BULLETIN D'INFORMATIQUE APPROFONDIE ET APPLICATIONS

N° 38 JUIN 1994

Publication gratuite trimestrielle de l'Université d'Aix - Marseille II
58, boulevard Charles Livon. F - 13284 Marseille Cedex 07
Téléphone : (33) 91 39 65 00 Télécopie : (33) 91 31 31 36

Edition 1996

ISSN 0291 - 5413

BULLETIN D'INFORMATIQUE APPROFONDIE ET APPLICATIONS

N° 38 JUIN 1994

Publication gratuite trimestrielle de l'Université d'Aix - Marseille II
58, boulevard Charles Livon. F - 13284 Marseille Cedex 07
Téléphone : (33) 91 39 65 00 Télécopie : (33) 91 31 31 36

Edition 1996

ISSN 0291 - 5413

EDITORIAL,

Humour informatique et économie.

L'apparition de l'informatique sur le marché du travail provoqua les réflexes habituels. Cette science sale et bizarre était, c'est normal, regardée d'un drôle d'oeil par les beaux esprits. A commencer par les mathématiciens, pour lesquels l'objet était vraiment sale. Certains d'entre eux un peu moins sensibles à la répugnance, ou ne trouvant peut-être pas d'autre voie pour leur carrière inventèrent un moyen terme: la mathématique appliquée. Ce participe passé doit être prononcé très vite, et doit se fondre dans un soupir, presque en filigrane.

Dans le milieu du travail, le phénomène fut du même ordre, bien que les causes en soient légèrement différentes. Vous imaginez une entreprise avec ses cadres, sa comptabilité, son personnel, ses habitudes, sa routine et vous menacez d'introduire là dedans quelque chose dont on ne sait pas trop à quoi ça va servir sinon à bouleverser un équilibre fragile. Quelque chose de trop neuf en bref. Tout le monde commence à se regarder avec d'autres yeux. D'abord qui va s'occuper de ça ? on sent qu'il va falloir se mettre à des choses nouvelles, peut-être difficiles, et si en plus on rate on risque de passer pour quoi ? S'engager pour jouer une réputation acquise au prix de longues années d'efforts ? D'un autre côté laisser la place à un jeune godelureau qui va vous marcher sur les pieds, Haa ! l'affreux dilemme.

Or, il existe toujours dans un groupe, un casse-pieds, un enquiquineur, dont on ne sait trop comment se débarrasser. Voila l'occasion rêvée. On lui confie le département informatique, et on se frotte les mains en attendant qu'il se casse la figure. Et nombre d'entreprises se sont cassé la figure, si j'ose dire, en pratiquant de la sorte. Le personnel peu qualifié, le logiciel cher, peu commode et mal adapté, un ordinateur beaucoup trop gros, car le vendeur fourmille d'arguments pour expliquer à quelle vitesse croissent les besoins ...

C'était dans les débuts. Depuis les choses ont bien changé, l'arrivée sur le marché d'une micro-informatique puissante a déclenché une ère nouvelle. Désormais, tout produit qui n'a pas été conçu, élaboré, empaqueté par "l'informatique" est un produit douteux. Même les huissiers, s'ils ne veulent pas tâter du chômage sont obligés de prouver que leurs instrumentations sont pilotées par le dernier cri de l'informatique.

Les mathématiciens eux-mêmes ont enfin découvert l'informatique, et par exemple, pour le traitement des systèmes dynamiques difficiles à attaquer autrement, deviennent des consommateurs de milliers d'heures. Seuls les économistes n'ont pu encore saisir l'intérêt de cette surface sociale, mais l'efficacité des théories économique n'est plus à démontrer ...

L'affinement du logiciel, le perfectionnement des instruments de contact de l'ordinateur avec son environnement ont provoqué, dans les rangs mêmes des gens qui côtoient l'informatique de grandes brèches. D'abord ce sont les "opérateurs", ces manipulateurs de rubans magnétiques, et les "perfo", ces filles qui passaient leur temps à faire des trous dans des petits cartons qui ont disparu, au point qu'on ne sait même plus s'ils ont existé. Et puis, actuellement la profession de "secrétaire" commence à être sérieusement touchée, et toutes ces professions de simples manipulateurs amplement remplacés par des automates pilotés par ordinateur.

Le système ultra-libéral a d'ailleurs sauté à pieds joints sur les capacités de l'informatique à remplacer une main d'œuvre de plus en plus coûteuse à cause des lois sociales, de plus en plus spécialisée, de plus en plus permutable, grâce à la parcellisation du travail, et qui, remplaçable en dernier ressort par des automates, se retrouve illico au chômage. Même les banques s'y mettent dont les immenses réseaux informatisés remplacent et au delà, des troupes de petits comptables, qu'on jette par milliers à la rue.

Ils ont bon air les Saint Jean Bouche d'Or de la résolution de la fracture, quand ils brandissent l'étendard de la lutte contre le chômage, alors que celui-ci, comme par un hasard étrangement malheureux, continue de gonfler, sourd aux incantations hypocrites de nos énarques. Par ailleurs, comme disent les présentateurs de télévision, le paysage industriel change. Et pendant que les milliers de métallurgistes, désormais inutiles pleurent sur leur usine morte qui les a abrutis toute leur vie d'un vacarme infernal, qui leur a brûlé le cuir et les poumons de ses feux et de ses déjections redoutables, les grandes banques internationales jonglent avec les richesses qu'ils ont amassées et dont ils ne profiteront jamais.

Si l'acier a perdu de son importance, le trafic de finances, rendu facile par la souplesse qu'apporte l'informatique, permet aux banques et à leurs séides d'investir et de naviguer dans un maximum d'affaires qui n'ont que peu de chances de revenir pour améliorer le sort de ceux qui sont à l'origine des richesses manipulées.

Les hérauts de la lutte à mort contre le chômage continuent sans vergogne à pousser cet étrange cri de guerre, qu'il soit de gauche, de droite ou du centre, peu importe, alors que de toute évidence la nouvelle société rejette les travailleurs, non par méchanceté mais simplement comme une denrée trop coûteuse. De plus en plus nombreuses sont les entreprises qui conservent une vague raison sociale

sur l'hexagone, mais dont les activités s'exercent sur un sol où le travailleur se paye à la roupie de sansonnet. Et grâce aux progrès des autoroutes de l'information, les traitements informatiques eux-mêmes sont rejetés dans ces paradis de la main d'œuvre bon marché.

Cette tendance qui consiste à remplacer au maximum une main d'œuvre quelquefois turbulente, bien qu'assagie par la pression du chômage en tout cas toujours onéreuse, par des machines plus productives, est en pleine accélération.

Bien que freiné et méprisé par une caste élitiste, notamment par l'inégalité de répartition des moyens, l'accroissement du niveau intellectuel de la population est réel, mais contrairement à ce qu'on pourrait imaginer, ceci induit un appauvrissement. Simplement parce que la connaissance n'est pas traitée comme une valeur humaine, mais comme une valeur marchande. Quand il y a trop de docteurs, on en met un maximum au chômage, et on sous-paye les autres en les employant à n'importe quoi, autre gaspillage.

Nous traversons une phase où véritablement la machine peut avantageusement remplacer l'homme dans bien des tâches ingrates, pénibles, dégradantes. Ce n'est pas forcément un mal. Bien des philosophes du XIX ème siècle annonçaient et souhaitaient cet âge qui pourrait devenir un âge d'or.

Mais c'est compter sans les margoulins.

Se foutent-ils de nous ces personnages importants, énarques et autres prétendus-diplômés, qui continuent à ronronner les mêmes inepties, et avec le même bonheur, qu'au début de ce siècle? Ou est-ce la paresse intestinale de leur pauvre cerveau qui leur donne cette allure nettement Alzheimer?

Peut-être faudrait-il un peu revenir sur la notion sacrée de travail, qui, jointe au chômage n'est plus qu'un instrument de destruction de la revendication sociale, et envisager un autre moyen de partage des richesses que celui qui passe par une dévaluation complète des valeurs humaines, connaissance et dignité. Mais précisément ceux qui sont au pouvoir sont les premiers à tricher, soit par abus de pouvoir pour ceux qui ont le vrai pouvoir, celui de l'argent, soit par aplatissement devant les premiers, pour les autres.

E. Bianco

LE MOMENT DE L'ANGOISSE

Jean Yves MAISONNEUVE

Présentation de Jean - Michel Knippel (*)

Après un moment d'humour, dû à l'éditorial d'Edmond Bianco, plongeons nous dans "le moment de l'angoisse"?

Le langage n'a pas cessé au cours de l'histoire de la pensée de l'homme et de la femme de soulever l'interrogation. L'inconscient est structuré comme un langage, d'après la thèse de Lacan. Le développement des langages formels, les travaux des logiciens, depuis Frege et Turing, ouvraient la voie à l'automatisation du discours. En même temps, les linguistes révélaient des structures immanentes aux langues naturelles.

Au carrefour des théories formelles et linguistiques, l'informatique dote une machine de tous les apparats du langage. L'automate doté d'un langage doit pouvoir être compris par l'utilisateur humain. Il s'agit de permettre à la machine d'atteindre au langage des hommes et femmes, et d'en automatiser les traitements les plus divers (indexation et résumés de textes, traduction d'une langue à une autre, interfaces en langue naturelle...).

Certains virent dans nos machines des images de notre propre corps et puisèrent dans ce reflet mécanique. Il s'agit au travers de cet article d'engager une réflexion pour déterminer ce que le reflet informatique de notre esprit peut nous apprendre sur nous-mêmes.

Turing, Von Neumann ont posé la question de savoir si une machine pouvait penser. La réponse nous autorise à la lecture de la suite...

(*) inspiré de Gérard Chazal.

LE MOMENT DE L'ANGOISSE.

Pour vous parler de l'angoisse,du moment de l'angoisse,j'ai travaillé un article de Freud,daté de 1919,"Das Unheimliche",traduit en français sous le titre :"L'inquiétante étrangeté".Hé bien,je n'y ai rien compris,rien de rien,jusqu'à ce que je m'avise que dans cet article Freud ne nous parle pas tant d'étrangeté,aussi inquiétante fusse-t-elle,que d'histoires d'esprits,de fantômes,de revenants,de hantise dirais-je si ce mot n'était pas très affadi dans l'usage actuel de la langue.Ainsi il y est question de "unheimlich Hauss",de maison hantée,et c'est au coeur de la question.De même Freud a écrit "Die Traumdeutung",non pas la science des rêves,mais l'oniromancie.

En exergue,voici une citation de Jacques Lacan,tirée de son séminaire sur "Les quatre concepts fondamentaux de la psychanalyse":

Le monde est omnivoyeur,mais il n'est pas exhibitionniste;il ne provoque pas notre regard.Quand il commence à le provoquer,alors commence aussi le sentiment d'étrangeté.

Pour essayer de situer l'affect d'angoisse,en tant qu'il concerne la psychanalyse,je vais vous parler,donc, de l'étrangeté,...de la hantise.

Dans une note en bas de page de son article,Freud donne une anecdote personnelle,un exemple,qui a un certain relief.

Le Professeur Sigmund Freud était seul,assis dans un compartiment de wagon-lit,pendant un de ses voyages annuels sans doute.Un cahot un peu plus fort que les autres,et voici que la porte donnant sur le cabinet de toilette s'ouvrit.Alors entra un monsieur d'un certain âge,en robe de chambre,le bonnet de voyage sur la tête.Ce que voyant,Freud se dit que ce monsieur s'était trompé de direction en quittant le cabinet de toilette situé entre les deux compartiments.Mais alors,abasourdi,écrit-il,il s'aperçut que l'intrus était...sa propre image que lui renvoyait le miroir placé sur la porte intermédiaire.

Freud a noté que ce double qui apparaissait là ne l'avait pas effrayé.A ce point de son article,il venait de discuter la thématique du double,spécialement présente parmi d'autres que traite la littérature,en particulier Hoffman dans "Les élixirs du Diable".Ce double,il ne l'avait pas reconnu: c'était,dans ce miroir un instant invisible comme tel,un autre,radicalement étranger,juste avant,et en somme je dirai au moment même,de lui apparaître comme le plus proche.Mais il a noté aussi que ce qui lui était ainsi donné à voir lui avait foncièrement déplu.Et à l'instant où il fut détrompé,il fut abasourdi;pas soulagé,nous pouvons le remarquer.

L'étrangeté,l'Unheimliche,c'est un phénomène qui affecte le Moi.J'étais installé confortablement,Heimlich,comme à la maison,et quelque chose se détraque.Les règles changent sans crier gare.Quelque chose qui ne devrait pas se produit.Freud parle d'esprits et de fantômes dans son article,et de maison hantée:c'est une bonne métaphore,et certainement il vaudrait de redonner toute sa force au mot "hantise",toute sa charge fantomatique.

Freud ne croit pas aux fantômes,c'est sûr.Mais que ça revienne,que ça fasse retour,il prend ça au sérieux.

Il est très étonnant,mais pas tant que cela si nous suivons Lacan dans sa thèse qui pose que *l'inconscient est structuré comme un langage*,de lire que Freud appuie pratiquement toute sa conception du phénomène sur une analyse linguistique.

Sa trouvaille,à laquelle il consacre toute la première partie de l'article,c'est que l'adjectif "heimlich" (composé à partir de "Heim",équivalent du "home" anglais),prend des sens opposés,d'un côté :familier,proche,intime, de l'autre: caché,dangereux,hanté.Dans le second cas,il prend le sens de "unheimlich".

Ca permet à Freud de conclure à la lumière de ce que lui a appris la psychanalyse: que ce qui est qualifié d'"unheimlich",c'est ce qui était "heimlich",mais a subi le refoulement,et,faisant retour,n'est pas reconnu.L'étranger de cette sorte n'est autre que le très familier d'avant le refoulement.

Lacan reprend ceci dans son séminaire "L'angoisse",pour insister au passage sur l'importance de cette analyse linguistique dans le texte de Freud,justifiant la prévalence que lui,Lacan,donne aux fonctions du signifiant.

Un autre intérêt,et c'est ce qui nous occupe aujourd'hui,c'est de situer un point,une place,qu'il appelle,à partir du "Heim",la maison de l'homme.Et de préciser que l'angoisse est liée à tout ce qui peut apparaître à cette place.

Dans le cas de Freud,le déplaisir,comme il le note,peut être considère comme un reste d'angoisse.

Mais quel est le statut de l'angoisse?Il est tout entier contenu dans le "Un" de "Unheimliche".Ce "Un" qui pose sur le familier,sur le connu,sur l'amical,le sceau de l'inconnu,du fantomatique,du dangereux,c'est l'équivalent de ce qui fonctionne dans certaines langues,au moins dans leur écriture,comme index qui change,ou détermine,le sens d'un mot.C'était le cas en égyptien,par exemple,à quoi se réfère Freud.Voir à ce sujet son article "Sur les sens opposés des mots originaires".

Ainsi l'angoisse fonctionne comme index,comme signal,dans ces cas d'étrangeté,d'influence,de hantise,signal lié à ce que la psychanalyse freudienne établit: qu'il s'agit du retour du refoulé;autrement dit l'angoisse se produit au lieu même du refoulement.Freud généralise cette valeur de signal à tous les cas d'angoisse.

C'est le privilège de l'angoisse,dont Lacan dit que c'est un affect qui ne trompe pas: quand elle apparaît,cela signale qu'une certaine proximité est atteinte.Ainsi dans la psychanalyse est-il question du moment,de la date du début de l'angoisse,et guère de ses caractères physiologiques,qui,eux,intéressent la médecine.Et cette date,ce moment,c'est dans ses coordonnées langagières qu'il sera replacé.

UN MODELE DE CONCEPTION BASE SUR LES CONCEPTS DE MODULES, DE CLASSES ET D'INSTANCES

Michel LAI
INGENIA
Technopôle de Château-Gombert
Europarc, Bât C
13013 MARSEILLE
Tel : 91 05 50 22, Fax : 91 66 35 99
E-Mail : lai@ingenia.fr

Résumé: Le but de cet article est de présenter dans un contexte méthodologique, une démarche basée sur les concepts de module, de classe et d'instance. L'émergence des méthodes de conception et d'analyse dites orientées objets a eu pour conséquence l'introduction de notions pouvant comporter différentes interprétations. Il est fait souvent référence aux types abstraits de données, ces derniers faisant à leur tour appel aux algébres formelles informatiques. Cependant, ces justifications restent généralement insuffisantes pour expliquer les caractéristiques des langages par objets actuellement utilisés. Dans le cadre de la définition d'une nouvelle méthode de conception, nous montrons comment s'articulent les notions essentielles pour permettre aux utilisateurs de langages tels que TELOS[LEL 92], OBERON[MOS 92], C++[STR 87], CLOS[CLO 90], EIFFEL[MEY 90] et ADA[TUC 92] d'exprimer leur solution en conformité à la fois, avec le monde réel et le langage de programmation utilisé.

Abstract : The goal of this article is to present a unifying process of the concepts of module, class and instance in a methodological context. Emerging object-oriented design and analysis methods have introduced notions which may be interpreted in different manners. One often refers to abstract data types, which are also connected to formal computanional algebra. However, these theorical justifications are not good enough for explaining the features of object-oriented languages used nowadays. Here, we define a new design method and we show how these essential notions are connected and how they enable users of languages such as TELOS, OBERON, C++, CLOS, EIFFEL and Ada express their solutions while conforming with the real world and the programming language they use.

1. Introduction

Plusieurs travaux relatent le besoin de distinguer les notions de module, de classe et de type[GOG 87-a][SYP 92][COO 92][ROS 92]. En particulier dans [GOG 87-b], les auteurs arguent et justifient la nécessité de différencier, les données, les objets, les modules, les types et les classes. Les divers langages de programmation ont adopté des points de vue différents, engendrant des implantations distinctes [MIT 90]. Ainsi, l'ensemble des utilisateurs ne dispose pas d'un consensus réel permettant d'éviter des incompréhensions préjudiciables à la réussite d'un développement. Ces inconvénients sont amplifiés lorsqu'il s'agit d'expliquer en phase d'analyse ou de conception les raisons des choix effectués. Malgré plusieurs travaux sur la

formalisation des concepts objets[ACM 90][ACM 92][OOP 91][TOO 92][ECO 92], une confusion persiste chez les utilisateurs des méthodes orientées objets telles que, par exemple, la méthode HOOD [HOO 91], la méthode OOD de Booch [BOO 92] ou la méthode OMT [OMT 92]. Ces méthodes – comme pour l'instant la grande majorité des méthodes utilisées dans l'industrie – sont peu formalisées. Les notations graphiques constituent une grande part de leur définition. Même si les fondements théoriques existent, l'utilisateur final n'a pas nécessairement la possibilité de les percevoir.

Dans le cadre d'un projet de définition d'une méthode d'une conception orientée objet, nous avons été amenés à refaire la génèse du processus de conception suivi par un ingénieur-concepteur. Afin de dégager le modèle conceptuel servant de base à notre méthode, nous avons examiné ce qui est nécessaire d'une part, en phase d'analyse ou de conception, d'autre part en phase de codage suivant les langages employés.

Nous avons ainsi dégagé des points forts à partir desquels les spécifications d'un outil de conception ont été réalisées. Cet outil appelé OTool a été développé en utilisant Le-Lisp, AIDA et MASAI[1]. Une version de l'outil est prévue avec une interface ELSY. Le résultat de la conception est alors intégré dans l'environnement de développement pour des conditions optimales d'utilisation.

Dans un premier temps, nous présentons la notion de concept sur laquelle est basée l'élaboration de notre démarche. Ensuite, nous montrons comment la construction d'un type abstrait de données permet de dériver, soit le concept de module, soit celui de classe. Puis, nous abordons une discussion sur l'utilisation d'un objet en phase de conception. La présentation des différentes relations utilisées entre les objets par les concepteurs permettra d'introduire notre modèle de conception.

2. La notion de *concept*

Avec l'introduction de l'analyse et de la conception orientée objet, le découpage strict en phases de cycle de vie du logiciel est difficile à réaliser. Il est possible de retrouver des objets en analyse, en conception ainsi qu'en phase de codage. Plus nous nous rapprochons du monde réel et de l'utilisateur de l'application à concevoir, plus les notions d'objet et de classe tendent à fusionner. L'essentiel peut être désigné par un triplet $< N, D, O >$ où N est un nom, D un ensemble de données formelles typées (agrégation de données élémentaires, structure ou enregistrement) et O un ensemble d'opérations. Une opération pouvant être par la suite considérée comme une procédure ou une fonction. Au début de l'analyse, par analogie avec le monde réel ou par expérience, le concepteur introduit des idées d'objets en choisissant des noms appropriés. Les noms ainsi choisis sont fortement chargés de signification, principalement pour le créateur de ces noms. Cependant, cette connaissance n'est pas exprimée. Initialement, cette connaissance demeure une simple chaîne de caractères. Les noms introduits par le concepteur en phase initiale de l'analyse forment ce que nous appelons les concepts de l'application.

---

[1] OTool est en cours de commercialisation par la société ILOG. Le-Lisp, AIDA, MASAI et ELSY sont des produits distribués par la société ILOG SA. Gentilly.

Définition 2.1 - Concept : Un concept est un nom introduit par le concepteur en phase d'analyse pour désigner une entité du monde réel.

Du fait de la simplicité de la présente définition, nous nous interdisons d'utiliser une quelconque interprétation d'un concept. Un concept est à prendre comme un axiome. Voici quelques exemples de concepts issus d'une application : *VOL, PILOTE, COMPTE, AVION.* Il est préférable de prendre un nom commun au singulier pour dénoter un concept. Une formalisation moins intuitive de la notion de concept peut être trouvée dans [BIE 90] pour des applications liées au traitement de la connaissance – concepts génériques, liens taxonomiques et rôles –.

Définition 2.2 - Concept pluriel : Un concept pluriel désigne un regroupement de *copies* d'un même concept – collection, liste, ensemble etc...–

Exemple : *VOLS, PILOTES, COMPTES* ou *AVIONS.*

A partir d'un concept de base, la démarche conceptuelle de l'architecture logicielle d'une application va s'exprimer en précisant les caractéristiques structurelles ou opérationnelles sous jacentes à un concept. Suivant la priorité donnée par le concepteur, un concept sera étendu par l'adjonction d'un ensemble de noms représentant, soit un ensemble de données, soit un ensemble d'opérations. L'ajout de caractéristiques relève de la description statique des propriétés attachées à un concept. L'ajout d'opérations est relatif à la description dynamique ou comportementale d'un concept. La première association d'un nom à une liste de noms – pouvant eux-mêmes être des concepts – est une abstraction permettant d'identifier un ensemble de noms par un autre. Cette association seule ne relève pas de la démarche orientée objet. Construire un enregistrement de données peut se faire en Pascal ou en COBOL. La dénotation par un nom d'une opération est relative à des aspects fonctionnels. Le plus souvent en termes de spécifications initiales, les actions à réaliser sont évoquées avant même d'avoir introduit des concepts. La donnée d'un nom d'une opération est souvent moins signifiante que celle d'un attribut, sauf pour des opérations bien connues comme *empiler, depiler, trier....* Aussi il est nécessaire d'apporter une caractérisation des opérations en donnant leur domaine de définition ainsi que leur co-domaine.

Définition 2.3 - Signature d'une opération : Soit un nom dénotant une action du domaine d'analyse d'une application à réaliser. La signature conceptuelle σ de ce nom est l'ensemble des concepts nécessaires pour définir l'action qu'il dénote.
Soit $f : C_1 \times \ldots \times C_n \rightarrow C_{n+1}$ alors $\sigma(f) = C_1 \times \ldots \times C_n \rightarrow C_{n+1}$.

Par la suite, nous désignerons une opération par un nom de verbe à l'infinif. Donnons un exemple de signature conceptuelle. Soit l'opération qui consiste à déterminer la durée total de vol d'un avion pour une période donnée. Supposons les concepts suivants donnés : *VOL, DUREE, AVION, PERIODE.* Cette opération a pour signature :

$$\sigma(LireDureeTotale) = VOL \times AVION \times PERIODE \rightarrow DUREE$$

Pour tout concept, il est possible d'introduire un ensemble de caractéristiques. L'ensemble des caractéristiques est initialement exprimé également par des noms. En effet, si on considère un attribut par exemple *Age* de la classe *PERSONNE*, il ne

fait aucun doute qu'il s'agit probablement de l'âge de la personne. Le type *T-Age* qui permettrait de définir correctement le champ *Age* est donc ici surperflu. Quel intérêt aurait-on à obliger le concepteur de déclarer *Age : T-Age* ? Cependant, cette omission de type ne peut être admise que pendant un temps. Par la suite, l'expression de la définition de *T-Age* sera exigée, ce qui introduira certaines difficultés. Si naturellement il vient à l'esprit que *T-Age* est un sous type d'entiers positifs non nul, qu'en sera-t-il de la valeur maximale admise ou comment exprimer le fait qu'un âge puisse s'exprimer en mois, en années entières ou même sous une forme fractionnaire ? Ces différentes formes d'interprétations d'une même entité ont été formalisées par l'introduction des classes multi-aspects par J.C Royer dans [ROY 92]. En ce qui concerne notre propos, nous avons seulement besoin de dénoter l'ensemble des attributs typés – pouvant ou non comprendre des variables d'instances – par un nom. Dans la plupart des langages de classes ce nom est celui de la classe, tandis que dans les langages basés sur l'extension de types tels que OBERON 2 [MOS 92] et Ada9X [TUC 92], le nom est celui d'un type article placé nécessairement dans une structure syntaxique englobante ayant aussi un nom – nom de paquetage ou nom de module –.

Le principe d'abstraction et d'information cachée appliqué aux concepts se traduit par la désignation de ses caractéristiques par un nom. Nous dirons que ce nom dénote le type de la structure des données associées à un concept. Il est à noter qu'il s'agit bien de données formelles – pas de correspondance réelle avec la mémoire d'un calculateur –. Par convention un nom de type d'une structure commencera par *T-* suivi du nom du concept.

Définition 2.4 - Type d'un concept : On appelle type d'un concept dénoté par un nom, un nom *T-nom* désignant la nature des caractéristiques associées à ce concept.

Le type d'un concept peut-être visible ou non. Si le type d'un concept est caché son nom devra être suivi par le mot **hidden**. Le détail des caractéristiques d'un type peut être rendu visible. Le nom du type sera alors suivi du mot **public**. Le cas où le nom de type ne serait pas visible alors que les données le seraient est exclu. Le tableau ci-dessous récapitule les cas considérés.

sans option	Public	Hidden
seul le nom de la structure est visible	le nom et la structure sont visibles	ni le nom ni la structure sont visibles

Si une opération donnée a pour image un produit cartésien de concepts alors elle peut être transformée en un ensemble d'opérations ayant chacune un seul concept pour ensemble d'arrivée.
Soit l'opération $oper : C_1 \times \cdots \times C_n \rightarrow D_1 \times \cdots \times D_k$, alors il existe k opérations telles que : $\forall j, 1 \leq j \leq k, oper_j : C_1 \times \cdots \times C_n \rightarrow D_j$ est une projection de l'image de *oper* suivant la composante D_j.

Une fois l'ensemble des opérations du domaine d'analyse ramené à des opérations ayant un seul concept pour image — pouvant ou non déjà se trouver dans le domaine de définition de l'opération —, le concepteur peut classer ses concepts par importance dans chacune des signatures. Le concept jugé le plus important aux yeux

du concepteur — relativement aux autres concepts d'une signature donnée — sera le concept auquel sera rattachée l'opération ayant cette signature. Les autres concepts seront dits être liés par une relation d'utilisation. Ce qui nous permet d'introduire notre définition d'un type abstrait de données.

3. Les types abstraits de données

Notre point de départ est la considération informelle d'entités telles que les utilisateurs intuitivement les définissent au début d'un projet. Bien que les types abstraits de données constituent un domaine très développé, nous ne retiendrons qu'une interprétation simplifiée. Des bases formelles sur les types abstraits de données sont présentées dans [GUT 78]. Des exemples pratiques sont disponibles dans [MEY 90]. La partie axiomatique d'un type abstrait de données est ici volontairement omise.

Définition 3.5 - Type abstrait de données : On appelle type abstrait de données — en abrégé TAD — un triplet $[\![C, T[\textbf{hidden}, \textbf{public}], O]\!]$ où C est un concept, T un type de concept et O un ensemble d'opérations $oper_i$ telles que l'image de chaque opération est composée d'un seul concept et C est dans $\sigma(oper_i)$.

La recherche des TDA est le but de la phase d'analyse. D'où la définition suivante :

Définition 3.6 - Réification : On appelle réification le processus de transformation d'un concept en un type abstrait de données.

Par abus de langage nous désignerons un TAD par le nom de son concept. A chaque TAD est associée une représentation dont la syntaxe est la suivante :

< Définition d'un TAD > ::=< nom-concept >
utilise < Liste de Concepts >
< nom-type > [**hidden**,**public**]
< Liste des opérations >

Exemple :

PERIODE **utilise** *DATE, DUREE*
T-PERIODE **hidden**
Definir : DATE × *DATE* → *PERIODE*
Debut, Fin : PERIODE → *DATE*
Duree : PERIODE → *DUREE*

Définition 3.7 - Encapsulation de données dans un TAD : Si le type de concept d'un TAD est *hidden*, ses opérations ont implicitement un effet de bord sur ses données.

Par suite les opérations peuvent être simplifiées en supprimant le nom du TAD des interfaces des opérations. Dans ce cas, il est inutile de conserver le nom du type de concept. L'exemple donné plus haut devient alors :

PERIODE **utilise** *DATE, DUREE*
Definir : DATE × *DATE* →
Debut, Fin : → *DATE*
Duree : → *DUREE*

A partir de la connaissance d'un TAD, nous dérivons les deux notions fondamentales de structuration d'une architecture logicielle. A savoir les notions de module et de classe. Pour la notion de module, il suffit d'introduire la présence effective de données – instances de types de concepts –. Pour la notion de classe, il suffit d'introduire une relation particulière entre les TAD. Cette relation est celle définie par l'héritage. Parmi les nombreuses définitions de l'héritage nous conservons celle qui est la plus répandue et la plus intuitive à comprendre. Il s'agit de l'héritage basée sur l'extension de type. Avec ce choix, l'héritage correspond à une réunion de types et une réunion d'opérations. Il est nécessaire de distinguer les éléments explicites d'un TAD et les éléments implicites. Les éléments explicites sont ceux qui apparaissent syntaxiquement dans la définition d'un TAD. Tandis que les éléments implicites sont ceux qui donnent la réelle caractérisation du TAD. La définition des éléments implicites d'un TAD définissent une relation d'héritage sur les TAD.

Définition 3.8 - Relation d'héritage entre TAD : On dit qu'une unité de TAD τ_1 hérite de l'unité de TAD $\tau_2, \tau_1 \neq \tau_2$, (notation $\tau_1 \succ \tau_2$) $\stackrel{def}{\Longleftrightarrow}$ implicitement

1) $\overleftarrow{T_{\tau_1}} \supseteq \overleftarrow{T_{\tau_2}} \cup T_{\tau_1}$,
2) $\underleftarrow{O_{\tau_1}} \supseteq \underleftarrow{O_{\tau_2}} \cup O_{\tau_1}$,
3) τ_2 est un TAD *virtuel* de τ_1.

La notation $\leftarrow$ indique l'information résultante du fait de l'héritage. Une unité ne peut pas hériter d'elle-même. La relation d'héritage est transitive. Un TAD virtuel ne posséde pas d'opération propre. Les opérations d'un TAD virtuel doivent être définies par ses héritiers.

De nombreux travaux portent sur la sémantique de la relation d'héritage [MIT 90][ROY 92]. Une nouvelle fois nous nous limiterons à une définition pragmatique. Au delà de la justification rigoureuse de l'implantation de telle ou telle interprétation, les définitions de simples règles pratiques d'utilisation font cruellement défaut. A notre point de vue, la mise en œuvre d'une relation d'héritage est motivée par les besoins suivants :

- réutilisation
- factorisation
- spécialisation
- classification
- multiformation

La réutilisation vise à importer des codes et des données existants. La factorisation procède d'une mise en commun de caractéristiques – attributs ou méthodes – suivant une démarche ascendante. La spécialisation permet d'étendre des propriétés déjà existantes par adjonction de nouvelles caractéristiques – démarche descendante – . La classification relève d'un besoin *naturel* de hiérarchisation arborescente. La multiformation, au travers du polymorphisme, permet la sélection *horizontale* d'une sous-classe parmi n sous-classes issues d'un même classe parente. Dans certains cas la relation d'héritage est aussi employée pour des objets composites. Ce qui devrait rester une exception. En revanche, il est rare dans les langages de pouvoir limiter l'héritage aux seules entités désirées. Dans certains langages, il est possible de faire l'énumération explicite des méthodes héritées de la super-classe. Cette possibilité étant toutefois inhabituelle, elle ne sera pas conservée.

$$[\![C]\!] = \{\tau, \tau \in A : \epsilon = 1\}$$

L'ensemble $[\![M]\!]$ est appelé l'ensemble des méta-modules. L'ensemble $[\![C]\!]$ est appelé l'ensemble des méta-classes. Les notions de méta-module ou de méta-classe sont nécessaires pour spécifier l'outil supportant la méthode que nous définissons. Du point de vue des utilisateurs ces notions demeurent cachées.

4. Les modules

Les modules sont depuis longtemps des entités structurantes importantes [WIR 79]. Malheureusement, en l'absence d'une définition standard d'un module, on trouve sous ce vocable de nombreuses significations. Ainsi dans le projet COMET, les modules constituent les lieux où se trouve la connaissance de la conception. Un module étant une structure et des spécifications de comportement inter-reliées par des conventions et des contraintes [MAR 92]. La hiérarchie entre modules comme dans [HOO 91] se retouve également dans [MAC 90] avec un apport particulier sur la description des comportements. Avec la généricité, les modules ont pu, trop rapidement, être assimilés à des sortes de classes. Enfin, avec les propositions d'Ada9X, une paquetage peut devenir un paramètre générique effectif, lors de l'instanciation d'un paquetage générique [SHE 92].

Avec la notion de méta-module, nous possédons le moule adéquat pour construire des modules effectifs. Un module est construit à partir d'un méta-module par une opération d'instanciation - isomorphisme canonique - donnée par :

$$\begin{array}{rccc} \tilde{\zeta}: & [\![M]\!] & \rightarrow & M \\ & \hat{m} & \mapsto & \tilde{\zeta}(\hat{m}) = m \end{array}$$

L'application $\tilde{\zeta}$ a pour propriété de conserver la relation *use* définit sur $[\![M]\!]$. Autrement dit :

$$\forall \hat{m}_1, \hat{m}_2 \in [\![M]\!]^2, \hat{m}_1 \; use \; \hat{m}_2 \Rightarrow \tilde{\zeta}(\hat{m}_1) \; \tilde{use} \; \tilde{\zeta}(\hat{m}_2)$$

La relation $\tilde{use}$ est la relation induite par l'application $\tilde{\zeta}$. Par abus de langage on pose : $\tilde{use} = use$.

Définition 4.15 - Relation d'inclusion dans M : Soient les modules m_1 *et* m_2 alors on définit l'inclusion $m_1 \subset m_2$ par :

- $\forall m_1, m_2 : m_1 \subset m_2 \Rightarrow m_1 \neq m_2$
- $\forall m_1, m_2, m_3 : (m_1 \subset m_2) \wedge (m_2 \subset m_3) \wedge (m_1 \not\subset m_3)$
- $\forall m_1, m_2 : (m_1 \subset m_2) \wedge (m_2 \not\subset m_1)$

Définition 4.16 - Visibilité d'un module : Soit U_m l'ensemble des modules utilisés par m. Soit $m_i \in M$ tel que $m_i \subset m$ alors la visibilité du module m_i est égale à

$$\{m_j, j \neq i, m_j \in M \wedge m_j \subset m\} \cup U_m$$

Notation : On pose $S_m = \{m_j, 1 \leq j \leq n, (m_j \in M) \wedge (m_j \subset m)\}$ l'ensemble des modules inclus dans le module m. Si $S_m = \emptyset$ le module m est dit *terminal.*

Entre les opérations d'un module et celles des modules qu'il contient, il ¡existe une relation exprimée par les propriét'es suivantes.

Définition 3.9 - Relation de visibilité sur l'ensemble des TAD : On dit qu'un TAD τ_1 voit le TAD $\tau_2 \overset{def}{\Longleftrightarrow}$ (τ_2 est dans la portée de visibilité de τ_2) $\overset{def}{\Longleftrightarrow}$ (τ_1 est autorisé à utiliser τ_2 ou les entités de sa partie visible – explicitement ou implicitement –).

Notation : $\tau_1 \triangleright \tau_2$: τ_1 voit le TAD τ_2.
Propriétés :

- $\tau_1 \triangleright \tau_2 \Rightarrow \tau_2 \not\triangleright \tau_1$
- $\forall \tau, \tau \not\triangleright \tau$
- $\exists \tau_1, \tau_2, \tau_3 : (\tau_1 \triangleright \tau_2) \wedge (\tau_2 \triangleright \tau_3) \wedge (\tau_1 \not\triangleright \tau_3)$

Définition 3.10 - Relation d'utilisation entre TAD : La relation d'utilisation entre types abstraits de données (notée *use*) est construite sur la relation de visibilité de la manière suivante :
(τ_1 *use* τ_2) $\overset{def}{\Longleftrightarrow}$ ($\tau_1 \triangleright \tau_2$) $\wedge$ (($type(\tau_2)$ est utilisé par τ_1) $\vee$ (une opération de τ_2 est appelée par τ_1) $\vee$ (τ_2 appartient à la signature d'une opération de τ_1)).

Définition 3.11 - Opération d'absorption : On dit qu'un TAD τ est le résultat d'une opération d'absorption d'un TAD τ_2 par un TAD $\tau_1 \overset{def}{\Longleftrightarrow}$

- $(T_\tau = T_{\tau_1} \cup T_{\tau_2}) \wedge$
- $(O_\tau = O_{\tau_1} \cup O_{\tau_2}) \wedge$
- $(\forall j \mid \tau_1 \succ \tau_j \Rightarrow \tau \succ \tau_j) \wedge$
- $(\forall k \mid \tau_2 \succ \tau_k \Rightarrow \tau \succ \tau_k)$

Définition 3.12 - Exportation d'un type de concept : Si le type de concept d'un TAD est public, ses opérations peuvent avoir un effet de bord seulement sur les données associées aux TDA *utilisateurs*. Un TAD muni d'un type de concept public ne possède pas de données propres.

Définition 3.13 - Unité : On appelle unité formée à partir d'un TAD, une entité composée de deux parties : la spécification – décrite par le TAD –, l'implantation – partie dans laquelle se trouve les données formelles associées au TAD ainsi que le code des opérations du TAD –.

A ce niveau de notre exposé, nous avons suffisamment d'éléments pour aborder la séparation entre les notions de module et de classe. La distinction des deux notions relève d'un choix – normalement délibéré – du concepteur.

Définition 3.14 - Fonction de discrimination : A partir d'un type abstrait τ et en fonction d'une analyse donnée d'un domaine d'application, le concepteur définit une fonction de choix ξ définie par :

$$\begin{array}{ccccc} \xi : A & \rightarrow & & & \{0,1\} \\ \tau & \overset{\xi}{\mapsto} & \epsilon = & \begin{cases} 0 & \text{si } \tau \in [\![M]\!] \\ 1 & \text{si } \tau \in [\![C]\!] \end{cases} & \end{array}$$

La condition d'appartenance à $[\![M]\!]$ – respectivement à $[\![C]\!]$ – exprime la décision du concepteur de construire un module – respectivement une classe –. On pose par convention :

$$[\![M]\!] = \{\tau, \tau \in A : \epsilon = 0\}$$

Propriétés :
- Si $S_m \neq \emptyset \ \wedge \ U_m \neq \emptyset \Rightarrow \forall m_i \in U_m \ \exists m_k \in S_m : m_k \ use \ m_i$.
- Si $S_m \neq \emptyset \Rightarrow \forall {op^i}_m \in m \ \exists m_k \in S_m, \exists! op^j_{m_k} \in m_k : m.op^i_m() \Rightarrow m_k.op^j_{m_k}()$.

Pour différencier les raisons de création d'un module et pour bien distinguer la notion d'un objet construit à partir d'un module ou d'une classe nous ajoutons les précisions suivantes.

Définition 4.17 - Serveur de données : Un serveur de données est une unité de TAD possédant des données effectives cachées décrites par un type de concept ayant l'attribut *hidden*.

Définition 4.18 - Serveur d'opérations : Un serveur d'opérations est une unité de TAD ne possédant pas de donnée cachée dont le type de concept est public ou non.

Dans le cas d'un serveur d'opérations, l'unité est aussi un serveur de type de concept. Les données effectives associées à ce type seront alors des parties d'unités utilisatrices.

Définition 4.19 - Module en tant que serveur : Un module est soit un serveur de données, soit un serveur d'opérations.

Le dernier cas devrait être, d'un point de vue méthodologique, réservé uniquement pour les bibliothèques d'utilitaires. Le cas où un module serait à la fois serveur de données et d'opérations peut également arriver. Le concepteur doit chercher toutefois à l'éviter.

A partir d'un module on définit un constructeur d'objet *unique* pour chaque module. Alors que pour les classes nous avons la possibilité d'en construire plusieurs. Pour avoir plusieurs objets à partir d'un même module il faut utiliser les modules génériques – noté $[M]$ –.

$$\begin{array}{rccc} \tilde{\theta}: & M & \rightarrow & \tilde{O}_b \\ & m & \mapsto & \tilde{\theta}(m) = \tilde{o} \end{array}$$

L'application $\tilde{\theta}$ est définie comme la déclaration d'un objet $\tilde{o}$ ayant pour type le type m. Pratiquement, cette déclaration n'a d'utilité que lorsque des données sont effectivement créées – cas de modules serveurs de données –. Le langage de programmation Ada [Ada 83] fait figure d'exception avec les types tâche –*TASK type*, modélisation de comportements et non de données –. Cependant, lorsqu'une seule tâche est déclarée, la déclaration d'un type tâche devient implicite. De manière interne IBM, dans un langage de conception dénommé SEDL[SED], considérait la définition d'un module comme un type. Cela permettait de déclarer des modules comme des variables et de les utiliser en tant que telles – tableaux, *record* de modules, passages de paramètres –. Il est à noter que ces types d'un genre spécial peuvent avoir certaines restrictions – l'affectation entre tâches est interdites en Ada, mais l'affectation de pointeurs sur un type tâche est autorisée –. Par compatibilité avec les situations les plus courantes, nous adopterons les conventions suivantes :
On pose : L'ensemble $\tilde{O}_b$ est égal à M et $\forall m \in M, \tilde{\theta}(m) = m$.
De plus les éléments de M ne définissent pas de type. D'où la définition ci-après.

Définition 4.20 - Instance d'un module : Une instance d'un module est le module lui-même.

C'est ce type de définition couramment admise qui provoque une confusion chez les utilisateurs.

Nota : une opération est un cas dégénéré de module. Par abus de langage on pose $nom(m) = nom(op)$ et seul ce nom est ensuite utilisé pour dénoter l'opération ou le module.

5. Les types

Avant de passer aux classes, nous rappelons quelques définitions nécessaires sur les types, pour justifier par la suite le statut particulier donné aux classes. Dans la pratique, il est courant de tout ramener à des types. Cette manière de procéder est criticable lorsqu'il s'agit d'exposer clairement la démarche de conception utilisée.

Définition 5.21 - Type : Un type est défini par induction à partir de types simples. Un type simple est, soit le type *integer, character, float, boolean*, soit un type enuméré. Un aggrégat de types simples distincts est un type. Un tableau d'un type simple donné est un type. Un ensemble d'un type simple donné est un type. Une liste d'un type simple est un type.

Définition 5.22 - Constructeur de type : Un constructeur de type est défini soit par un aggrégat, un tableau, un ensemble ou une liste.

Un constructeur de type appliqué à un type définit un nouveau type.

Un type $t \in T$ est dénoté par un nom noté $nom(t)$. Par abus de langage on pose $nom(t) = t$. Un type définit un domaine de définition de valeurs D_t et un ensemble d'opérations O_t s'appliquant à des variables typées par le type t. Les opérations pouvant être définies sont :

- l'affectation
- les opérateurs de relation $(=, <, \neq, \ldots)$
- les opérateurs $(\times, div, mod, \ldots)$
- des procédures et des fonctions quelconques utilisant le type t.

Définition 5.23 - Contrainte : Une contrainte sur un type t est définie par une restriction de l'ensemble de définition D_t. Une contrainte est exprimée soit par :
- un intervalle de valeurs d'un type simple
- un choix dans un agrégat multiple
- un intervalle de valeurs d'un type simple d'un intervalle indéfini d'un tableau.

Définition 5.24 - Sous type : Un sous-type $st \in T$ d'un type t est défini par la donnée d'une contrainte sur le domaine D_t de t.

Le fait de considérer les classes comme des types et l'héritage comme une relation de sous-typage ne fait pas l'unanimité. Le lecteur intéressé peut se reporter à [PAL 91]. Nous conserverons donc le concept de typage pour les structures de données classiques à l'exception des classes.

Définition 5.25 - Généralisation d'une contrainte sur un type : Une contrainte est un prédicat défini sur l'ensemble des valeurs D_t définies par le type t.

Exemple :

```
type Case is record
    Lettre : A..H ;
    Chiffre : 1..8 ;
end record;
type Plateau is array(<>) of Case ;
subtype Echiquier is Plateau (1..8,1..8) ;
subtype Diagonale is Echiquier;
    where Rang (Case.Lettre) = Rang (Case.Chiffre);
    end where;
```

Nota : La syntaxe utilisée ci-dessus n'est pas relative à un langage de programmation existant.

Un sous-type ne définit pas un nouveau type. Cela autorise d'affecter entres elles des variables d'un type et d'un sous-type (sans restriction de sens, à condition de respecter la contrainte fixée par le sous type).

Le concept de type étendu est facile a appréhender. C'est la démarche de spécialisation qui est privilégiée dans certains langages comme Oberon 2, Ada9x et Object Pascal ou Think Pascal. Dans Ada9X l'adjonction de l'héritage au travers des types étendus, prolonge le concept de types dérivés qui existait déjà [TUC 92]. Ce concept est celui qui correspond intuitivement le plus à une réunion de types. A l'instar du sous-typage, les types dérivés définissent de nouveaux types.

Exemple : (utilisation de Ada9x)

```
type Basic_Window is tagged record
        Id : Window_Id ;
        Lower_Left : Point ;
        Lower_Right : Point ;
end record;
type Fancy_Window is new Basic_Window
        with record
                Border_Width : Pixel_Count := 1 ;
                Border_Color : Color := Black ;
        end record ;
```

6. Les classes

De façon similaire aux modules nous définissons un constructeur de classes à partir de la notion de méta-classe. D'où la définition suivante.

$$\begin{array}{rccc} \zeta: & [\![C]\!] & \rightarrow & C \\ & \hat{c} & \mapsto & \zeta(\hat{c}) = c \end{array}$$

L'ensemble C définit l'ensemble des classes. Sur cet ensemble on récupère les relations induites *use* et $\succ$. Par abus de langage nous conserverons les mêmes notations. Des opérations de constructions d'objets sont définies à leur tour. Il est à

remarquer que nous pouvons également considérer la définition de classe générique. La relation $\succ$ est transitive, non réflexive et non-symétrique.

Propriétés :

- $\forall c \in C, c \not\succ c$
- $\forall (c_1, c_2, c_3) \in C^3, c_1 \succ c_2 \ \wedge \ c_2 \succ c_3 \Rightarrow c_1 \succ c_3$
- $\forall (c_1, c_2) \in C^2, c_1 \neq c_2 \ \wedge \ c_1 \succ c_2 \Rightarrow c_2 \not\succ c_1$

Si on considère une classe dérivée c_1 comme un sous-type de sa classe parente c_2, nous avons la possibilité d'affecter une valeur de type c_1 à une variable de type c_2 (l'inverse n'est pas autorisé). Autrement dit, si $< o_1, c_1 > \ \wedge \ < o_2, c_2 >$ alors l'affectation $o_2 \leftarrow o_1$ est autorisée tandis que $o_1 \leftarrow o_2$ ne l'est pas.
Nous constatons que l'ajout d'un champ ou la redéfinition d'un attribut avec un intervalle de valeurs plus grand appararaît, cette fois ci comme une extension et non comme une restriction du domaine de définition.

7. Schéma général

Le fondement de notre démarche méthodologique se trouve dans l'utilisation du concept de base bâti sur la notion de TAD. Par la suite, l'important est la distinction faite entre les modules et classes et dans une moindre mesure avec les types. Distinguer les objets issus de modules de ceux issus de classes, nous semble être fondamental. Le schèma suivant résume le contexte que nous avons retenu.

$$
\begin{array}{ccccccc}
A & \xrightarrow{\tilde{\rho}} & [\![M]\!] & \xrightarrow{\tilde{\phi}} & [M] & \xrightarrow{\tilde{\gamma}} & M \times \cdots \times M \\
 & & & \searrow \tilde{\zeta} & \downarrow \tilde{\gamma_i} & & \\
 & \searrow \rho & & & M & \xrightarrow{\tilde{\theta}} & \tilde{O_b} \\
 & & [\![C]\!] & \xrightarrow{\phi} & [C] & \xrightarrow{\gamma} & C \times \cdots \times C \\
 & & & \searrow \zeta & \downarrow \gamma_i & & \\
 & & & & C & \xrightarrow{\theta} & O_b \times \cdots \times O_b \\
 & & & & \downarrow \theta_i & & \\
 & & & & O_b & &
\end{array}
$$

8. Utilisations des modules, des classes et des objets

La phase de conception relative à la définition de l'architecture logicielle d'une application se passe géréralement sans évoquer de déclaration de variable. Cependant, la justification de l'utilisation d'une démarche est plus aisée à faire lorsqu'on évoque les implications d'un choix au niveau de la phase de codage. Ainsi, pour percevoir les analogies ou les distinctions faites entre les objets construits à partir d'un module ou d'une classe nous allons donner un exemple après avoir rappelé quelques définitions.

Définition 8.26 - Visibilité d'un nom : Soit N un ensemble de noms dénotant des entités dans un environnement E donné. On note $N_E^{l_i}$ l'ensemble des noms visibles

lequel nous reviendrons, car la place nous manque ici, concerne les différentes interprétations ou conversions d'un objet. Les objets multi-aspects peuvent être spécifiés formellement en utilisant par exemple VDM [JON 92].

Exemple :

norm-temp : (Fahrenheit ∪ Celsius) → Celsius
norm-temp(t) ≜ **if** t ∈ Fahrenheit **then**
let mk-Fahrenheit(v) = t **in**
mk-Celsius ((v-32)*5/9)
else t

Déterminer la classification la plus adéquate pour représenter un objet, par exemple une température, constitue le genre de questions pour lesquelles nous cherchons des réponses satisfaisantes.

Définition 8.31 - Déclaration de variable : La déclaration d'une variable peut se noter : UneVariable : NomdeType ; Un module n'étant pas un type il n'est pas possible de déclarer , MonModule : NomModule ; En revanche, une classe étant une *sorte* de type, nous pouvons déclarer : NomObjet1, NomObjet2 : NomDeClasse ;

La déclaration d'un module s'effectue implicitement – structure syntaxique – comme d'ailleurs celle d'une classe, lorsque la notion de méta-classe n'est pas accessible à l'utilisateur. La création d'un module serveur de données introduit automatiquement et implicitement un objet associé. Un fois, un objet créé, nous constatons une similitude d'accès dans les deux cas – module ou classe –. L'exemple donné ci-après illustre cette situation.

Notation pointée : L'utilisation d'éléments d'un module ou d'un objet est notée Nom_Du_Module.Nom_de_l'élément.

Exemple :
Module AVION **is** ..... **end Module** ; – Déclaration d'un module. **Classe** AVION (VEHICULE_AERIEN)**is** ..... **end Classe** ; – Déclaration d'une classe. Airbus : AVION ; – Déclaration d'un objet de la classe AVION. Airbus.vole ; – envoi d'un message *vole* à l'objet Airbus. AVION.vole(Airbus) ; – appel de la procédure *vole* du module AVION avec le paramètre Airbus. **Module** AVION [Modele] **is** .....**end Module** ; – Déclaration d'un module générique. **Module** Airbus **is** AVION [Modele_Airbus] ; – Déclaration d'un module particulier. Ce qui pourrait tout aussi bien s'écrire sous la forme : Airbus : AVION [Modele_Airbus] ; Nous obtenons alors : Airbus.vole ; – Appel de la procédure *vole* du module Airbus. Ce qui est similaire à l'expression sur les classes écrite plus haut.

9. Conclusion

Nous avons essayé de présenter les concepts de notre méthode basée sur une approche par les types abstraits de données. Adoptant le point de vue de plusieurs auteurs sur la question, nous avons présenté des définitions permettant de séparer clairement les constituants essentiels d'une conception orientée objet. Chaque notion identifiée sert de modèle de spécification à un outil en cours de développement. Ce travail constitue une étape préliminaire. Il sera par la suite complété par l'énoncé de régles

à partir de la location l_i.

Définition 8.27 - Variable : Soit V_D un ensemble de variables relatives à un domaine D et T_D un ensemble de types pour ce même domaine. L'ensemble $Decla$ des déclarations est défini par un sous-ensemble du produit cartésien $V_D \times T_D$.

Notation : On note une déclaration par un couple $< x, t >$. Une déclaration n'est valide que si $t \in N_E^{l_i}$ où l_i est le lieu du point de déclaration. Une déclaration permet d'établir un lien entre un nom et une variable ainsi qu'un domaine de définition par l'intermédiaire du type t. On dit que la variable x est typée par le type t. Par abus de langage on pose $nom(x) = x$.

Définition 8.28 - Valeur typée : On appelle valeur v d'un type donné t une instance de t. Une instance d'un type t est défini par induction sur les types. Nous nous contenterons de donner ici quelques exemples.
- Si $t = integer$ alors $\{v, v \in V_t\} = -Maxint..Maxint = I$,
- Si $t = tuple(ch_1 : integer, ch_2 : integer, ch_3 : integer)$ alors $\{v, v \in V_t\} = I \times A \times I$.

Définition 8.29 - Association valeur/variable : Soit une déclaration $< x, t >$ et v_t^i une valeur de type t. L'initialisation (où l'affectation) d'une variable par une valeur est définie par :
précondition : $(val(x) = \#)\{x \leftarrow v_t^i\}$ postcondition : $(val(x) = v_t^i)$.
Le symbole $\#$ désigne la valeur indéfinie. Plus généralement on définit l'affectation de deux variables par :

$$x \leftarrow y \stackrel{def}{\Longleftrightarrow} type(x) \doteq type(y) \ \wedge \ x \leftarrow val(y)$$

L'équalité $\doteq$ entre types est sujette à une sémantique particulière selon que l'on utilise des types, des sous-types, des types dérivés, des extensions de type ou des types restreints (limités privés).

Lors de la phase de conception, le concepteur pourra déterminer une hiérarchie de modules et une hiérarchie de classes. La seule *passerelle* autorisée entre les deux mondes est définie par la relation donnée ci-dessous.

Définition 8.30 - Relation d'inclusion d'une classe dans un module : On dit qu'une classe c de C est incluse dans un module m si :
$m \in M, c \in C, c \subset m \stackrel{def}{\Longleftrightarrow} (S_m = \emptyset) \ \wedge \ N_E^{l_c} \subset U_m \ \wedge \ nom(c) \in N_E^{l_{op_m^i}}, \forall op_m^i \in O_m$

En plus de la connaissance précise des entités manipulées, l'information pertinente d'une conception globale se trouve dans l'expression des relations. Les principales relations à considérer sont désormais classiquement :

- La relation IS_A ou IS_KIND_OF entre classes,
- La relation INSTANCE_OF entre classes et méta-classes ou entre objets et classes,
- La relation PART_OF entre classes – composition –,
- La compatibilité entre objets – projet TAME [OIV 92] –.

La description précise de ces relations et surtout l'énoncé des règles justifiant leurs utilisations font l'objet d'un futur papier. Un autre aspect sur

méthodologiques. Pour terminer cet exposé, nous reprendrons les conclusions d'un atelier sur les méthodes orientées objet[OOP 91]. Les bases d'un modéle objet nécessitent une base formelle supportant : - une phase de test du modéle pour déterminer s'il est bien formé, - une formalisation des propriétés du domaine, - la possibilité d'utiliser les langages spécifiques aux applications, - l'agrégation, - le partitionnement en classes et en objets, - une phase de test pour vérifier la complétude et la consistance du modèle, - la validation du modèle par rapport aux systèmes existants. Notre contribution n'a porté que sur une partie de ces aspects. Les points inarbordés font l'objet de travaux supplémentaires qui seront présentés prochainement.

10. Bibliographie.

[ACM 90] *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, San Francisco, California, January 17-19, 1990. Association for Computing Machinery.

[Ada 83] *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD 1815 A, January 1983.

[ACM 92] *Nineteenth Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 19-22, 1992. Association for Computing Machinery.

[BIE 92] Brigitte Biebow, *Formalisation et extensions des graphes conceptuels : Les graphes conceptuels et d'autres modéles de représentation*, actes des 4èmes journées nationales, PRC-GDR, Intelligence Artificielle, Marseille 19-21 octobre 1992, Ed. Teknea.

[BOO 92] Grady Booch, *Conception Orientée Objets et applications*, Addison Wesley Publishing Company Inc, 1992.

[CLO 90] Guy L. Steele jr., *Common LISP The Language*, second edition, Digital Press, 1990.

[COO 90] William R. Cook, Walter L. Hill, Peter S. Canning, *Inheritance is not Subtyping*, [ACM 90]

[DIS 82] Robert-Michel Di Scala, *Algèbres formelles informatiques*, Faculté des sciences et techniques, Université François-Rabelais, TOURS.

[ECO 92] *ECCOP'92, European Conference on Object Oriented Programming*, Utrecht, The Netherlands, jue/July 1992, Proceedings, Lectures notes in Computer Science 615, Springer Verlag.

[GIR 90] Xavier Girod, *MECANO a method for Object-Oriented Software Construction*, Proceedings of the TOOL'S Conference, Paris 1990, p.145.

[GOG 87-a] Joseph A. Goguen, José Meseguer, *Unifying functional, Object-Oriented and Relational Programming with Logical Semantics*, Peter Wegner and Bruce Shriver, Editors, Research directions in object-oriented programming, Computers Systems Series, MIT Press, 1987.

[GOG 87-b] Joseph A. Goguen, David Wolfram, *On Types and FOOPS*

[GRA 90] Justin O. Graver, Ralf E. Johnson, *A type system for SmallTalk*, [ACM 90]

[GRE 90] Eric Grégoire, *Logiques non monotones et intelligence artificielle*, Edition Hermès, Paris 1990.

[GUT 78] J. Guttag, J. J. Horning, *The Algebraic Specification of Abstract Data Types*, Acta Informatica, vol 10, pp 27-52, 1978.

[HAY 91] Fiona Hayes, Derek Coleman, *Coherent models for object-oriented analysis*, [OOP 91]

[HOO 91] *Hood Reference Manual*, Hood Users Group, Contact. F. Hass, CRI, Denemark.

[JON 92] Cliff B. Jones, *VDM : une méthode rigoureuse pour le développement du logiciel*, traduction de Michel Lemoine, Masson 1992.

[LEL 92]Ilog SA, *LE-LISP Version 16, Reference Manual*, Gentilly, France.

[MAC 90] R. Mac Gregor, *The evolving technology of classification-based knowledge representation systems*, in principle of semantic networks : Explorations in the representation of knowledge, john Sowa Ed., San Mateo, CA: Morgan Kaufmann,1990.

[MAR 92] William Mark, Sherman Tyler, James Mc Guire, and Jon Schlogsberg, *Commitment-Based Software Development*, IEEE transactions on software engineering, vol 18, N 10, october 1992.

[MEY 90] Bertrand Meyer, *Conception et programmation par objets pour du logiciel de qualité*, Interéditions, 1990.

[MIT 90] John C. Mitchell, *Toward a typed foundation for method specialization and inheritance*, [ACM 90]

[MOS 92] Hanspeter Mössenböck, *The Oberon System*, Tutorial of Seventh International Conference and Exhibition, Dortmund - Germany, TOOLS Europe'92, march 31 - april 92.

[NER 92] Jean-Marc Nerson, *Object Oriented Analysis and Design*, Tutorial, TOOL'S 92 [TOO 92].

[OIV 92] Markku Oivo, Victor R.Basili, *Representing Software Engineering Models : the TAME Goal Oriented Approach*, IEEE transactions on software engineering, vol 18, N 10, october 1992.

[OMT 92] Michael Blaha, Frederick Eddy, *Object-Oriented Modelling and Design*, Utrecht, The Netherlands, june 29-july 3, 1992, ECOOP'92, Tutorial T2.

[OOP 91] *Conference on Object-Oriented Programming Systems, Languages, and Aplications*, OOPSLA'9, Conference Proceedings, Edited by A. Paepcke, 6-11 october 91, Phœnix Arizona, ACM Press, and *Object Oriented Modelling Workshop*, Addendum to the Proceedings, OOPSLA'91, 9-11 october 91 , Phœnix Arizona, Ed. Jerry L. Archibald, OOPS Messenger vol. 3, number 4, october 92.

[PAL 91] Jens Palsberg, Michæl I. Schwartzbach, *Three Discussions on Object-Oriented Typing*, Report on ECOOP'91, Workshop W5.

[ROS 92] Jean-Pierre Rosen, *What Orientation Should Ada Objects Take ?*, Communications of the ACM, november 1992, volume 35, number 11.

[ROY 92] JC. Royer, *A New Set Interpretation of the Inheritance Relation and its Checking*, OOPS Messenger, volume 3, July 1992.

[SCH 91] Jens Palsberg, Michael I. Schwartzbach, *Object-Oriented Type Inference*, [OOP 91]

[SCH 92] Harald Schaschinger, *ESA : An Expert Supported OOA Method and Tool*, ACM SigSoft, Sotfware Engineering Notes, vol. 17, number 2, april 92, page 50-56.

[SED] *SEDL : Software Engineering Design Language*, IBM.

[SHE 92] Jun Shen, Gordon V. Cormack, *On Generic Formal Package Parameters in Ada 9X*, Ada letters, may/june 1992 , page 110, volume XII, number 2, ACM SigAda.

[STR 87] B. Stroustrup, *C++*, Addison-Wesley, 1987.

[SZY 92] Clemens A. Szyperski, *Import is Not Inheritance, Why We Need Both : Modules and Classes*, Proceedings ECOOP'92 European Conference on Object-Oriented Programming, Utrecht, The Netherlands, June/July 1992, Editor O.Lehrmann Mandsen, Springer-Verlag.

[TOF 92] Mads Tofte, *Principal signatures for higher-order Program modules*, [ACM 92]

[TOO 92] *Technology of Object Oriented Languages ans Systems* - Proceedings of the Seventh International Conference, TOOL'S 92, Dortmund, Germany, March 30 - April 2, 1992.

[TUC 92] S. TuckerTaft, *Ada 9X*, Communications of the ACM, november 1992, volume 35, number 11.

[TYS 92] S. Tyszberowicz, A. Yehudau, *OBSERV - A prototyping Language and Environment*, ACM Transactions on Software Engineering and Methodology, volume 1, number 3, july 1992, page 269-309.

[WIR 79] Niklaus Wirth, *The module : a system structuring facility in high-level programming languages*, Lestures Notes in Computer Sciences, Language Design and Programming Methodology, Proceedings of a Symposium, Sydney, Australia, Sep. 79, Springer Verlag.

LE LABORATOIRE D'INFORMATIQUE THEORIQUE ET D'APPLICATIONS DE MARSEILLE ... EN QUELQUES MOTS

BUTS:
Le but du LITAM est le développement de la recherche fondamentale ou appliquée, concernant les propriétés fondamentales de l'informatique. Les interactions entre les domaines du matériel et logiciel sont au coeur de cette tâche.

THEMES DE RECHERCHE:
Les applications sont réalisées en utilisant les résultats obtenus par l'élaboration de théories. Diverses constructions sont entreprises dans les domaines suivants:

(EQUIPE DE MONSIEUR LE PROFESSEUR E.BIANCO)
La conception des machines et des circuits:
-application des propriétés fondamentales de l'informatique à la conception des processeurs;
-langages des machines théoriques adaptés à l'étude des propriétés fondamentales de l'informatique.
Les automates finis:
-utilisation dans la compilation et les systèmes autojectifs;
-dictionnaires électroniques: représentations d'ensembles en machines.

(EQUIPE DE MONSIEUR LE PROFESSEUR R.CUSIN)
Systèmes d'aide à la décision multicritères et multiobjectifs:
-méthodes interactives;
-méthodes relevant des problématiques de tri et de rangement.
Systèmes dynamiques:
-modèle dérivé de l'application dite " logistique ";
-systèmes non linéaires: formes biologiques.

EFFECTIFS:
L'équipe permanente, composée de dix membres enseignants-chercheurs des universités françaises, est dirigée par Monsieur le Professeur E.BIANCO. Le groupe de chercheurs en formation doctorale est actuellement d'une dizaine de personnes sous la responsabilité de Monsieur le Professeur R.CUSIN.

Activités et publications:

En permanence:

Groupe de Formation Doctorale d'Informatique Fondamentale et Sciences de la Computation

Formation à la recherche des jeunes

Tous les mois:

Séminaire de recherche du Laboratoire

Un membre de l'équipe expose un thème de travail ou d'information

Tous les 3 mois:

Bulletin d'informatique approfondie et applications. ISSN 0291-5413 MARSEILLE

Articles scientifiques soumis à comité de lecture, éditorial, notes, annonces

Tous les 6 mois:

Séminaire International du Laboratoire

Un Professeur étranger est invité pour exposer ses travaux

Tous les ans:

Thèses du Groupe Formation Doctorale

Présentation des thèses soutenues dans le cadre de l'équipe doctorale

Tous les 2 ans:

Journées Internationales d'Informatique Fondamentale de Marseille

Communications entre des Universités internationales: Annaba (Algérie), Bonn (Allemagne), Homs (Syrie), Moncton (Canada), Pierre et Marie Curie - Paris (France), Santiago (Chili), etc ...

Université de Provence
Atelier de Reprographie
Centre Saint Charles
3, place Victor Hugo
F - 13331 Marseille Cedex 3