$
et $\theta_2 = [X \leftarrow X_3]$ et $\theta_2(mortel(X)) = mortel(X_3)$
et donc $\sigma_2 = upg(mortel(X_3), mortel(X_2)) = [X_2 \leftarrow X_3]$
$\sigma_2(\theta_2(animal(X)), boit(X_2, Y_2), poison(Y_2)) =$
$animal(X_3), boit(X_3, Y_2), poison(Y_2).$

$$\frac{\overbrace{animal(X_3)}^{1}, \overbrace{boit(X_3, Y_2)}^{2}, \overbrace{poison(Y_2)}^{3}}{animal(X_3), boit(X_3, cigue)} \quad C_3,3\in\{1,2,3\}$$

avec $C_3 = poison(cigue).$
et $\theta_3 = \epsilon$
et donc $\sigma_3 = upg(poison(cigue), poison(Y_2)) = [Y_2 \leftarrow cigue]$
$\sigma_3(animal(X_3), boit(X_3, Y_2)) = animal(X_3), boit(X_3, cigue).$

$$\frac{\overbrace{animal(X_3)}^{1}, \overbrace{boit(X_3, cigue)}^{2}}{homme(X_5), boit(X_5, cigue)} \quad C_4,1\in\{1,2\}$$

avec $C_4 = animal(X) : -homme(X).$
et $\theta_4 = [X \leftarrow X_5]$ et $\theta_4(animal(X)) = animal(X_5)$
et donc $\sigma_4 = upg(animal(X_5), animal(X_3)) = [X_3 \leftarrow X_5]$
$\sigma_4(\theta_4(homme(X)), boit(X_3, cigue)) = homme(X_5), boit(X_5, cigue).$

$$\frac{\overbrace{homme(X_5)}^{1}, \overbrace{boit(X_5, cigue)}^{2}}{boit(socrate, cigue)} \quad C_5,1\in\{1,2\}$$

avec $C_5 = homme(socrate).$
dans $\{homme(platon)., homme(socrate).\}$
et $\theta_5 = \epsilon$
et donc $\sigma_5 = upg(homme(socrate), homme(X_5)) = [X_5 \leftarrow socrate]$
$\sigma_5(boit(X_5, cigue)) = boit(socrate, cigue).$

$$\frac{\overbrace{boit(socrate, cigue)}^{1}}{} \quad C_6,1\in\{1\}$$

avec $C_6 = boit(socrate, cigue).$
et $\theta_6 = \epsilon$
et donc $\sigma_6 = upg(boit(socrate, cigue), boit(socrate, cigue)) = \epsilon.$

Figure 7.2 : *Exemple d'application de la SLD-résolution*

La substitution ... | ( ) est une substitution calculée d'une réfutation SLD. Et ... | ( ) (R0) est la réponse calculée de la réfutation SLD.

$$\sigma_0 \ldots \sigma_6 = [X \leftarrow socrate][Y_2 \leftarrow cigue],$$

$$(\sigma_0 \ldots \sigma_6)|_{V(meurt(X))} = ([X \leftarrow socrate][Y_2 \leftarrow cigue])|_{\{X\}} = [X \leftarrow socrate]$$

et

$$(\sigma_0 \ldots \sigma_6)|_{V(meurt(X))}(meurt(X)) = [X \leftarrow socrate](meurt(X)) = meurt(socrate)$$

Figure 7.3 : *Exemple d'application de la SLD-résolution*

Ainsi, *meurt(Socrate)* et *meurt(X)* sont des conséquences logiques des clauses du programme.

2.3 Stratégie de sélection et stratégie de recherche

A chaque fois qu'un calcul d'une résolvante (dérivation) doit se faire, plusieurs possibilités se présentent. *Une stratégie de sélection* choisit, dans une dérivation, l'atome à réduire. Si nous représentons une dérivation sous forme d'un arbre dit arbre de preuve, comme illustre la figure 7.4, la stratégie de sélection correspond à un parcours de cet arbre.

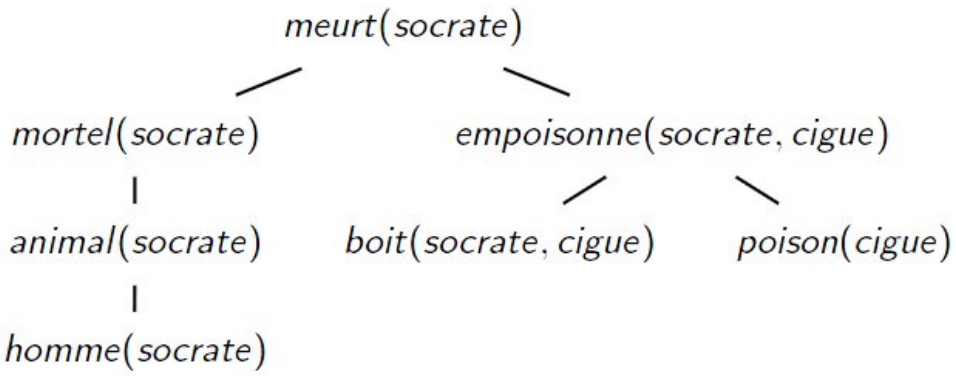


Figure 7.4 : *Arbre de preuve*

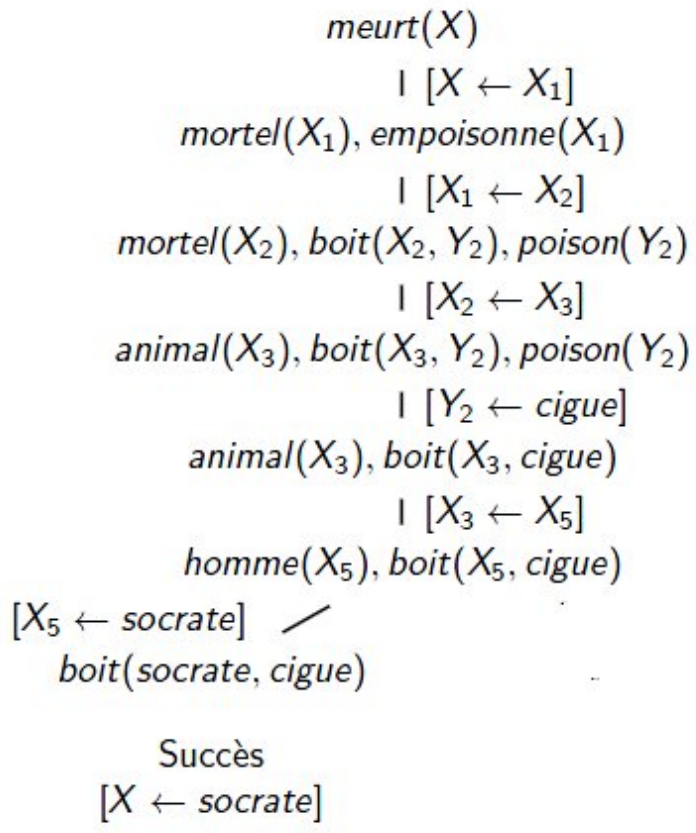


Figure 7.5 : *Arbre SLD*

Une stratégie de recherche choisit la clause pour réduire l'atome considéré. Pour une stratégie de sélection donnée, l'arbre SLD (figure 7.5) explicite toutes les dérivations possibles pour chaque but cherché.

La stratégie de recherche correspond au parcours d'une branche de l'arbre SLD. La stratégie de sélection et la stratégie de recherche sont des mécanismes de contrôle de la dérivation. Une stratégie de recherche est complète si tous les choix de clauses pour résoudre un but sont explorés après un nombre finis d'´etapes.

2.4 La négation par l'échec

La négation par l'échec (en anglais NAF pour negation as failure, ou NBF pour negation by failure) est une règle d'inférence non monotone en programmation logique, utilisée pour la dérivation de ~F à partir de l'échec de la dérivation de F. c'est-à-dire, pour démontrer not F (la négation d'une formule F) on va tenter de démontrer F. Si la démonstration de F échoue, on considère que not(F) est démontrée.

Le langage Prolog pratique la négation par l'échec. Pour Prolog, not(F) signifie que la formule F n'est pas démontrable, et non pas une formule fausse. C'est ce que l'on appelle l'hypothèse du monde clos. La négation par l'échec est une forme faible de négation logique.

3. Le langage Prolog

Le langage de programmation Prolog est l'un des principaux langages de programmation logique. Le nom Prolog est un acronyme de PROgrammation en LOGique. Le but était de créer un langage de programmation où seraient définies les règles logiques attendues d'une solution et de laisser le compilateur la transformer en séquence d'instructions.

Prolog permet la mise en œuvre d'un sous ensemble de la programmation logique en clause de Horn avec les propriétés suivantes :

- ♦ Prolog utilise une stratégie de recherche en profondeur d'abord ;
- ♦ Prolog utilise la stratégie 'le plus à gauche' comme stratégie de sélection ;
- ♦ Prolog utilise l'unification sans test d'occurrence ;
- ♦ Prolog pratique la négation par l'échec sous hypothèse du monde clos ;
- ♦ Prolog propose des prédicats arithmétiques prédéfinis ; des prédicats métalogiques sur les types ; des prédicats ensemblistes et d'ordre supérieur ;

Les données manipulées en Prolog sont des objets élémentaires qui peuvent être des variables ou des constantes. Les variables sont des combinaisons de lettres, chiffres et du caractère "_". Avec la contrainte sur le premier caractère qui doit être une lettre alphabétique majuscule ou '_'. Les constantes peuvent être des nombres, des symboles (chaînes de caractères dont la première lettre est en minuscule, ou une chaîne de caractères quelconque entre les caractères ".

La programmation logique sous Prolog est une forme de programmation qui définit les applications à l'aide d'un ensemble de *faits* élémentaires et de *règles logiques*. Ces faits et ces règles sont exploités par un démonstrateur de théorème ou moteur d'inférence, en réaction à une question ou une requête.

La structure générale d'un programme Prolog est illustrée dans le listing 7.1. Le listing 7.2 contient un petit exemple d'un programme en Prolog où les symboles « :- » , « ; » et « , » signifient respectivement « si », « ou » et « et ». .

```
Domains
        Construction de nouveaux types
Predicates
        Définitions des prédicats
Clauses
        Faits et règles
Goal
        But
```

Listing 7.1 Structure d'un programme Prolog

```
Predicates
        Parent(symbol, symbol)
        Frere(symbol, symbol)
        Ascendant(symbol, symbol)
Clauses
        Parent(a, b).
        Parent(b, c) .
        Parent(b, d).
        Parent(d, e).
        Parent(f, d).
Frere(X, Y) :- Parent(Z, X), Parent(Z, Y), Not(X=Y).
Ascendant(X,Y)  :- Parent(X, Y).
Ascendant(X,Y)  :- Parent(X, Z), Ascendant(Z,Y).

/* On peut aussi écrire ascendant comme suit :
Ascendant(X,Y)  :- Parent(X, Y);
Parent(X, Z),Ascendant(Z, Y).

*/
```

Listing 7.2 Exemple d'un programme Prolog

Deux sortes de requêtes peuvent être utilisées pour interagir avec un programme Prolog. La première étant les questions dont les réponses peuvent être *oui* ou *non*. La seconde regroupe les questions dont les réponses sont des valeurs vérifiant certaines relations logiques. Le listing 7.3 contient quelques requêtes relatives au programme du listing 7.2 ainsi que leurs réponses.

```
➢  Question : frere(d, c)
   Réponse   Yes
➢  Question : frere(X, d)
   Réponse   X = c
➢  Question : frere(d, X)
   Réponse   X = c
➢  Question : Ascendant(X, e)
   Réponse
   X = d
   X = a
   X = b
   X = f
```

Listing 7.3 Exemple de requêtes.

3.1 La structure de liste

Les listes sont des structures de données de base en Prolog. Une liste est une structure binaire récursive dont la syntaxe est la suivante :

```
[ ] : est la liste vide

[ Tête | Queue ] : est la liste où le premier élément est Tête
et le reste de la liste est Queue.

[ a,b,,7,'toto' ] : est une liste de constantes.

Par exemples :

[X|L] = [a,b,c] donne YES {X=a,L=[b,c]}
[X|L] = [a] donne YES {X=a,L=[]}
[X|L] = [] donne NO
[X,L] = [a,b,c] donne NO
[X,Y|L] = [a,b,c] donne YES {X=a,Y=b,L=[c]}
[X|L] = [A,B,C] donne YES {X=A, L=[B,C]}

Il est possible d'imbriquer les listes, par exemples :

[X|L] = [a,[]]. donne YES {X=a, L=[[]]}
[X|L] = [a,[b,c],d]. donne YES {X=a, L=[[b,c],d]}
[X] = [a|[]]. donne YES {X=a}
[X,Y] = [a|[b|[]]]. donne YES {X=a, Y=b}
[X,Y] = [a|[b]]. donne YES {X=a, Y=b}
[X,a] = [b,Y]. donne YES {X=b, Y=a}
```

Listing 7.4 Syntaxe de la structure liste.

Prolog propose des prédicats prédéfinis pour la manipulation des listes. Le tableau 7.1 regroupe quelques uns en proposant une brève présentation pour chaque prédicat.

3.2 Tests sur les types

Prolog offre la possibilité de tester les types des termes :

- var(?term) réussit si le paramètre est non instancié
- nonvar(?term) réussit si le paramètre est instancié
- atom(?term) réussit si le paramètre est instancié par un atome
- integer(?term) réussit si le paramètre est un entier
- float(?term) réussit si le paramètre est un réel
- number(?term) réussit si le paramètre est un entier ou un réel
- atomic(?term) réussit si le paramètre est un nombre ou un atome
- compound(?term) réussit si le paramètre est un terme composé (liste non vide ou structure)
- callable(?term) réussit si le paramètre est un atome ou un terme composé

- list(?term) réussit si le paramètre est une liste

Prédicat	Rôle	Utilisations

length	Calcul de la longueur d'une liste	`length(+Liste, +Int)`
		`_____`
		`?- length([1,2,3,4,5], Long).`
		`Long = 5`
		`_____`
		`?- length(L, 3).`
		`L = [_G316, _G319, _G322]`
		`Yes`

member	Appartenance à une liste	member(?term,?list) réussit si le premier argument est membre de la liste passée en deuxième argument.

append	Concaténation de listes	`append(?List1, ?List2, ?List3).`
		`Réussit si List3 est la concaténation de List1 et List2.`
		`?- append([1,2,3], [4,5,6], L3).`
		`L3 = [1, 2, 3, 4, 5, 6]`
		`ajouter le contenu de List2 à la suite de List1 et unifier le résultat dans List3.`

reverse	Inversion d'une liste	`| ?- reverse([1,2,3,4,5], L2).`
		`L2 = [5, 4, 3, 2, 1]`
		`Inverse l'ordre des éléments de List1 et unifie le résultat dans List2.`

last	Dernier élément d'une liste	`last(?List, ?Elem).` `Unifie Elem avec le dernier élément de la liste List.`
La syntaxe [X|Y]	Récupérer le premier élément d'une liste	

Tableau 7.1 Quelque prédicat pour manipuler les listes

3.3 Arithmétiques

Une expression arithmétique est constituée par des opérateurs et des nombres (constantes ou variables instanciées). Les opérateurs sont les opérateurs classiques : -, +, *, /, **, //, rem, mod, abs, min, max, sign, sqrt, sin, cos, tan, log, exp, ...

Les opérateurs binaires (sauf min et max) peuvent être utilisés en notation préfixée, +(2,3), ou infixée, 2+3.

Les opérateurs ne sont pas des prédicats mais des fonctions!

Une expression arithmétique est évaluée si elle apparait comme argument d'un prédicat qui porte sur des nombres. L'évaluation est forcée par le prédicat « is (?var,+evaluable) ». Par exemples : R = 2+3. donne YES {R = 2+3 ; R is +(2,3). donne YES {R = 5} ; is(T,4 mod 3) donne YES {T = 1}.

4. Conclusion

La programmation logique est une forme de programmation qui définit les applications à l'aide d'un ensemble de faits élémentaires et de règles logiques. Ces faits et ces règles sont exploités par un démonstrateur de théorème ou moteur d'inférence, en réaction à une question ou requête. Cette approche se révèle beaucoup plus souple que la définition d'une succession d'instructions que l'ordinateur exécuterait.

La programmation logique est particulièrement adaptée aux besoins de l'intelligence artificielle, dont elle est un des principaux outils.

La programmation logique est un paradigme particulier, ou les calculs se font sous forme de preuves logiques. Pour résoudre un problème, un programmeur ne décrit pas à la machine ce qu'elle doit faire, comme en programmation impérative, il ne fait que lui décrire le problème sous forme logique. La programmation revient à spécifier les propriétés du résultat du programme et non pas le processus pour arriver à ce résultat.

Chapitre VIII :

Tendances multi-paradigme

1. Introduction

Arrivant à ce point du cours, nous pouvons faire le constat qu'il existe aujourd'hui plusieurs paradigmes différents de programmation. Chacun de ces paradigmes a ces points forts et son lot d'inconvénients. Nous pouvons légitimement se poser les questions sur les possibilités de fertilités croisées entre différents paradigmes et le potentiel de combiner des concepts issus de paradigmes différents pour construire des langages supportant les concepts de plus d'un seul paradigme.

Est-il toujours possible de classer un langage de programmation dans tel ou tel paradigme ? Est ce que le fait d'avoir tous les éléments techniques qui caractérisent un langage fonctionnel par exemple (fonctions valeurs de première classe, …) fait d'un langage un langage fonctionnel, indépendamment de son "style".

Dans chaque paradigme de programmation, rares sont les langages que nous pouvons qualifier de *''purs ''*. C'est-à-dire, des langages qui supportent d'une manière stricte que les concepts clés du paradigme auquel ils appartiennent. Par exemple, les langages fonctionnels purs comme HASKELL n'autorisent pas l'opération l'affectation. Néanmoins, pour certaines raisons les langages fonctionnels Lisp, Schem et la famille des langages ML sont impurs puisque ces langages permettent de faire des affections d'où il est possible qu'il ait effets de bord.

2. Langages impurs et langages multi-paradigme

Considérons un langage de programmation *lg* appartenant un l'un des paradigmes de programmation que l'on note *P*. $P \in$ {*paradigme impératif, paradigme orienté objet, paradigme fonctionnel, …*}.

Le langage *lg* peut être qualifié de langage *pur*, si la syntaxe et les caractéristiques de ce dernier permettent la mise en œuvre et la manipulation des concepts de la programmation du paradigme *P exclusivement* et sans aucune violation des principes de base du paradigme *P*. dans le cas contraire, le langage est qualifié *d'impur*. Par exemple, le langage *C++* est un langage orienté objet *impur* car le principe stipulant que les attributs d'un objet ne peuvent pas être ni inspectés ni modifié que par les méthodes de l'objet n'est pas respecté. *C++* permet de déclarer certains attributs des objets comme étant *publiques* d'où ces attributs sont accessibles par tout le programme et non pas seulement les méthodes figurant dans la même calasses que ces attributs.

Le langage *lg* peut être qualifié de langage *multi-paradigme*, si la syntaxe et les caractéristiques de ce dernier permettent la mise en œuvre et la manipulation des concepts de la programmation de deux paradigmes distincts P_1 et P_2 ou plus. Par exemple, le langage *OCaml* étend les possibilités du langage fonctionnel *Caml* en permettant la programmation orientée objet et la programmation modulaire. Pour toutes ces raisons, *OCaml* entre dans la catégorie des langages multi-paradigme.

2.1 Pureté vs impureté en programmation fonctionnelle

La programmation fonctionnelle est restée presqu'exclusivement académique depuis son apparition. Elle commence à connaître un essor réel. Mais elle bouleverse complètement les habitudes de la programmation impérative et celle orientée objet et demande ainsi un effort certain d'apprentissage pour être efficacement utilisée.

La programmation fonctionnelle refuse toute notion de variable non constante et d'effets de bord : les seuls objets manipulés sont des fonctions ou des constantes : l'opération d'assignation est inexistante. Les fonctions peuvent être passées en argument à d'autres fonctions, ou même être retournées comme valeur. Cela donne souvent une formulation très compacte et générique d'algorithmes qui seraient fastidieux à exprimer en programmation impérative.

Une des implications importante de ce genre de fonctions, dites pures est que l'absence d'effet de bord fait que leur exécution ne dépend aucunement de leur environnement d'exécution. Cela justifie en partie tout l'intérêt qu'on prête à ce genre de paradigme de programmation. Cependant, Cette pureté élégante et théoriquement très satisfaisante, permet elle de faire effectivement quelque chose sans boucles ni variables ? C'est le paradoxe de la programmation fonctionnelle pure.

Pour des raisons d'efficacité et d'autres raison liées à la gestion des entrées-sorties, plusieurs langages fonctionnels sont impurs. Ils permettent, lorsque le programmeur le décide, d'utiliser la notion d'état avec tous les risques liés aux effets de bord que cela comporte. Lisp, Scheme, ML sont des langages fonctionnels impurs. Alors que Hasckell et Miranda sont qualifiés de purs.

2.2 Pureté vs impureté en programmation orientée objet

L'utilisation d'un langage *orienté objet* ne garantit pas une programmation orientée objet pure. L'approche globale que permet la programmation orientée objet est souvent imbriquée avec l'approche séquentielle permise par le langage de programmation sous-jacent. L'approche globale permet de fixer des repères où le fil séquentiel prend corps. Mais on peut tout aussi bien faire de la programmation objet "pure" dont l'aspect algorithmique est écarté ou tout au moins complètement transparent pour l'utilisateur.

Par exemple, en Java les unités logiques sont les classes. Java est un langage oriente presque pur. Il est seulement presque pur car (i) il a quand même des types élémentaires (primitifs) qui ne sont pas des objets instances de classes. (ii) on peut créer des méthodes de classes statiques (static) qui sont utilisables sans instancier d'objet (iii) la possibilité de définir des attributs protégés accessibles aux classes dérivées et aussi aux classes du même paquetage.

En *C++*, on peut écrire des programmes sans l'obligation de les structurer en classes ; On a la possibilité de déclarer des attributs d'une classe comme publiques et les rendre ainsi accessibles aux autres classes du programme. *C++* autorise les données et les fonctions membres statiques. En effet, les données n'appartiennent plus aux objets de la classe, mais à la classe elle-même, et il n'est plus nécessaire de connaître l'objet auquel le pointeur s'applique pour les utiliser. De même, les fonctions membres statiques ne reçoivent pas le pointeur sur l'objet, et on peut donc les appeler sans référencer ce dernier. *C++* autorise la déclaration de fonctions et de variables qui n'appartiennent à aucune classe. Et qui sont accessibles dans n'importe quelle autre fonction. Il est clair que pour ces raisons ainsi que d'autres, le langage *C++* n'est pas un langage objet pur. Cette impureté, trouve ses origines du fait que *C++* est le produit d'une extension du langage impératif *C*. Il faut savoir que la couche objet n'est pas un simple ajout au langage *C*, c'est une véritable extension. En effet, les notions qu'elle a apportées ont été intégrées au C à tel point que le typage des données de *C* a fusionné avec la notion de classe. Ainsi, les types prédéfinis *char*, *int*, *double*,… etc représentent à présent l'ensemble des propriétés des variables ayant ce type. Ces propriétés constituent la classe de ces variables, et elles sont accessibles par les opérateurs. Par exemple, l'addition est une opération pouvant porter sur des entiers (entre autres) qui renvoie un objet de la classe entier. Par conséquent, les types de base se manipulent exactement comme des objets. Du point de vue du *C++*, les utiliser revient déjà à faire de la programmation orientée objet.

2.3 Langages multi-paradigme

La programmation orientée objet est le paradigme de programmation le plus courant actuellement. D'autres paradigmes assez différents ont été amenés à maturité au cours des dernières années. Aucun des paradigmes de programmation existants n'est adapté pour résoudre de manière optimale tous les types de problèmes, bien que chacun ait son domaine d'action où il peut faire valoir ses atouts. Toutefois, l'application simultanée de plusieurs

paradigmes semble appropriée dans certaines situations. En conséquence, il existe depuis, des langages de programmation multi-paradigme.

La tendance multi-paradigme dans les langages de programmation peut avoir deux formes différentes pour combiner dans un même langage de programmation plusieurs paradigmes différents.

La première forme de combiner plusieurs paradigmes consiste à augmenter un langage donné appartenant à un paradigme donné par une couche de concepts relatifs à d'autres paradigmes de programmation permettant ainsi de supporter plusieurs paradigmes de programmation dans le même langage. Comme exemples, nous pouvons citer le langage Pascal Objet, OCaml, Objective C, et ADA 95.

Le langage Pascal Objet est un langage orienté objet dérivé du Pascal. Il a été créé en 1990 par la société Borland comme une amélioration du Turbo Pascal. Il s'agissait alors d'ajout de l'objet au Turbo Pascal. Le Pascal Objet prend un nouvel essor en 1995 avec la sortie du Delphi, toujours à l'initiative de Borland.

OCaml, anciennement connu sous le nom d'Objective Caml, est l'implémentation la plus avancée du langage de programmation Caml. OCaml est le successeur de Caml Light, auquel il a ajouté entre autres une couche de programmation objet. Caml (prononcé camel, signifie Categorical Abstract Machine Language) est un langage qui favorise initialement un style fonctionnel impur.

L'Objective-C est un langage de programmation orienté objet réflexif. C'est une extension du C ANSI. C'est une augmentation du C par deux couches différentes. La première couche étant orientée objet différente de celle qui a augmenté le *C* pour produire du C++, la différence réside dans : sa la distribution dynamique des messages, son typage faible ou fort, son typage dynamique ainsi qu'elle ne permet pas l'héritage multiple contrairement au C++. La seconde couche rend de ce langage un langage réflexif. La réflexion est la capacité d'un programme à examiner, et éventuellement à modifier, ses structures internes de haut niveau lors de son exécution. On distingue deux techniques utilisées par les systèmes réflexifs : *l'introspection*, qui présente la capacité d'un programme à examiner son propre état ; et *l'intercession*, qui est la capacité d'un programme à modifier son propre état d'exécution ou d'altérer sa propre interprétation ou signification.

Le langage de programmation ADA a été conçu dans les années 1970 comme un langage de programmation impératif et concurrent, adapté en particulier pour la mise en œuvre des systèmes à grande échelle et les systèmes embarqués. ADA a été étendu dans les années 1990, principalement pour soutenir la programmation orientée objet. On utilise généralement les appellations ADA 83 et ADA 95 lorsqu'il est nécessaire de distinguer entre les deux versions du langage.

Dans cette première forme de combinaison, généralement, un langage de programmation ayant connu un grand succès et une large utilisation comme Pascal ou C sert à la base pour construire un nouveau langage. L'objectif est de rendre l'apprentissage des nouveaux langages simple et souple pour les grandes communautés de programmeurs familiers et hautement qualifiés dans les langages servants de base.

La deuxième forme de combiner plusieurs paradigmes dans un seul langage de programmation consiste à construire à partir du zéro ce dernier prenant comme préoccupation de départ que le langage à concevoir doit supporter les différents concepts relatifs aux différents paradigmes à combiner. Par exemple, le langage de programmation Scala a été construit pour supporter à la fois la programmation fonctionnelle et celle orientée objet.

Le langage Scala a été conçu à l'école polytechnique fédérale de Lausanne pour exprimer les modèles de programmation courants dans une forme concise et élégante. Son nom vient de l'anglais *Scalable language* qui signifie à peu près « *langage adaptable* » ou « *langage qui peut être mis à l'échelle* ». Il peut en effet être vu comme un métalangage. Scala intègre les paradigmes de programmation orientée objet et de programmation fonctionnelle, avec un typage statique. Il concilie ainsi ces deux paradigmes habituellement opposés et offre au développeur la possibilité de choisir le paradigme le plus approprié à son problème. Les développeurs habitués à un seul paradigme (par exemple ceux ayant utilisé principalement Java qui repose sur la programmation orientée objet) peuvent trouver ce langage déroutant et difficile car il nécessite l'apprentissage de concepts différents si on veut pouvoir exploiter tout son potentiel. Néanmoins, il est tout à fait possible de l'utiliser dans un premier temps comme remplaçant de Java, en profitant alors de sa syntaxe épurée, puis d'utiliser les différents « nouveaux » concepts au fur et à mesure de leur apprentissage.

3. Conclusion

La tendance de supporter par un même langage de programmation différents concepts relatifs à différents paradigmes de programmation a donné naissance à plusieurs langages de programmation impurs ou multi-paradigme. Cela est à l'origine de plusieurs avantages comme la faciliter d'apprendre de nouveaux langages en explantant les expériences d'utilisation des langages leurs servants de bases. Néanmoins, Dans les langages de programmation conçus par combinaisons de plus d'un paradigme, le risque d'une prolifération de fonctions mal intégrées est important en tentant de satisfaire les exigences contradictoires imposées par les différents paradigmes combinés. Un Contrôle ferme doit être assuré lors de la construction de tels langages à fin d'assurer que ces deniers puissent largement éviter ce sort.

Exercices

Chapitre I

Exercice 1 : Ecrire une fonction dont le résultat produit dépend du type de passage de ses paramètres par valeur ou par variable.

Exercice 2 : Les langages de programmation qui n'autorisent pas la récursivité comme les premières versions du Fortran, ont-ils besoin d'une pile centrale pour gérer les enregistrements d'activation des procédures ?

Exercice 3 : Si deux types T1 et T2 sont équivalents par nom, T1 et T2 sont ils équivalents structurellement ?

Exercice 4 : Les types First et Second sont ils structurellement équivalents dans le code suivant :

```
type
First =  record
A : integer;
B : record
B1, B2 : integer;
end;
end;

Second =  record
A: record
A1, A2 : integer;
end;
B : integer;
end;
```

Exercice 5 :

1) Les types COULEUR et COLOR sont ils équivalents dans le code suivant :

```
type COLOR is (RED, WHITE, YELLOW, GREEN, BLUE,
BROWN, BLACK);
type COULEUR is (WHITE, RED, YELLOW, GREEN, BLUE,
BROWN, BLACK);
```

2) L'équivalence structurelle se rapporte à des formulations dans lesquelles une certaine forme de règles d'équivalence est définie entre les types sur la base de leurs propriétés structurelles. Dans le cas des d'énumération plusieurs degrés d'équivalence structurelle sont envisageables. Quel sont ces degrés ?

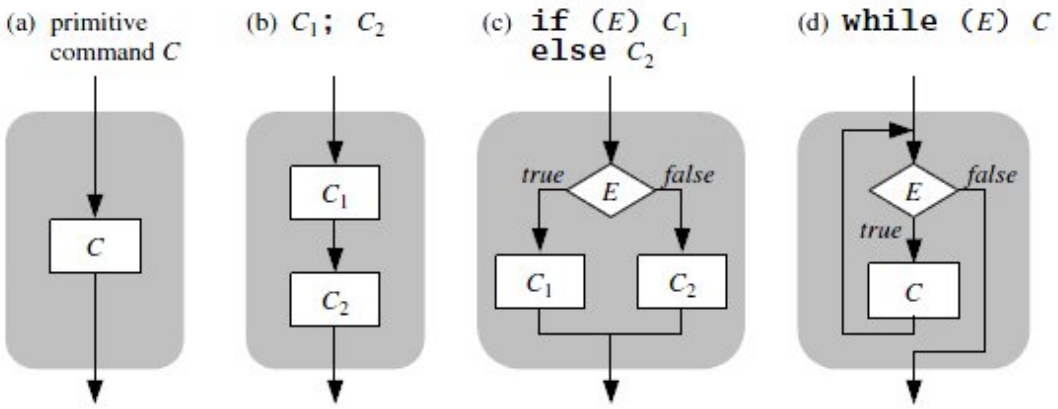
Exercice 6 : Vérifier si le code suivant provoque des erreurs de type lors de sa compilation.

```
1  variable

2            Premier : pointer to integer ;
3            Second : array  0..9  of
4                              Record
5                              Troisieme:  character;
6                              Quatrieme:  integer;
7                              Dernier : (Sam, dim, lun, mar, mer, jeu, ven)
8                              end;

9    begin
10   Premier := nil;
11   Premier := &Second[1].Quatrieme;
12   Premierˆ := 4;
13   Second[3].Quatrieme:=(Premierˆ+Second[1].Troisieme)* Second[Premierˆ].Quatrieme;
14   Second[0] := [Troisieme : 'x'; Quatrieme : 0; Deriner : jeu];
15   end;
```

Exercice 7 : Soit le fragment du code suivant écrit en C, C++ ou JAVA :

```
p = 1; m = n; a = b;
while (m > 0) {
if (m % 2 != 0) p *= a;
m /= 2;
a *= a;
}
```

1) Donnez un organigramme (diagramme) équivalent à ce code dans lequel faites une distinction claire entre les différents sous diagrammes correspondant aux : bloc du While et celui du IF. (La figure ci-dessous illustre comment schématiser les différentes formes d'instructions).



2) Réécrire ce code entièrement avec des branchements (conditionnels et\ou non conditionnels), puis expliquer pour quoi ce code est difficile à lire.

Exercice 8 : Faites une évaluation du langage Pascal selon les critères d'évaluation vus dans le cours.

Chapitre II

Exercice 1

Citer les entités que nous pouvons faire passer comme arguments de procédures ou de fonctions en Pascal. Quelles sont les entités que nous ne pouvons pas les utiliser à cet égard, pourquoi ?

Exercice 2

Soit la procédure ADA suivante :

```
procedure multiply (m, n: in out Integer) is
begin
m := m * n;
put(m); put(n);
end;
```

1) Supposons que i contient la valeur 2 et que j contient la valeur 3. Quels sont les résultats produits par les appels *multiply(i, j); multiply(i, i);*

2) Supposons maintenant que le mécanisme de paramètres par variable est utilisé à la place du mécanisme copy-in-copy-out. Quel est le résultat produit par chacun des appels de procédure ci-dessus. Expliquer les différences.

Exercice 3

Soient les procédures ADA suivantes :

```
     type Vector is array (1 .. n) of Float;
     procedure add (v, w: in Vector; sum: out Vector) is
(1)  begin
     for i in 1 .. n loop
     sum(i) := v(i) + w(i);
     end loop;
(2)  end;
     procedure normalize (u: in out Vector) is
(3)  s: Float := 0.0;
     begin
     for i in 1 .. n loop
     s := s + u(i)**2;
     end loop;
     s := sqrt(s);
     for i in 1 .. n loop
     u(i) := u(i)/s;
     end loop;
(4)  end;
```

1) Considérons l'appel de procédure *add (a,b,c).* Montrer que cet appel produit le résultat attendu, si des mécanismes de copie ou de référence de paramètres sont utilisés. Est-ce que c'est pareil pour l'appel *normalize (c) ?*

2) Montrer que l'appel *add (a,b,b)* produit le résultat attendu, si des mécanismes de copie ou de référence de paramètres sont utilisés. Montrer que l'inoffensif aliasing peut survenir dans ce cas.

Exercice 4

Choisissez un langage de programmation (comme C ou PASCAL) qui ne supporte pas les packages, et les types abstraits de données.

a) Comment le package trigonométrique présenté ci-dessous peut être programmé dans le langage choisi ?

b) Quels sont les inconvénients de la programmation de cette façon?

```
package Trig is
function sin (x: Float) return Float;
function cos (x: Float) return Float;
end Trig;
package body Trig is
twice_pi: constant Float := 6.2832;
function norm (x: Float) return Float is
begin
. . . -- code pour calculer x modulo twice_pi
end;

function sin (x: Float) return Float is
begin
. . . -- code pour calculer le sin  de norm(x)
end;
function cos (x: Float) return Float is
begin
. . . -- code pour calculer le cos  de norm(x)
end;
end Trig;
```

Exercice 5

En utilisant un langage de programmation impératif, écrire à la fois une version itérative et une version récursive d'une fonction qui calcule b^n, b et n étant donnés comme arguments de la fonction. Quelles sont les variables locales nécessaires dans chaque version?

Exercice 6

Considérons le package *Dictionaries* vu dans le cours. Supposons la représentation suivante du type *Dictionary :*

```
capacity: constant Positive := 1000;
type Dictionary is
record
size: Integer range 0 .. capacity;
words: array (1 .. capacity) of Word;
end record;
-- Les mots sont stockés dans l'ordre croissant dans words(1..size).
```

Soient d1 et d2 de variables de type *Dictionary* et w une variable de type *Word*. Les quelles des instructions ADA suivantes sont légales.

```
d1.words(1) := w;
add(d1, w);
```

```
d2 := (d1.size, d1.words);
d2 := d1;
if d1 = d2 then . . . end if;
if d1.size > 0 then . . . end if;
```

Exercice 7

a) Concevoir un type abstrait *complexe*, dont les valeurs sont des nombres complexes. Equipez votre type abstrait avec des opérations telles que : l'addition complexe, la soustraction complexe, et la multiplication complexe.

b) Proposer une représentation efficace pour votre type abstrait.

Exercice 8

a) Concevoir un type abstrait *Date*, dont les valeurs sont les dates du calendrier. Equipez votre type abstrait avec des opérations telles que : le test d'égalité, la comparaison (" est avant "), ajoutant ou en soustrayant un nombre donné de jours, et la conversion au format ISO " aaaa-mm-dd ".

b) Proposer une représentation efficace pour votre type abstrait.

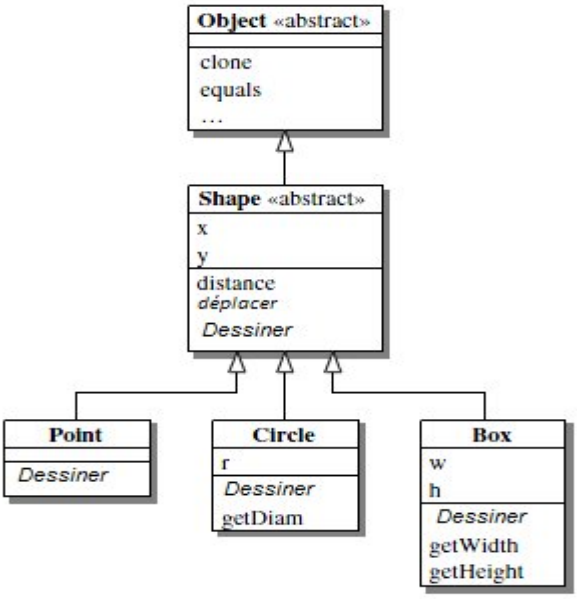
Chapitre III

Exercice 1

Ecrire en C++, une classe des nombres fractionnaires. Un nombre fractionnaire se compose d'un entier numérateur et d'un entier dénominateur. Une instance doit comporter les deux éléments constituant du nombre fractionnaire. Votre implémentation doit toujours réduire les nombres sous la forme la plus simple et propose une surcharge de tous les opérateurs arithmétiques.

Exercice 2

Considérons l'hiérarchie de classe illustrée dans la figure ci-dessous. Définir de nouvelles classes pour d'autres formes géométriques. Chaque classe doit disposer d'une méthode *dessiner* permettant d'afficher la forme. Placer les classes nouvellement définies adéquatement dans l'hiérarchie. Définissez une classe d'images, de sorte que chaque objet image est une collection de formes. Proposer une méthode *dessiner* qui affiche toute l'image, en affichant toutes ses formes composantes.



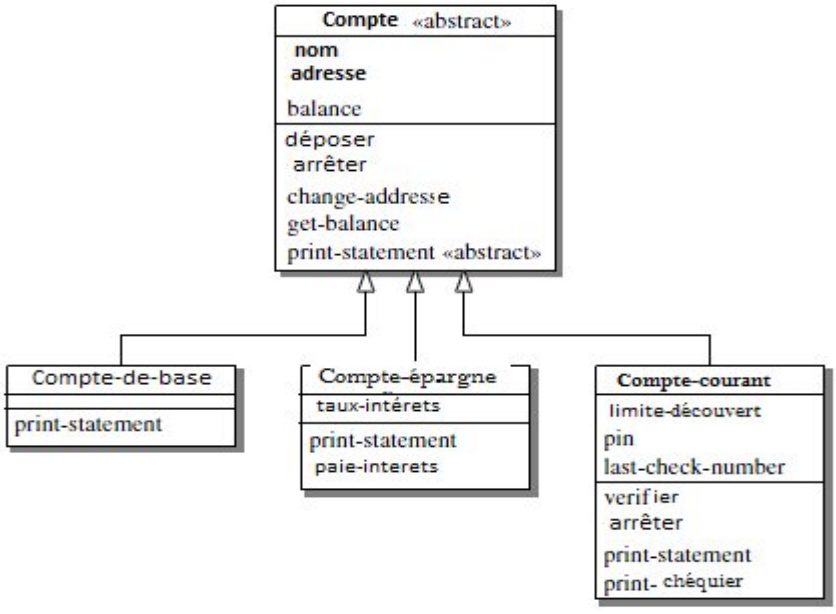
Exercice 3

Une banque propose les types suivants des comptes clients. Un compte de base qui n'a pas d'options. Un compte d'épargne qui rapporte des intérêts à la fin de chaque jour sur le solde courant. Un compte courant assurant au client une facilité de retirer à découvert, un chéquier, une carte de crédit, et un code PIN permettant la vérification de l'identités du client par les guichets automatiques.

Supposons que la hiérarchie des classes de la figure ci-dessous a été conçue pour le logiciel du maintient des comptes.

a) Quelles sont les méthodes héritées et les méthodes remplacées par chaque sous-classe.

b) Comment cette hiérarchie de classe peut être implémentée en C ++ ou en JAVA.



Exercice 4

Soit le code Java suivant :

```
1. public class Exercice
2. {
3.    static private String msg = null;
4.    static private int n;
5.
6.    Exercice(){
7.      n = 1;
8.      if (msg == null)
9.        msg = "Rouge";
10.     affiche();
11.   }
12.
13.   private void affiche(){
14.     System.out.println(n + msg);
15.     if (!msg.equals("Vert")){
16.       msg = "Vert";
17.       new Exercice();
18.     }
19.   }
20.
21.   public static void main(String[] args){
22.     Exercice x = new Exercice();
23.     n++;
24.     x.affiche();
25.     Exercice y = new Exercice();
26.     n++;
27.     x.affiche();
28.     y.affiche();
29.   }
30.
31. }
```

1) Ce programme peut-il être compilé ? Si oui, qu'affiche-t-il ? Si non, pourquoi ?

2) Mêmes questions en enlevant le mot-clé *static* de la ligne 3

3) Mêmes questions en enlevant le mot-clé *static* de la ligne 4

Exercice 5

La classe Robot modélise l'état et le comportement de robots virtuels. Chaque robot correspond à un objet qui est une instance de cette classe. Chaque robot :

• A un nom (attribut nom : chaîne de caractères)

• A une position : donnée par les attributs entiers x et y, sachant que x augmente en allant vers l'Est et y augmente en allant vers le Nord,

• A une direction : donnée par l'attribut direction qui prend une des valeurs "Nord", "Est", "Sud" ou

"Ouest"

• Peut avancer d'un pas en avant : avec la méthode sans paramètre avance()

• Peut tourner à droite de 90° pour changer de direction (si sa direction était "Nord" elle devient "Est", si c'était "Est" elle devient "Sud", etc.) : avec la méthode sans paramètre droite(). Les robots ne peuvent pas tourner à gauche.

• Peut afficher son état en détail (avec de simples System.out.println())

Le nom, la position et la direction d'un robot lui sont donnés au moment de sa création. Le nom est obligatoire mais on peut ne pas spécifier la position et la direction, qui sont définis par défaut à (0,0) et "Est".

1) Écrire les instructions Java qui permettent de définir la classe Robot, en respectant le principe de l'encapsulation des données.

2) On veut améliorer ces robots en en créant une Nouvelle Génération, les RobotNG qui ne remplacent pas les anciens robots mais peuvent cohabiter avec eux.

Les RobotNG savent faire la même chose mais aussi :

• Avancer de plusieurs pas en une seule fois grâce à une méthode *avance*() qui prend en paramètre le nombre de pas

• Tourner à gauche de 90° grâce à la méthode *gauche*()

• Faire demi-tour grâce à la méthode *demiTour()*

Écrire cette nouvelle classe en spécialisant celle de la première question, sans modifier celle-ci :

a) Dans un 1er temps, les nouvelles méthodes appellent les anciennes méthodes pour implémenter le nouveau comportement : avancer de n pas se fait en avançant de 1 pas n fois, « tourner à gauche » se fait en tournant 3 fois à droite, faire demi-tour se fait en tournant 2 fois

b) Donner une 2e solution plus efficace qui change directement l'état de l'objet sans faire appel aux anciennes méthodes (...mais attention aux droits d'accès !)

3) On veut mettre ensemble dans un tableau des objets de type Robot et de type RobotNG.

a) Comment déclarer le tableau ?

b) Comment afficher l'état de tous les robots contenus dans le tableau ?

4) Modifier la classe RobotNG pour pouvoir activer un mode « Turbo » et le désactiver. Dans ce mode, chaque pas est multiplié par 3. L'appel à la méthode *afficher*() devra indiquer à la fin si le robot est en mode Turbo ou pas.

Chapitre IV

Exercice 1

Faites une comparaison détaillée entre le concept d'objet et le concept de composant.

Exercice 2

Faites une comparaison détaillée entre le développement à base de composants et celui basé services.

Chapitre V

Exercice 1

Quel est l'intérêt de la programmation par aspect ?

Exercice 2

Donner une expression d'un point de jonction (*pointcut*) pour appeler une méthode publique de n'importe quelle classe qui retourne un entier.

Exercice 3 :

Dans un système bancaire, Se limiter aux méthodes de programmation 'classiques' conduit ainsi à une complexification du code, de son développement et de sa maintenance. Proposez une conception préliminaire dans laquelle vous insistez sur les différentes préoccupations.

Chapitre VI

Exercice 1

Écrire en Scheme des fonctions qui déterminent :

– si un entier est positif.

– si un entier est pair.

– si les trois paramètres entiers forment un triplet pythagoricien.

– si deux entiers sont de même signe.

Exercice 2

Écrire une fonction Scheme *min2entiers* qui calcule le minimum de deux entiers passés en paramètres.

Exercice 3

Écrire une fonction Scheme *min3entiers* qui calcule le minimum de trois entiers passés en paramètres.

Exercice 4

Ecrire une fonction récursive en Scheme qui prend deux entiers et calcule le premier à la puissance du second.

Exercice 5

Écrire une fonction Scheme qui calcule le pgcd de deux entiers positifs.

Chapitre VII

Exercice 1

Un inspecteur veut connaître les suspects qu'il doit interroger pour un certain nombre de faits : il tient un individu pour suspect dès qu'il était présent dans un lieu, un jour où un vol a été commis et s'il a pu voler la victime. Un individu a pu voler, soit s'il était sans argent, soit par jalousie. On dispose de faits sur les vols, par exemple, La personne 'M'a été volée lundi au cinéma, la personne 'J', mardi au restaurant, la personne 'L', jeudi au stade. Il sait que 'A' est sans argent et que 'E' est très jalouse de 'M'. Il est attesté par ailleurs que 'A' au restaurant mercredi, 'E' au restaurant mardi. et que 'E' était au cinéma lundi. (On ne prend pas en compte la présence des victimes comme possibilité qu'ils aient été aussi voleurs ce jour là). Ecrire le programme Prolog qui, à la question suspect(X), renverra toutes les réponses possibles et représenter l'arbre de recherche de Prolog.

Exercice 2

Définir en Prolog les clauses nécessaires à la concaténation des listes, exemple "conc(⌈a⌋, ⌈b, c⌋, ⌈a, b, c⌋)." est vrai, "conc(X, ⌈a⌋, ⌈b⌋)." est impossible. Dessiner l'arbre de résolution de "conc(U, V, ⌈a, b⌋)." en marquant sur chaque arc les instanciations des variables.

Exercice 3

Construire en Prolog les prédicats "dernier(X, L)" où X est le dernier élément de L, et "zajouter(X, L, R)" où R est la liste L augmentée de X en dernière position.

Exercice 4

Proposer un prédicat Prolog de la forme member(X,L) tel qu'il est vrai ssi X est un élément d'une liste L.

Exercice 5

Etant donné les prédicats parent(X,Y) est vrai si X est parent de Y, masc(X) vrai si X est masculin, fem(X) vrai si X est féminin, permettant de décrire une famille, et le programme Prolog suivant, décrivant une certaine famille :

```
parent(Djamel,mohammed).
parent(mariya,mohammed).
parent(amine,Djamel).
parent(Loubna,Djamel).
parent(Farid,mariya).
parent(annissa,mariya).
masc(Djamel).
masc(mohammed).
masc(Farid).
masc(amine).
fem(mariya).
fem(annissa).
fem(Loubna).
grandparent(X,Y) :- parent(Z,Y), parent(X,Z).
```

Donner l'arbre SLD du but suivant : grandparent(X,mohammed), fem(X).

Bibliographie

- Programming language design concepts, David A. Watt and William Findlay, John Wiley & Sons Ltd, 2004.
- Advanced programming language design, Raphael A. Finkel, Addison-Wesley, 1996.
- Essentials of Programming Languages, Daniel P.Friedman Mitchell W and Christopher T.Haynes, MIT Press, 2001.
- Building Reliable Component-Based Software Systems, Ivica Crnkovic and Magnus Larsson, Artech House, 2002.
- La programmation orientée aspect pour Java/J2EE. Pawlak Renaud, Retaillé Jean-Philippe, and Seinturier Lionel. Eyrolles, 2004.
- Approche basée agents mobile et composants pour développer des applications ouvertes et adaptables, Siam Abderrahim, thèse de l'université de Constantine 2, 2015.
- Software Engineering 9th ed, Lan Sommerville, Addison-Wesley, 2011.