Conception par objets : MECANO, une Méthode et un Environnement de Construction d'ApplicatioNs par Objets

Xavier Girod

THESE

Présentée par

Xavier Girod

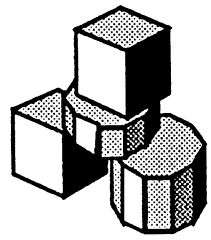
Pour obtenir le titre de

Docteur de L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

(Arrêté Ministériel du 5 Juillet 1984)

Spécialité Informatique

CONCEPTION PAR OBJETS



MECANO:
une Méthode et un Environnement de Construction d'ApplicatioNs par Objets

---

Date de Soutenance : **le 21 juin 1991**

Composition du Jury:

S. KRAKOWIAK
J. BEZIVIN
P. COINTE
B. MEYER
J. ESTUBLIER

Thèse préparée au sein du Laboratoire de Génie Informatique

A Claire et Johan

Je tiens à remercier

Monsieur Sacha Krakowiak, Professeur à l'Université Joseph Fourier de Grenoble, pour l'honneur qu'il me fait de présider le jury de cette thèse. Il fut un de mes enseignants en maîtrise d'informatique et je salue en lui le chercheur et le pédagogue exceptionnels,

Monsieur Jean Bezivin, Professeur à l'Université de Nantes, pour le soutient qu'il m'a toujours témoigné en particulier lors de notre première rencontre à TOOLS'89. Qu'il soit chaleureusement remercié pour avoir bien voulu rapporter ce travail, ses remarques et ses conseils ont largement contribués à l'amélioration de ce document,

Monsieur Pierre Cointe, Directeur de l'équipe mixte Rank-Xerox/LITP, qui a spontanément accepté de juger ce travail et m'a encouragé par ses conseils amicaux,

Monsieur Bertrand Meyer, Président de Interactive Software Engineering Inc., à Santa Barbara, Californie, Professeur associé à l'Université de Santa Barbara. Il est le père d'EIFFEL, langage qui fut à la source de mon travail. Je tiens à le remercier pour avoir accepté de participer à ce jury malgré son activité qui l'appelle aux quatre coins du monde,

Monsieur Jacky Estublier, Chargé de Recherche au CNRS, qui est le directeur de cette thèse, pour m'avoir accueilli au sein de son équipe et pour la confiance totale qu'il m'a toujours accordée,

Philippe Morat, Maître de Conférence à l'Université Joseph Fourier, collègue et ami depuis maintenant 6 ans. Ce travail est le fruit d'une collaboration étroite, je le remercie chaleureusement de son dévouement et de son amitié inébranlables, qu'il soit assuré de la réciprocité de mes sentiments,

Farid Ouabdelselam, Maître de Conférence à l'Université Joseph Fourier, qui m'a toujours soutenu depuis mon DEA. Je tiens à le remercier pour sa relecture scrupuleuse et les conseils qu'il m'a prodigués,

Pierre Claude Scholl, Professeur à l'Université Joseph Fourier, Directeur de l'UFR d'informatique de Grenoble, pour son soutient pendant ces quatre années où j'ai découvert et apprécié le métier d'enseignant, et pour m'avoir accueilli dans son équipe lors de ma première année de recherche en DEA, la confiance qu'il m'a toujours accordé m'honore,

Je remercie mes parents, Thérèse et Paul, pour leur soutient et l'intérêt qu'ils ont toujours porté à l'avancement de mes travaux, qu'ils reçoivent toute mon affection,

Je voudrais terminer en dédiant ce travail à Claire Malverti, ma compagne depuis 13 ans, qui a su me supporter avec une patience et une affection de tous les instants. Ce travail, qu'elle a relu et corrigé, ne serait pas sans elle, je la remercie tendrement,

Je dédie aussi ce travail à Johan, qui a eu l'idée saugrenue de venir au monde quelques jours avant la soutenance de la thèse de son Papa,

TABLE DES MATIERES RESUMEE

PARTIE 1:
L'APPROCHE OBJET POUR LA CONSTRUCTION DES LOGICIELS

CHAPITRE1
INTRODUCTION: CONCEPTION, MODELE OBJET ET QUALITE

CHAPITRE 2
CONCEPTION PAR OBJET

PARTIE 2 : MECANO, METHODE ET ENVIRONNEMENT

CHAPITRE 3
LA METHODE MECANO

CHAPITRE 4
CONCEVOIR AVEC MECANO

CHAPITRE 5
LE POSTE DE CONCEPTION MECANO

ANNEXES ET BIBLIOGRAPHIE

ANNEXE A
PASSAGE DES TYPES ABSTRAITS AUX OBJETS: EXEMPLE

ANNEXE B
SYMBOLES UTILISES EN MECANO

ANNEXE C
MENUS ET OPERATIONS SUR LES ENTITES

ANNEXE D
CONCEPTION DU POSTE: EXEMPLES

BIBLIOGRAPHIE

CHAPITRE 1

INTRODUCTION: CONCEPTION, MODELE OBJET ET QUALITE

Objet, définition du Larousse encyclopédique:
Ce qui se présente à la vue...
Chose matérielle façonnée en vue d'un usage précis...
Monde extérieur, connu immédiatement par nos sens...

LE PARADIGME OBJET

0. PRESENTATION DU RAPPORT

0.1. Description de l'étude

Ce travail propose une méthode de conception pour les applications développées avec le modèle "objet". Cette méthode, baptisée "MECANO", s'appuie sur le modèle objet présent dans la plupart des langages de programmation et propose une extension de ce modèle pour prendre en compte les problèmes spécifiques de la conception par objet.
Afin de préciser la nature de cette méthode, nous situons dès à présent notre travail en précisant les buts et apports de la méthode.

Ce que n'est pas MECANO:

• La méthode que nous proposons n'est pas une méthode d'analyse. En ce sens que nous ne donnons pas de principes pour établir une relation entre les objets du monde réel et les objets du modèle d'analyse. Le lecteur est renvoyé aux méthodes d'analyse orienté-objet que l'on peut trouver dans la littérature [Shlaer 88], [Coad 91], [Rumbaugh 91].
• MECANO n'est pas une méthode pour une implantation dans un langage classique. Nous pensons, et ce choix sera discuté dans le rapport, qu'une vraie conception par objets doit déboucher sur une implantation dans un langage orienté-objet.
• MECANO n'est pas une méthode pour des applications exigeant un typage faible et des langages dynamiques. Notre fil directeur est le génie logiciel; nous nous attachons à décrire (et à vérifier) le maximum d'informations le plus tôt possible dans la conception. Les langages candidats à l'implantation sont donc des langages typés statiquement.
• MECANO n'est pas un nouveau langage orienté-objet. La méthode s'intéresse uniquement à la phase de conception. Un formalisme essentiellement graphique est proposé. L'outil associé à la méthode, s'il peut générer des squelettes de code dans un langage particulier, n'impose pas de contraintes fortes (autres que celles de conception) au concepteur. Il est capable de gérer les incohérences passagères inhérentes à toute activité créatrice, et la conception de logiciels en est une.

Ce qu'est MECANO:

• C'est une méthode de conception bâtie essentiellement sur les concepts du langage EIFFEL: objet/classe, classe virtuelle, relation client, assertions, généricité contrainte, déclaration d'export, héritage multiple.
• MECANO ajoute aux concepts du langage EIFFEL de nouveaux concepts: taxonomies de l'héritage et de la délégation, structuration des classes en composites, domaines et applications.
• On privilégie une approche "type abstrait" des classes.
• MECANO apporte des réponses concrètes au problème de la réutilisabilité et de l'évolutivité.
• MECANO est une méthode et un environnement de conception.

Tous ces points seront longuement développés dans la suite du rapport.

Ce travail a été développé dans le cadre d'un contrat avec la société HEWLETT-PACKARD (Eybens).

0.2. Plan du rapport

Ce rapport est divisé en deux parties.

Partie 1:
Elle est consacrée à l'étude du modèle objet et des méthodes de conception basées sur ce modèle.

Dans ce premier chapitre, nous posons la problématique de la qualité des logiciels et nous présentons les approches "classiques" de la conception. Le lecteur familiarisé avec le génie logiciel peut passer directement au §2..
La première partie du §2. donne un panorama des langages objets et présente les concepts du paradigme objet. La deuxième partie effectue un rappel sur les types abstraits et montre le lien étroit existant entre le modèle des types abstraits et le paradigme objet. Les lecteurs connaissant les types abstraits et les concepts basiques du modèle objet pourront passer à la suite.
Nous discutons à la fin de ce chapitre des apports de l'approche objet sur la qualité du logiciel.

Dans le chapitre 2, nous analysons les différentes méthodes orienté-objet. Ceci nous conduit à dégager des besoins pour une nouvelle méthode dédiée à la conception par objets. Les lecteurs familiers des méthodes d'analyse et de conception par objets pourront lire directement la partie synthèse.

Partie 2:
Elle est consacrée à notre travail proprement dit.
Le chapitre 3 expose la méthode MECANO. Nous donnons les concepts et les règles de construction des entités de la méthode.
Le chapitre 4 décrit la mise en oeuvre de la méthode. Le formalisme et le processus de conception sont exposés. Un exemple complet est traité afin d'illustrer la méthode.
Le chapitre 5 concerne l'environnement de gestion de la méthode appelé "Poste de Conception MECANO". La philosophie du poste et ses fonctionnalités sont développées.

1. LA QUALITE DU LOGICIEL

"Le logiciel est en crise!" Ce n'est pas une découverte, et la majorité des communications traitant du génie logiciel depuis vingt ans commencent par énoncer cet état de fait. On peut cependant être surpris par la continuité de cette crise et par le peu d'évolution concernant le processus de création de logiciels. Quand on compare cette évolution aux progrès du matériel (progression géométrique), on prend la mesure du fossé qui s'est créé. La question que l'on peut se poser est la suivante "*Pourquoi le logiciel n'évolue-t-il pas aussi vite et aussi bien que le matériel?*" ou ce qui revient au même: "*Pourquoi les logiciels construits en 1991 sont-ils toujours aussi coûteux à développer et d'aussi mauvaise qualité que ceux développés il y a 20 ans?*" On peut trouver diverses causes à cet état de fait liées à la nature logique des programmes: pas d'effet de "série", pas de vieillissement physique mais des détériorations successives (il est plus facile de modifier un programme qu'un circuit). Il semble cependant que d'autres raisons soient mises en cause: mauvaises habitudes, mauvaises méthodes, mauvais outils. On citera à titre d'exemple la part des coûts de maintenance dans le développement logiciel [Lientz 80]. On estime que si le coût d'une erreur est de 1 quand cette erreur est détectée pendant la phase de définition, il est multiplié par 1,5 à 6 pendant la phase de développement et par 60 à 100 si l'erreur n'est détectée qu'à la maintenance. On considère que la maintenance consomme actuellement 70% des ressources d'un projet.

Dans la figure 1, cette part a été éclatée en différentes activités et causes: on s'aperçoit que les changements de besoins des utilisateurs consomment près de 42% de la maintenance c'est à dire 29% du coût total. Deux facteurs entrent en jeu. Le premier vient de la demande des utilisateurs qui devient forte au fur et à mesure de l'utilisation d'un logiciel. Mais ce facteur n'agit pas sur les coûts puisqu'on peut penser qu'une évolution, à partir du moment où elle est clairement exprimée donne lieu à une révision du coût global en accord avec l'utilisateur. Le deuxième facteur, plus significatif, est sans doute qu'il est très difficile de faire évoluer les logiciels tels qu'ils sont construits à l'heure actuelle.
Une autre part importante du coût de la maintenance vient des changements dans les formats de données: 17% de la maintenance soit 12% du coût total. Deux causes peuvent être dégagées: les données ne sont pas suffisamment abstraites et les accès sont trop dispersés dans les logiciels classiques. Enfin on remarquera le peu d'incidence de l'optimisation des programmes aussi bien en temps qu'en mémoire (4%), ces problèmes devenant de moins en moins important grâce à l'évolution du matériel.

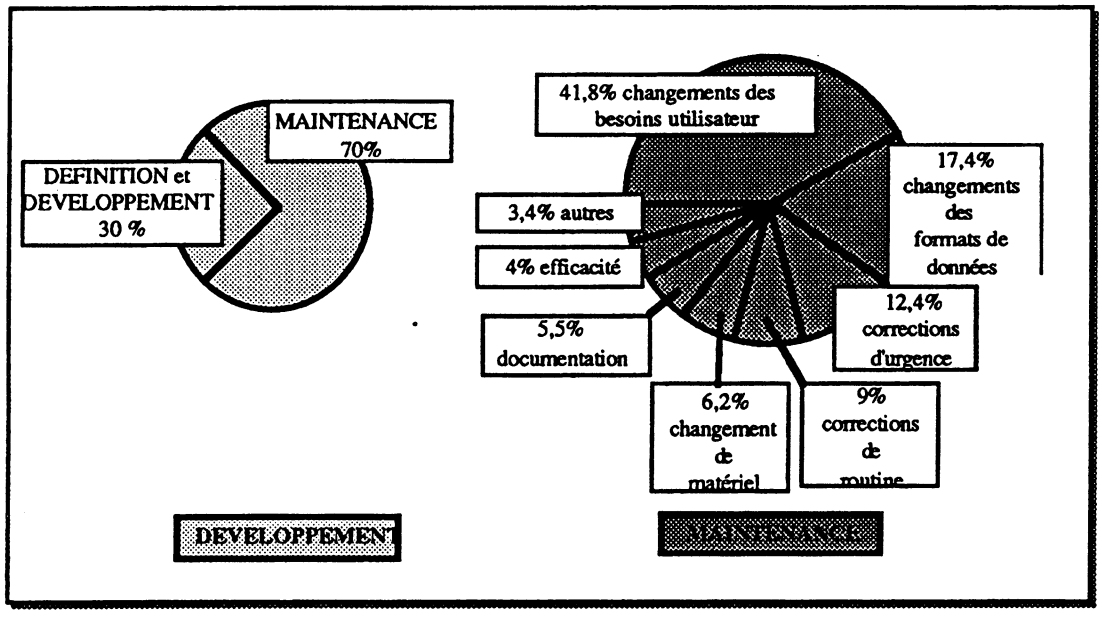


figure 1 : coût de développement et de maintenance

L'approche par objets peut être considérée comme une évolution importante[1] pour la conception d'applications logicielles. En effet, la conception par objets et le développement du matériel ont en commun une très forte réutilisation des composants, ce qui permet de gagner en temps et en qualité. Tous deux s'appuient sur un processus ascendant dans lequel des composants élémentaires sont associés pour construire des composants plus riches et plus complexes [Cox 89].

1.1. Introduction : les deux points de vue de la qualité

Tous les indicateurs de la crise du logiciel reviennent au même problème central : la qualité du logiciel. Pour résoudre la crise, il faut utiliser des méthodes et des outils permettant de produire des logiciels de qualité : c'est l'objet du génie logiciel. Les études de McCall [McCall 77] sur la qualité sont largement citées [Pressman 87], [Meyer 88],

[1] "Object Oriented Programming: An Evolutionary Approach" est le titre du livre de B.J. Cox [Cox 86].

[Adam 87]. McCall considère la qualité du logiciel de deux points de vue différents: l'approche de l'utilisateur et l'approche du concepteur.

• **Le point de vue de l'utilisateur (externe)**
L'utilisateur est l'acheteur du système ou au moins une personne qui interagit avec le système. Par extension, une personne formulant une évolution sera considérée comme utilisateur.
L'utilisateur exige et constate des **FACTEURS**[1] de QUALITE EXTERNE. C'est à dire des qualités qu'il peut mesurer ou qualifier d'un point de vue externe au logiciel.

• **Le point de vue du concepteur (interne)**
Le concepteur est la personne qui conçoit, réalise ou maintient le produit.
Il s'efforce de réaliser un logiciel satisfaisant des **CRITERES**[2] de QUALITE INTERNE.

Dans la suite nous allons détaillons ces facteurs et critères en nous basant sur l'étude de McCall. Nous montrerons les corrélations entre les facteurs et les critères.

1.2. Les facteurs de qualité

Les facteurs de qualité peuvent se regrouper en trois grand secteurs (figure 2):

• les facteurs liés aux qualités opérationnelles du logiciel. Ces facteurs sont intrinsèquement liés au logiciel et ne dépendent que de lui. Il s'agit de l'exactitude, la robustesse, l'efficacité, l'intégrité et la convivialité.

• les facteurs favorisant l'adaptation au changement et à la correction: ils sont plutôt associés à la maintenance corrective ou évolutive. Ce sont l'extensibilité, l'adaptabilité et la testabilité.

• Enfin, les facteurs associés à l'adaptation à l'environnement: ils considèrent l'adaptation du logiciel au monde extérieur (autres systèmes informatiques, autres projets). Il s'agit de la réutilisabilité, la portabilité et la compatibilité.

Nous nous attachons à définir cinq facteurs particulièrement importants: l'exactitude, la robustesse, l'adaptabilité, l'extensibilité et la réutilisabilité.

• L'EXACTITUDE : c'est la capacité du logiciel à accomplir correctement les tâches définies par les spécifications externes. Autrement dit: le logiciel final fait-il ce qui a été prévu?

• LA ROBUSTESSE : c'est la capacité du logiciel à réagir correctement à des situations anormales. Les spécifications externes étant souvent floues, ambiguës ou incomplètes, on peut se demander dans quelle mesure le concepteur a prévu ce qui n'était pas prévu, et si le système supporte sans mal des situations non spécifiées au départ.

Ces deux premiers facteurs de qualité opérationnelle sont souvent regroupés sous le vocable de FIABILITE. Un logiciel est fiable s'il réalise les fonctions qu'on lui demande et s'il supporte des situations exceptionnelles sans mettre en cause son intégrité.

---

[1] Facteur, définition du Larousse : Agent, élément qui concourt à un résultat

[2] Critère, définition du Larousse: Principe auquel on se réfère et qui permet de juger, d'estimer. La différence essentielle entre facteur et critère est que ce dernier est mesurable. Des métriques permettent de mesurer les critères de qualité [Adam 88].

• L'ADAPTABILITE : c'est la capacité d'un logiciel à pouvoir être adapté à des changements dans les spécifications. Par changement, on considère les erreurs donnant lieu à des corrections ou des légères modifications. Dans ce cas, il s'agit essentiellement de maintenance corrective.

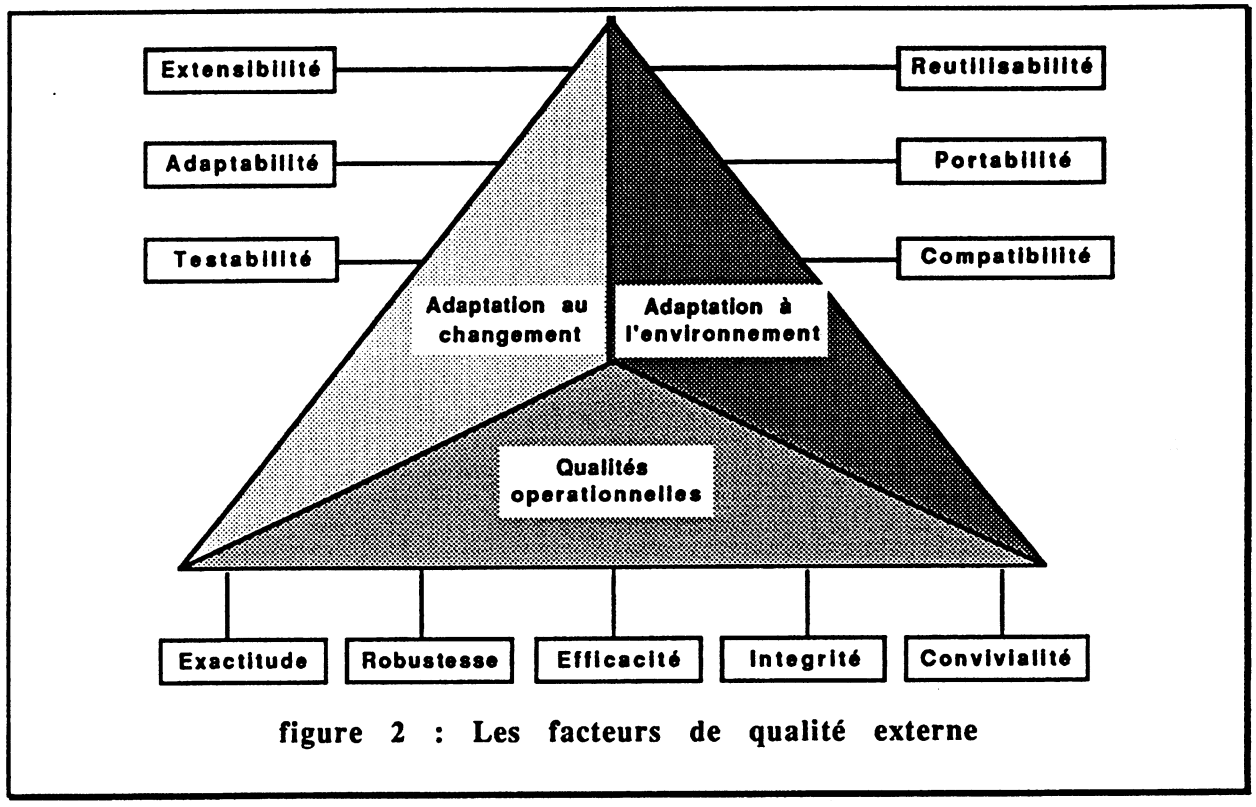


figure 2 : Les facteurs de qualité externe

• L'EXTENSIBILITE[1] : c'est la capacité d'un logiciel à prendre en compte de nouvelles fonctionnalités non prévues au départ. Ce facteur est différent du précédent dans le sens où il correspond à une augmentation des capacités du logiciel. Un logiciel est extensible si on peut facilement lui ajouter des fonctions et services. Ce facteur est très important puisqu'il permet de développer le logiciel de façon incrémentale.

La FLEXIBILITE est un terme générique intégrant les deux facteurs ci-dessus; on dira qu'un logiciel est flexible s'il peut à la fois se plier aux corrections/modifications et aux extensions.

• LA REUTILISABILITE : c'est la capacité d'un logiciel à être réutilisé en tout ou partie pour de nouvelles applications. C'est un facteur clé pour l'industrie du logiciel, en particulier pour la productivité et la qualité. Réutiliser c'est gagner du temps, mais surtout c'est gagner en qualité car pour être réutilisable (et réutilisé) un composant logiciel doit posséder un niveau qualitatif élevé : il doit être validé, robuste, documenté, évolutif ...

Ces facteurs de qualité externe doivent bien sûr être recherchés dans tout développement logiciel. Pour cela, on relie ces facteurs à des critères de qualité interne qui eux sont directement mesurable sur le logiciel. On pourra dire d'un logiciel qu'il possède tel ou tel critère, et par extension on pourra dire d'une méthode de conception de logiciel qu'elle favorise ou pénalise tel critère.

---

[1] ou EVOLUTIVITE selon les auteurs

1.3. Les critères de qualité

Nous présentons ici les critères qui nous semble les plus importants, c'est à dire ceux qui interviennent le plus souvent dans des facteurs de qualité.

• LA COMPLETUDE : c'est le degré d'implantation des spécifications. Un logiciel est complet si toutes les spécifications externes sont opérationnelles.

• LA CONSISTANCE : c'est la possibilité de faire des retours arrière dans le cycle de développement. En particulier de faire remonter une erreur détectée en maintenance au niveau de l'implantation, de la conception voire de l'analyse. Une façon d'obtenir la consistance est d'utiliser des techniques (formalismes) de conception uniformes au cours du développement.

• LA MODULARITE : c'est la décomposition du logiciel en composants facilement appréhendables et relativement indépendants (Cf 1.5.).

• LA GENERALITE : plage d'application potentielle des composants logiciels (Cf 1.6.).

• L'AUTO-DOCUMENTATION : possibilité d'extraction de la documentation depuis les composants logiciels.

1.4. Corrélation entre facteurs et critères

Pour chaque facteur de qualité recherché, on peut donner une liste de critères à atteindre pour le logiciel. La figure 3 représente la fonction de corrélation :

$$CF : \{Critères\}^* \rightarrow \{Facteurs\}$$
$$(c_1, c_2, \ldots, c_n) \rightarrow f$$

si $f = CF(c_1, c_2, \ldots, c_n)$ on dira que le facteur de qualité externe f est atteint en satisfaisant les critères de qualité interne c1,c2,...,cn.

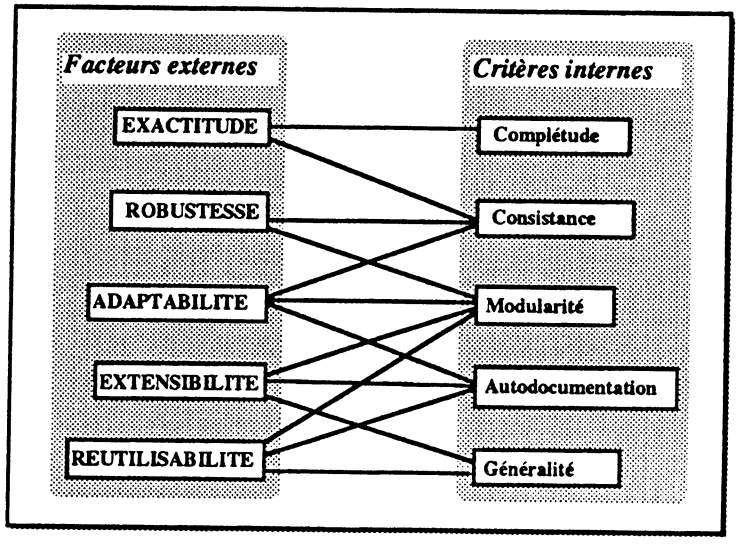


figure 3 : Corrélation entre facteurs et critères

La fonction inverse:

$$FC : \{Facteurs\}^* \rightarrow \{Critères\}$$
$$(f_1, f_2, \ldots f_n) \rightarrow c$$

$c = FC(f_1, f_2, \ldots, f_n)$ signifie que le critère c concourt aux facteurs de qualité $f_1, f_2, \ldots, f_n$.

Il ressort de l'étude de ces fonctions que la modularité est un critère très impliqué dans la qualité logicielle. Ce n'est pas une révolution en soi puisque, depuis Parnas [Parnas 72], on sait qu'une architecture modulaire si elle est bien menée permet de construire des systèmes de façon plus rigoureuse et d'appréhender plus facilement des logiciels de taille importante.
D'autre part, la réutilisabilité et son critère associé la généralité est un facteur important puisqu'elle conduit à abaisser les coûts de revient et à augmenter la fiabilité des logiciels.
Nous reprenons dans la suite l'étude de B. Meyer [Meyer 88] sur ces deux aspects : modularité et généralité.

1.5. La modularité

Le terme de module est suffisamment large pour qu'il recouvre des réalités totalement différentes. En première définition, on peut dire qu'un module est une entité logicielle permettant de découper un programme en morceaux. A partir de là, un découpage en "tranche de saucisson" peut être apparenté à un découpage modulaire bien qu'il conduise à une architecture "spaghetti". Meyer propose un raffinement du critère de modularité en cinq sous-critères. Ces sous-critères devront être satisfaits par une méthode se réclamant de l'approche "modulaire".

• **DECOMPOSITION MODULAIRE**

La méthode doit aider à la décomposition d'un nouveau problème en sous-problèmes dont les solutions peuvent être recherchées séparément. Depuis Parnas, on sait que l'on doit rechercher un COUPLAGE minimum entre les modules, c'est à dire que la structure de connexion doit être la plus simple possible. De même, on s'attache à ce qu'un module ait une forte COHESION, à savoir que sa raison d'être doit facilement être comprise et définie[1]. A ce sujet, l'étude de Stevens [Stevens 74] fournit une taxonomie des critères de cohésion triés par ordre d'importance. Ce critère est absolument nécessaire puisqu'il offre le moyen d'effectuer une décomposition du problème en abstractions successives, ce qu'on appelle généralement conception descendante ou "Top-down"[2].

• **COMPOSITION MODULAIRE**

La méthode doit favoriser la production d'éléments logiciels combinables avec d'autres pour construire de nouveaux éléments. C'est le critère réciproque du précédent: il ne faut pas seulement pouvoir décomposer mais il faut pouvoir composer. Ce critère intervient fortement dans le facteur de réutilisabilité: pour réutiliser, on combine des composants déjà réalisés avec de nouveaux composants, ou bien des composants anciens entre eux. On parle souvent de conception "ascendante" ou "Bottom-up" pour

---

[1] Ce qui élimine d'office la décomposition en "tranches" qui n'ont d'autres raisons d'être que le fait d'avoir été découpées.

[2] Voir la discussion à ce sujet au chapitre 4.

caractériser ce type d'approche qui part d'entités déjà réalisées pour construire de nouvelles entités répondant aux spécifications.

• **COMPREHENSION MODULAIRE**

La méthode doit favoriser la production de modules facilement "appréhendables" séparément par un lecteur. Les modules sont des composants de taille humaine, compréhensibles par une personne. Rejoignant l'idée de couplage minimum, il faut que le module soit lisible indépendamment des autres modules ou avec un nombre réduit d'autres modules[1]. Ce critère favorise le facteur d'adaptabilité - si les composants sont appréhendables séparément les corrections et modifications restent confinées à ce composant et à ses voisins - et d'extensibilité - l'ajout de fonctionnalités et de services peut être localisé et effectué à l'endroit adéquat -.

• **CONTINUITE MODULAIRE**

La fonction :

$$SC : \{Spécifications\} \rightarrow \{Conceptions\}$$
$$s \rightarrow c$$

doit être continue. Cela signifie qu'un petit changement dans les spécifications conduit à un petit changement dans la conception, c'est à dire qu'il n'impacte qu'un seul ou quelques modules seulement. En particulier, l'architecture générale du système ne doit pas être affectée. Dans ce cas, l'extensibilité est améliorée.

• **PROTECTION MODULAIRE**

Une condition anormale à l'exécution d'un module reste confinée à ce module. Ce critère correspond à la prise en compte des opérations non prévues par les spécifications. Il faut que le module puisse de façon autonome et disciplinée gérer les anomalies. Le critère de robustesse est concerné dans ce cas.

A partir de ses sous-critères, on peut mettre en place les principes de la modularité [Meyer 88]. Ces principes, s'ils sont appliqués par une méthode de conception, doivent conduire au respect des sous-critères énoncés.
Le premier principe précise qu'un module doit correspondre à une unité syntaxique dans le formalisme de la méthode (P1: unité modulaire linguistique). Le second recommande un minimum de communication entre les modules, ce qui peut se traduire en disant que dans une conception "le nombre d'interlocuteurs est limité" (P2: peu d'interface). Le principe suivant concerne la taille des interfaces qui doit être minimum: "les dialogues sont limités" (P3: petites interfaces). Le quatrième principe exige que les interfaces soient explicites : "les dialogues se font à voix haute", autrement dit si deux modules communiquent, cette communication est établie explicitement dans la définition de l'un, de l'autre ou des deux modules (P4: interfaces explicites). Le dernier principe est le masquage d'information: toute information sur un module est privée (c'est à dire masquée des autres modules) sauf celles explicitement déclarées comme publiques: "on ne parle pas pour ne rien dire" (P5: masquage d'information).

De la même façon que pour les facteurs et les critères de qualité, on peut dresser le tableau de corrélation entre sous-critères et principes (figure 4).

---

[1] Des études en Psychologie ont montré que la mémoire à court terme de l'homme ne peut gérer que 5 à 9 informations élémentaires à la fois [Miller 63].

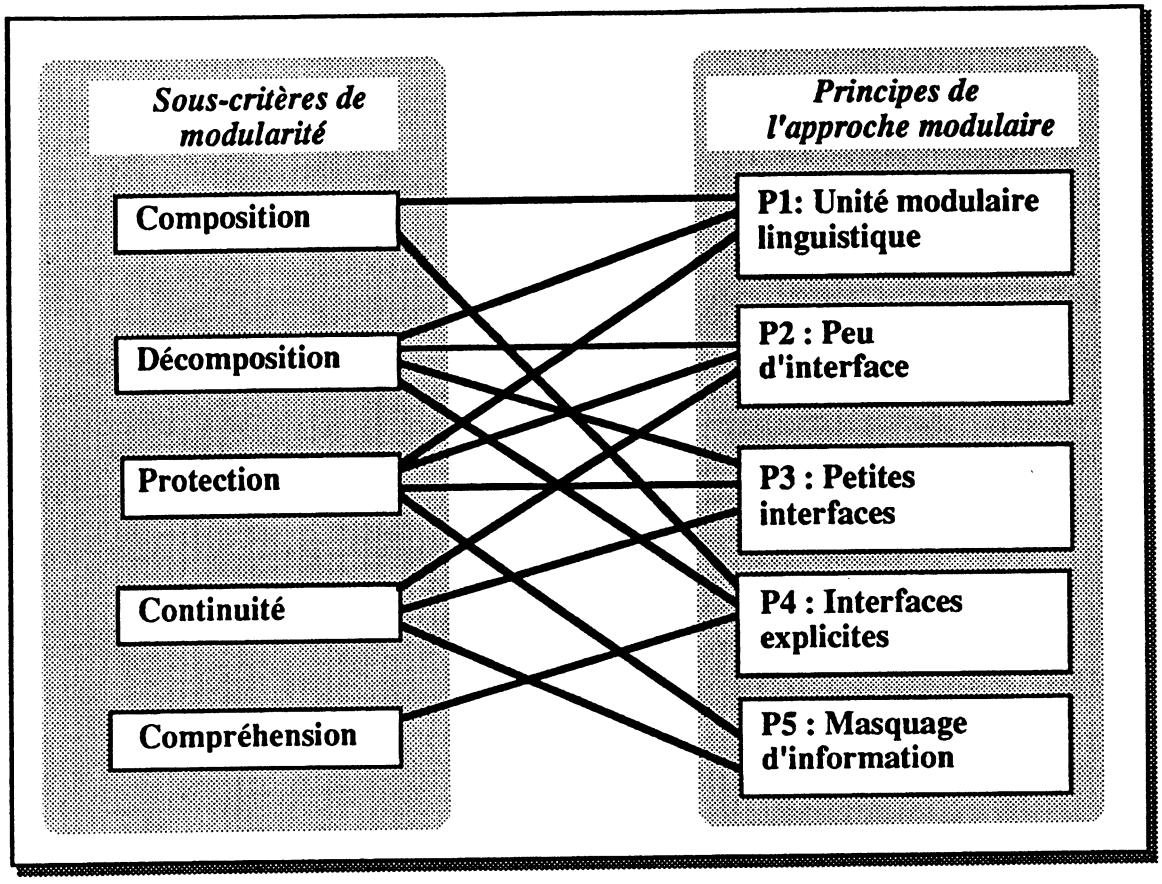


figure 4 : Corrélation entre sous-critères et principes d'une approche modulaire

1.6. La généralité

Une des différences les plus importantes entre logiciel et matériel est l'absence de réutilisation des développements et des programmes antérieurs. Il n'existe pas ou peu de possibilité de réutilisation, à part quelques bibliothèques très finalisées[1] permettant de concevoir de bas en haut sans toujours réinventer la roue. Les programmeurs se voient sans cesse contraints de réimplanter les mêmes algorithmes et structures de données à quelques variantes près.

Certains auteurs se sont penchés sur les causes de ce manque de réutilisation dans les développements de logiciels. Elles sont de deux sortes: techniques et non techniques.

• PROBLEMES NON TECHNIQUES:

Ils concernent le long terme et ne peuvent pas être résolus dans l'immédiat mais plutôt par une lente maturation des esprits sur ce sujet. Trois causes peuvent être dégagées [Meyer 88]:
- Une première cause est économique. La mise en place de catalogue de composants logiciels pose des problèmes économiques: Qui fournira ces catalogues ? à quels prix ? avec quelle garantie de qualité ?

1 et d'excellents ouvrages d'algorithmique! [Meyer 78], [Scholl 83], [Froidevaux 90]

- Les causes organisationnelles: comment mettre en place de tels catalogues ? Comment les diffuser, les faire connaître ? et surtout comment rendre ces catalogues faciles d'emploi et utiles ?
- L'aspect psychologique ou le "Not implemented here": c'est le syndrome bien connu de la peur de l'étranger, de la concurrence, ou tout simplement du refus de ce qu'on a pas inventé. Un programmeur vous dira souvent : "je peux le réécrire plus vite que je ne pourrai l'apprendre "[Cox 86]. Cette attitude n'est pas dénuée de bon sens étant donné que les méthodes classiques ne posent pas le problème de la réutilisation.

• **LES PROBLEMES TECHNIQUES:**

Les problèmes techniques soulevés par la réutilisabilité sont plus palpables et constituent un préalable à la résolution des problèmes non techniques. Partant du fait qu'on ne réutilise pratiquement jamais on peut se poser la question "pourquoi refait-on toujours la même chose?". Prenons l'exemple de la recherche dans une table, on s'aperçoit que la solution à ce problème classique et bien résolu est réimplantée régulièrement car il n'existe pas pour résoudre ce problème de composant logiciel suffisamment souple pour s'adapter à tous les cas de figures : table séquentielle ou non, types des éléments de la table, condition d'arrêt de la recherche, ...
C'est donc du critère de généralité qu'il s'agit. Les quatre principes suivant, énoncé par B. Meyer, garantissent la généralité d'un composant, donc son aptitude à être effectivement réutilisé:

• **Variante dans les types** :
Le composant réutilisable doit admettre des instances différentes de types. Ce principe introduit le concept de **généricité** (paramétrage du type). C'est l'exemple classique du module implantant la notion d'ensemble: il faut que le même module puisse être utilisé quel que soit le type des éléments manipulés.

• **Variante de structure de données et d'algorithme** :
On doit pouvoir fournir un composant adapté à différentes structures de données. Les mêmes noms d'opérations devront être applicables à ces différentes structures donneront accès à l'implantation voulue. Dans l'exemple de l'ensemble, cela signifie qu'une même fonction "ajouter_element" doit pouvoir avoir plusieurs interprétations possibles: ensemble implanté par des listes, ensemble implanté par des tableaux de booléens etc...

• **Indépendance de la représentation** :
L'utilisateur du composant ne doit pas être concerné par la représentation interne, aussi bien pendant la durée de vie du composant qu'au cours d'une même exécution. Lors de l'appel d'une fonction "appartient(x)" dans un code utilisateur, c'est le module utilisé qui détermine le code à appliquer en fonction du type d'ensemble concerné.

• **Création de sous-groupes** :
Des concepts peuvent être mis en commun dans des sous-groupes afin d'établir des abstractions intermédiaires. Cela permet d'affiner le composant et d'offrir à l'utilisateur des niveaux de richesse sémantique. L'**héritage** répond aussi à ce principe. On pourra par exemple différencier les ensembles stricts des multi-ensembles[1], les seconds possédant en plus la fonction:

nb-occurrences: Element X Multi-ensemble -> Entier

et vérifiant une axiomatisation différente [Froidevaux 90].

---

[1] ensembles avec répétitions.

Ces solutions techniques s'appuient sur des concepts proches de l'implantation et permettent de résoudre le problème de la réutilisation du code. On peut aussi poser le problème de la réutilisation des conception. En effet, une conception doit pouvoir mieux se plier à la réutilisation étant donné que son niveau d'expression est plus abstrait; on peut alors construire des modèles de conception suivant les domaines d'application. En fait, le problème majeur est celui de la consistance: la conception n'est pas réutilisée car elle ne correspond plus aux programmes, le cycle ayant été rompu pendant l'implantation ou la maintenance.
La figure 5 schématise le retour-arrière correspondant à la correction d'une erreur de conception détectée pendant la phase d'implantation.

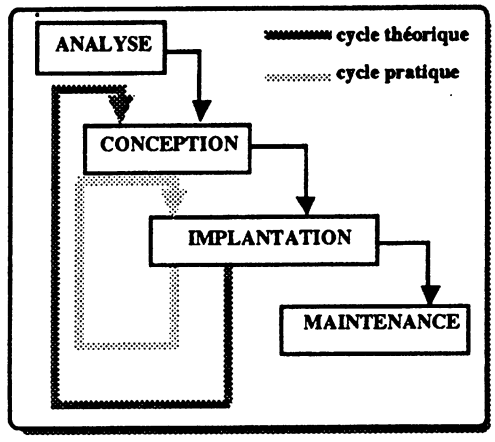


figure 5 : Erreur de conception
retour arrière théorique et pratique

Il faut donc maintenir la cohérence entre le code et la conception sous peine de réutiliser des choses incorrectes. Deux solutions peuvent être apportées : soit fournir des outils et environnements conservant -sur quels modèles?- la cohérence à tout moment, soit uniformiser le formalisme et les concepts entre les phases de conception et d'implantation. Dans ce cas, une erreur de conception est directement corrigée à l'intérieur d'un phase unique de conception/implantation (figure 6). Nous verrons que la méthode que nous proposons choisit la deuxième solution et que les concepts nouveaux qu'elle introduit s'inscrivent parfaitement dans le modèle objet des langages de programmation.

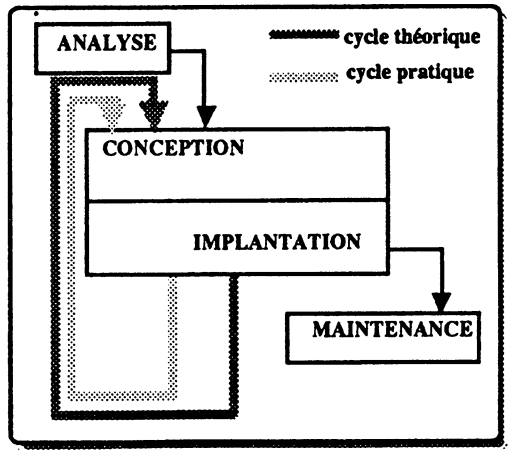


figure 6 : Erreur de conception
consistance conception <-> implantation

2. LA CONCEPTION

Dans cette partie, nous présentons les différentes tendances des méthodes de conception actuelles. L'objectif est de montrer les limites des méthodes classiques en terme de qualité des logiciels et d'introduire l'approche objet comme une réponse mieux adaptée.

Les approches classiques peuvent se classifier en trois catégories. On parlera de **conception fonctionnelle**, de **conception par les données** ou de **conception modulaire**[1] selon que l'approche privilégie les fonctions et opérations du système, les informations manipulées et les données ou bien que l'on recherche avant tout la décomposition en composants indépendants.

2.1. Conception fonctionnelle

Dans cette approche, le principe de décomposition suit le raffinement de la fonction abstraite du système. C'est la méthode qui a été introduite par Wirth [Wirth 71]. Le principe est de partir de la fonction principale du système et de la décomposer en sous-fonctions. On itère le processus sur les sous-fonctions pour obtenir finalement une structure dans laquelle les fonctions proches de la racine décrivent des abstractions procédurales de haut niveau et les fonctions de bas niveau sont les opérations primitives qui les implantent. Les données n'apparaissent qu'en tant qu'arguments de fonction et sont dispersées dans la structure.
Cette classe est bien adaptée aux problèmes séquentiels. Parmi les méthodes tirées de cette approche on peut citer SADT (utilisée comme méthode de conception) [IGL 82], [Ross 85], "Structured Design" [Stevens 74], [Myers 78], [Yourdon 79] et la programmation structurée en général [Dijkstra 76] .
Dans ce schéma, le composant logiciel est la fonction. La principale qualité recherchée et obtenue dans cette approche est l'exactitude: partant de la fonction globale du système, celle-ci est raffinée selon un processus strictement descendant en suivant les spécifications externes. Ceci conduit à un programme conçu pour assurer exclusivement les fonctionnalités demandées. Cette approche a apporté une plus grande rigueur dans le développement des programmes et a jeté les bases du génie logiciel.
Cependant, Elle nous semble insuffisante à plus d'un titre:

• **Evolution du logiciel** :
On sait que les demandes d'évolution d'un logiciel concernent presque exclusivement les fonctions et services qu'il rend. Aussi, une architecture basée sur les fonctions ne s'adaptera-t-elle que très difficilement à une évolution de ces fonctions. D'autre part, l'aspect fonctionnel d'un logiciel est souvent le moins important car il dépend beaucoup de l'interface: travaux "batch", interface type "session TTY", interface menu-déroulant non directive etc... Enfin, décomposer en fonctions c'est privilégier le contrôle et le séquencement des actions[2] qui sont sujets à évolution. En conséquence, cette approche pourra conduire à des architectures très différentes pour des problèmes de même nature au départ. On peut citer l'exemple d'un système d'exploitation qui dans le cas d'une interface "à la UNIX" possède la fonction abstraite principale suivante:

---

[1] On trouvera dans [Girod 87a] une étude sur des méthodes correspondant à ces différentes approches: SADT, MERISE, JACKSON, Structured Design, HIPO, MACH.

[2] Ces méthodes sont caractérisées par une approche "flot de données".

```
TOP_LEVEL
        faire
            lire une commande;
            exécuter une commande
        jusqu'à fin de session
```

alors qu'un système d'exploitation type MACINTOSH possède une fonction principale proche du modèle suivant:

```
TOP_LEVEL
    selon événement=
    click-souris :  selon localisation=
                    dans dossier: ouvrir-fenêtre-dossier
                    dans fichier: ouvrir-application
                    dans menu: ouvrir-menu;
    deplacement-souris: si boutton-presse
                        alors selectionner-zône
                        sinon deplacer-curseur
    etc...
```

Les fonctions abstraites de chaque programme sont très différentes alors que le problème est finalement le même : gérer des fichiers, des accès mémoire et disque ...

• **Dispersion des données:**
Les approches fonctionnelles ne considèrent le problème des structures de données que secondairement. Les données sont rattachées aux fonctions y compris les données centrales qui sont éclatées dans l'architecture fonctionnelle et difficiles à localiser.

• **Développement descendant ou "Top-Down":**
Ce type de développement en raffinements successifs s'il conduit à des logiciels fiables ne favorise pas du tout la réutilisation ni la fabrication de composants réutilisables (réutilisabilité). En effet, les fonctions sont conçues pour répondre spécifiquement aux besoins et ne sont pas suffisamment générales pour servir à d'autres développement.

• **Bilan en terme de qualité du logiciel:**
On peut dire que si cette approche offre la décomposition, la composition et la compréhension modulaires, elle n'assure pas la continuité modulaire. En effet le changement de l'interface utilisateur bouleverse l'architecture.
La généralité n'est pas non plus assurée car les fonctions sont trop spécifiques et les données dispersées. Enfin la flexibilité est absente: l'ajout de nouvelles fonctions demande une redécomposition générale.

2.2. Conception par les données

La conception par les données ou "structure de données" tend à prendre le contre-pied de l'approche précédente et privilégie une décomposition du logiciel basée sur les informations physiques manipulées par le système. Les unités de traitement sont ensuite définies et associées à ces données. Le programme final est vu comme un ensemble d'implantations de structures de données. Ces méthodes sont adaptées à tous les problèmes de type système d'information où le nombre de données est très important et les traitements relativement élémentaires.
Les méthodes WARNIER-ORR [Warnier 81], MERISE [Tardieu 82] et JACKSON [Jackson 83] sont des représentants de cette classe.

• **Bilan en terme de qualité du logiciel:**
Une telle approche améliore l'évolutivité, les données étant relativement stables dans un système. Par contre les fonctions deviennent secondaires, peu visibles et non structurées ce qui interdit de développer des logiciels à forte composante algorithmique et ne favorise pas la compréhension modulaire. En général cette approche sera réservée à de petits systèmes ne nécessitant pas de traitements complexes.
D'autre part, l'implantation physique des informations est directement accessible (pas d'abstraction de données). Cela rend plus difficiles la réutilisabilité (variante d'implantation, généralité) et à la continuité modulaire. L'adaptabilité se voit aussi fortement grèvée: 17,4% du coût de la maintenance vient du changement dans les formats de données. Par exemple, avec une telle approche, le simple passage de 6 à 8 du nombre de digits des numéros de téléphone peut avoir un impact considérable sur l'ensemble d'un système de gestion d'annuaire.

2.3. La conception modulaire: une méthodologie?

• **Qu'est-ce que l'approche modulaire?**
La conception modulaire consiste à décomposer le système en éléments relativement indépendants contenant à la fois des fonctions et/ou des structures de données [Parnas 72]. Le module satisfait au principe de masquage d'information en proposant une partie interface aux autres modules et en cachant son implantation. L'implantation (partie privée) est non accessible par les autres modules. L'interface peut contenir des fonctions, desactions ou des données, en import ou en export. Selon les méthodes, cette règle peut être plus stricte. Par exemple, dans les méthodes MACH [IGL 84] ou AMPHI [Girod 87b], les composants sont organisés en niveaux, ils exportent des actions vers le niveau supérieur et des données vers le niveau inférieur. La conception modulaire permet de créer des entités indépendantes ayant leur propre gestion et appréhendables de manière autonome.
Chaque module encapsule une entité de conception et implante une décision du concepteur. Cependant, on rappelle (principe 1 de la modularité) qu'une approche modulaire doit assurer un couplage minimum entre les modules et une cohésion maximale de chacun d'eux. C'est à la réponse apportée à la question *"que mettre dans un module?"* que l'on pourra juger une approche modulaire particulière.
On peut citer à ce sujet la comparaison de deux versions du "kwic problem[1]", l'une privilégiant une décomposition selon les traitements (module d'entrée, de sortie, de décalage circulaire et de tri), l'autre axée sur une décomposition en machines abstraites (titre, index, caractères lus, caractères édités) [IGL 84] La première décomposition conduit à un couplage sur les données désastreux: 5 structures de données, 1 à 3 accès par module, 1 à 4 accès par structure de données. La deuxième approche conduit à un couplage minimum: 4 structures de données encapsulées dans chacun des 4 modules, 1 accès par module, 1 accès par structure de données.

Dans l'approche modulaire, la complexité est réduite par niveau. Le programme devient facile à modifier ou corriger si les choix guidant la création d'un module sont judicieux.

En conclusion, l'approche modulaire offre un concept intéressant pour la structuration et la mise en place de l'architecture d'un logiciel, mais reste insuffisamment sémantique pour assurer des qualités telles que l'évolutivité ou la réutilisabilité. Il conviendra d'apprécier les qualités induites par une approche modulaire en fonctions des principes qu'elle propose (Cf §2.2.5.). C'est pour cette raison que nous pensons que l'approche modulaire ne constitue pas une méthodologie en tant que telle.

---

[1] KWIC (KeyWord In Context): problème de la production d'un index alphabétique de mot-clés pour une liste de titres d'ouvrages.

Des représentants de cette approche sont MACH [IGL 84] où le module est une machine abstraite, HIPO [Stay 76] dans laquelle le module réalise une fonction du système et les approches OOD ("Object-Oriented Design") dans lesquelles le module est l'objet [Booch 83] [Heitz 87] [Henry 89]. Nous aurons l'occasion de revenir sur ces méthodes dans le chaptre suivant.

2.4. Conception par objet

Nous introduisons la conception "par objet" ou "orienté-objet"[1] afin de montrer au moins intuitivement dans quelle philosophie elle se place par rapport aux trois approches précédentes. Cette approche sera longuement détaillée dans les chapitres 2, 3 et 4.
Dans une première définition, on peut dire que la conception par objet est une approche modulaire dans laquelle le critère de cohésion s'appuie sur le **modèle d'objet** ou **classe**. Un objet étant composé de données et d'opérations sur ces données, un modèle d'objet sera défini comme une implantation de type abstrait : spécification d'un type et réalisation du type. L'approche objet se situe à mi-chemin entre le concept modulaire et les méthodes de spécification. Du premier, elle reprend les principales qualités en les améliorant, des secondes, elle reprend le critère de cohésion (types abstraits).
Certaines méthodes considèrent l'approche objet comme l'adjonction d'une couche modulaire à l'approche structure de donnée; nous discuterons dans le chapitre 2 des problèmes de cette vue réductrice.

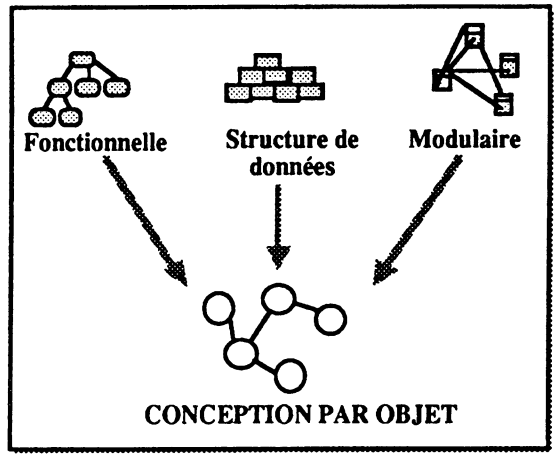


figure 7 : la conception par objet vue comme une évolution des méthodes classiques

D'autre part, l'approche objet peut être vue comme un compromis heureux entre les approches fonctionnelle et structures de données (figure 7). G. Booch déclare dans [Booch 86] que l'approche fonctionnelle reviendrait à ne parler qu'avec des verbes tandis que l'approche structure de données reviendrait à n'utiliser que les noms. L'approche objet réintroduit dans le langage informatique une notion somme toute naturelle : parler en mélangeant harmonieusement noms, verbes et adjectifs; ce qui permet au passage d'avoir le même discours à l'extérieur du système (utilisateur, analyste) qu'à l'intérieur (concepteur, programmeur). Cette propriété facilite grandement la lisibilité, l'appréhendabilité et donc la maintenance en général, ainsi

[1] Traduction franglaise de "object-oriented" à laquelle nous préférons le terme de "conception par objet".

que la consistance et la fiabilité. C'est aussi la raison du succès de cette approche qui a le mérite d'être naturelle.

Méthodologie	Avantages	Inconvénients	Qualités apportées	Qualités absentes
Fonctionnelle descendante	- niveaux d'abstraction	- données non structurées et dispersées - Top-down uniquement	- exactitude	- flexibilité - réutilisation - extensibilité
Structure de données	- stabilité des données	- traitements simples - petit système	- extensibilité	- fiabilité - adaptabilité
Modulaire	- masquage d'information - interface explicite - gros systèmes	- critère de modularité?	dépend de l'approche sous jacente	dépend de l'approche sous jacente
Par objet	- cohésion modulaire - bottom/up et top/down	- peu d'expérimentation	- réutilisabilité - extensibilité - adaptabilité - fiabilité	- ??

figure 8 : Tableau de synthèse sur les types de méthodologies

• Bilan en terme de qualité du logiciel

Le bilan que nous dressons ici est quelque peu prématuré puisque le reste de notre étude va justement porter sur la conception par objets. Nous allons revenir sur les apports de cette approche en terme de qualité sur les logiciels développés à la fin de ce chapitre. Cependant, à titre de comparaison avec les autres méthodologies, les qualités apportées par la conception par objet peuvent se résumer ainsi:

- meilleures réutilisation/réutilisabilité et évolutivité,
- apport d'un critère de cohésion modulaire très fort: un module représentant un objet de l'espace du problème possède une cohésion naturelle donc maximale,
- approche se prêtant aussi bien à une stratégie de conception descendante qu'ascendante.

3 . LE PARADIGME OBJET

Cette partie propose une introduction aux concepts du paradigme objet. En présentant le modèle objet par un de ses fondements théoriques (les types abstraits) et en définissant les concepts utilisés, nous poursuivons deux objectifs: aborder la programmation objet indépendamment d'un langage particulier, et mettre en place les définitions des termes et notions qui sont à la base de la méthode MECANO présentée dans la deuxième partie.

Nous présentons un rapide historique des langages à objets. Nous évoquons les langages les plus connus en les classifiant. Ceci nous permettra de placer notre étude dans le contexte des langages fortement et statiquement typés.

Un rappel sur la théorie des types abstraits considérés comme la base de l'approche objet est développé. Les types abstraits sont présentés comme un modèle idéal de spécification des classes.

Nous définissons ensuite les concepts principaux et la terminologie liés au modèle objet. Nous nous plaçons pour ce faire dans une approche résolument "typage statique", nous démarquant ainsi de l'école SMALLTALK à typage dynamique. Ceci nous permet de montrer le lien étroit entre le modèle objet et les types abstraits.

Enfin nous présentons un processus simple permettant de passer des types abstraits au modèle objet.

3.1. Historique et langages

3.1.1. Historique des langages à objets

L'étude suivante a été réalisée à partir de l'article de Saunders [Saunders 89] proposant un survol de 88 langages orienté-objet et de l'ouvrage de Masini "Les Langages à Objets" [Masini 89]. Le lecteur pourra se référer pour un langage précis à la bibliographie citée par Saunders ainsi qu'à l'article de Wolf comparant les langages C++ et FLAVORS [Wolf 89], l'ouvrage de Cox présentant OBJECTIVE-C, C++, ADA, et SMALLTALK [Cox 86], et celui de Meyer [Meyer 88] pour SIMULA, SMALLTALK, les extensions de C et celles de LISP.

Historiquement, l'approche objet est née avec SIMULA en 65. C'est le premier langage qui proposa les concepts de classes et d'objets pour structurer les programmes. Dans sa première version, SIMULA-1 était uniquement un langage de simulation [Dahl 66], la version suivante SIMULA-67 [Dahl 84] renommée SIMULA en 1986 [Kirkerud 89] est devenue un langage généraliste.

A partir de SIMULA, deux types d'approches vont être développées : l'école dite Scandinave développant des langages compilés avec un typage statique fort et un code procédural, et l'école SMALLTALK à typage dynamique, préférant des langages interprétés et plutôt fonctionnels (Cf figure 9).
Le cas des langages "basés objet" selon la terminologie de [Wegner 87] est un peu à part. Il s'agit de langages procéduraux comportant des constructeurs syntaxiques permettant de regrouper les éléments de programme dans des modules. Le plus abouti de ces langages est ADA.

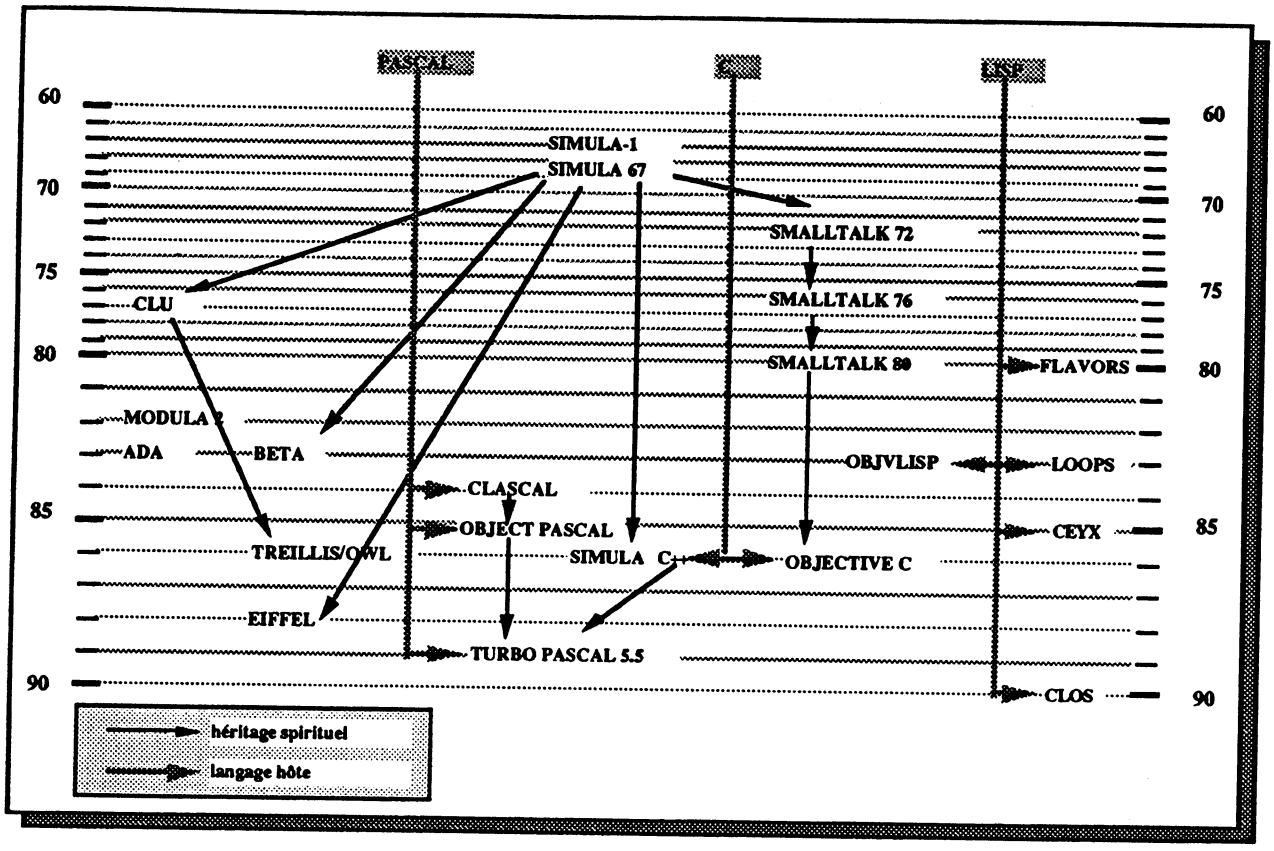


figure 9 : Langages à objets - chronologie et filiation

La figure 9 fait apparaître les langages et leur héritage spirituel (langages dont ils se sont inspirés). Les langages sont positionnés chronologiquement selon leur date d'apparition.
Quand il existe, le langage hôte est mentionné (C, PASCAL ou LISP). Le langage hôte est le langage considéré comme base pour implanter le modèle objet. Le terme "hybride" est parfois employé pour caractériser de tels langages offrant une extension objet à leur langage hôte. En effet, ces langages s'ils proposent les concepts objets, conservent le langage hôte pour la réalisation des méthodes. Généralement le paradigme objet cohabite avec le paradigme du langage hôte (procédural pour C et PASCAL, fonctionnel pour LISP)[1].
Dans la figure 10, les langages ont été regroupés par tendances: orienté-objet et basés objet, puis à l'intérieur des LOO: école Scandinave et école SMALLTALK.

3.1.2. Les langages basés objet

Ces langages se caractérisent par l'absence de la notion d'héritage (CLU) et pour certains (MODULA2, ADA) par l'absence de la notion de classe (modèle permettant la création d'instances) . MODULA2 propose le masquage d'information, CLU et ADA offrent la propriété d'abstraction de données [Stroustrup 87]. Ces langages ne sont généralement pas dédiés à une programmation par objets mais peuvent être candidats à l'implantation d'une conception orienté-objet (Cf Chapitre 2 et [Booch 86]). Si l'on se réfère à la classification de [Wegner 87], ces langages ne peuvent pas prétendre à l'appellation "Orienté-Objet".

---

[1] On notera que des extensions objets ont été réalisées pour la paradigme déclaratif : OBJLOG [Dugerdil 88] et SPOOL [Fukunaga 86].

ADA est un langage complet et complexe [ANSI 83]. Le composant modulaire est le "Package". Contrairement aux LOO dans lesquels une classe est à la fois un module et un type, le Package ADA n'est qu'un composant structurel du système. On peut transformer un ensemble de packages en un seul package de façon purement syntaxique. La création d'un package peut répondre à des objectifs très variés: du simple regroupement de constantes à l'implantation d'un type abstrait, en passant par la librairie de routines. En ce sens; on peut dire qu'ADA fournit les notions permettant d'adopter une approche modulaire. Le critère de cohésion est choisi par le concepteur, ADA n'en privilégiant aucun en particulier. Des constructeurs pour le parallélisme sont disponibles, par exemple, les mécanismes de synchronisation par rendez-vous. ADA permet la généricité et la surcharge, cependant l'héritage et la notion de classe/instance n'existent pas.
MODULA 2 est le successeur de Pascal et a été conçu par le même auteur [Wirth 82]. Le composant structurel est le Module. MODULA 2 est comparable à ADA sans la généricite et la surcharge.
CLU a été conçu à partir des travaux de B. Liskov sur les types abstraits [Liskov 74], [Liskov 77], [Liskov 81]. A part l'héritage, CLU implante tous les autres concepts objet. La classe s'appelle le "Cluster". Contrairement à ADA, CLU ne propose pas le parallélisme.

3.1.3. Les langages orienté-objet

Dans cette catégorie sont regroupés les langages qui possèdent les notions suivantes : classe, objet, envoi de message, liaison dynamique et héritage.

3.1.3.1. L'école scandinave

L'école scandinave ou européenne privilégie l'aspect statique en obligeant le programmeur à penser en terme de modèles (classes) qui donneront naissance à des instances à l'exécution. Cette approche se caractérise par un typage fort vérifié à la compilation et des mécanismes (polymorphisme, liaison dynamique, généricité) permettant d'apporter une certaine souplesse sans remettre en cause la rigueur du typage statique [Meyer 89a]. Les représentants sont SIMULA, C++ et EIFFEL. Ces langages sont dédiés à la réalisation de logiciels fiables, privilégiant la réutilisation et l'évolutivité.
Les programmes réalisés avec cette approche sont plus longs à mettre en oeuvre, conséquence de l'effort de formalisation et de spécification demandé. En revanche les programmes gagnent en fiabilité et maintenabilité. De plus cette approche s'adresse à tous les domaines d'application de l'industrie logicielle.
La classe est un composant déclaratif et statique. Elle donne naissance lors de l'exécution à des instances. Une instance possède un type: c'est la classe ayant créé l'instance.
Cette école regroupe des langages entièrement nouveaux (TREILLIS/OWL, EIFFEL, BETA) et des langages classiques auxquels on a adjoint une couche "objet" (OBJECT-PASCAL, C++).

Les "tout objet"

EIFFEL possède l'héritage multiple et la généricité contrainte. Il prend en compte la conception en introduisant des notions telles que les classes virtuelles, les méthodes retardées, les assertions sur les méthodes et l'invariant de classe vérifiables à l'exécution [Meyer 88]. Les méthodes peuvent être exportées, les attributs restent privés et ne peuvent être modifiés que via les méthodes.
TREILLIS/OWL est un langage proche d'EIFFEL, avec héritage multiple et code procéduralinspiré de SMALLTALK et de CLU [Schaffert 86]. L'héritage est comportemental (interface + implantation).

BETA est l'héritier direct de SIMULA, il intègre le parallélisme, les classes virtuelles (*pattern*) et l'héritage de méthode. L'édition de lien est complètement dynamique [Kristensen 87a, [Kristensen 87b].

Les "hybrides"

C++ est une extension de C. Un constructeur spécial permet de décrire des classes. Cependant la classe n'encapsule pas syntaxiquement ses composants. Les fonctions et attributs ne sont que nommés, leurs définitions interviennent en dehors de la classe. L'héritage simple est fourni et la prochaine version devrait offrir l'héritage multiple.
CLASCAL est une extension de PASCAL [Schmucker 86]. Il a été conçu pour offrir une couche d'accès à la toolkit de Lisa, le prédécesseur de Macintosh. Il a été remplacé par OBJECT-PASCAL [Tesler 85].
TURBO-PASCAL 5.5 est la version objet de TURBO-PASCAL [Borland 89]. Son approche est comparable à celle de C++ et s'inspire largement des travaux réalisés sur OBJECT-PASCAL. TURBO-PASCAL 5.5 privilégie l'approche structurelle[1] et offre classes et héritage simple dans un PASCAL modulaire.

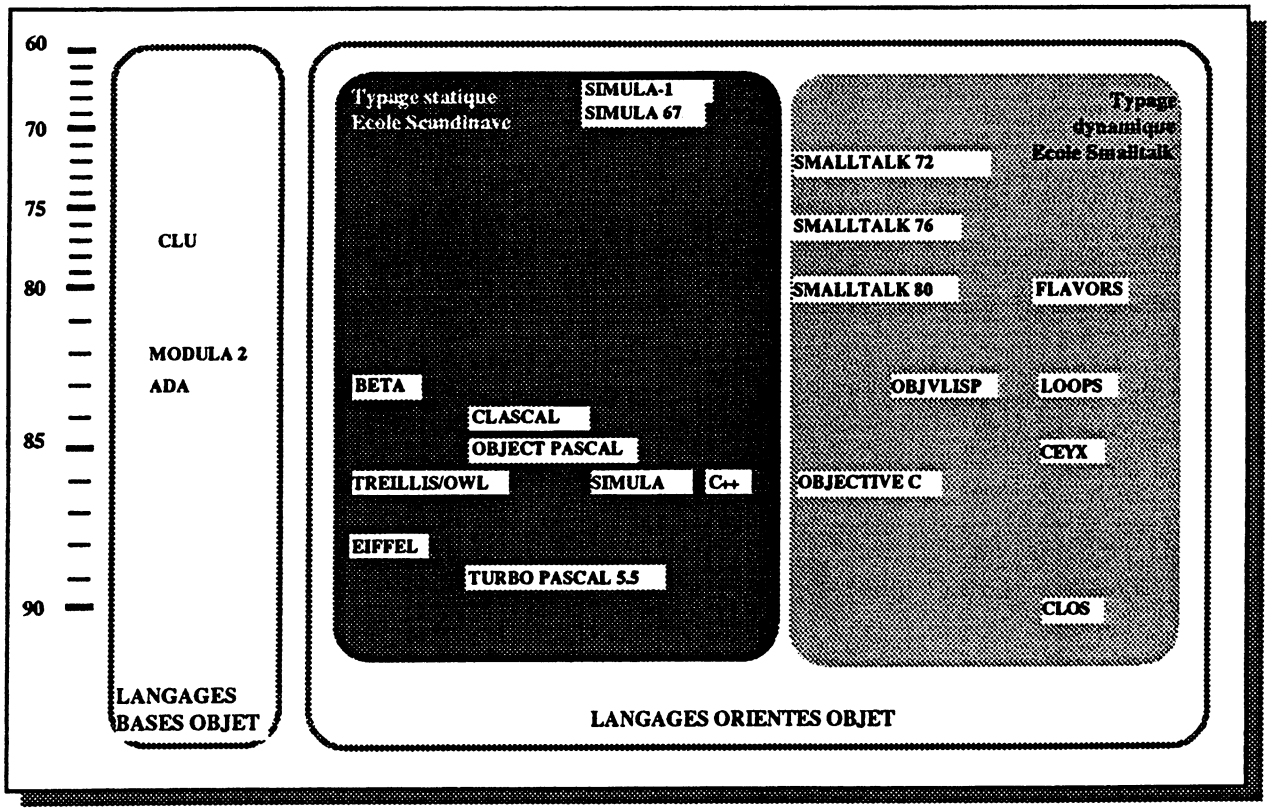


figure 10 : langages hôtes et typage

3.1.3.2. L'approche SMALLTALK

L'approche SMALLTALK préconise un typage dynamique et se caractérise par des langages interprétés, généralement dérivés de LISP et donc privilégiant le paradigme fonctionnel. Cette approche est traditionnellement orientée vers l'intelligence artificielle, le prototypage rapide, les interfaces graphiques, les environnements de

[1] "Des enregistrements qui héritent", tel est le crédo des concepteurs de Turbo-5.5.

programmation et tout développement de recherche en général. Le typage est faible ou n'existe pas et en conséquence aucune vérification statique ne peut être faite. Les langages sont généralement interprétés. L'édition de lien étant dynamique, une grande souplesse est apportée à la programmation permettant pour un appel procédural unique d'exécuter des codes différents selon la classe d'appartenance de l'objet destinataire. L'extrême souplesse de cette approche va de pair avec une mise au point rendue plus difficile.

SMALLTALK

SMALLTALK est le père historique de cette approche [Goldberg 83]. Les concepteurs de SMALLTALK ont été influencé par SIMULA mais ont cherché à pousser le paradigme objet au maximum.
La première version SMALLTALK-72 reprenait les concepts de classe et d'instance de SIMULA et a introduit l'envoi de message. La classe n'est pas encore un objet.
La version SMALLTALK-76 apporte la métaclasse *class* dont sont issues toutes les classes. *Class* est instance d'elle même pour éviter la régression à l'infini de l'instanciation. L'arbre d'héritage possède une racine prédéfinie: la classe *object* dont toutes les classes héritent.
En SMALLTALK-80 [Mevel 87] tout est objet. Chaque classe est elle-même un objet, instance de sa propre métaclasse. Les métaclasses sont toutes instances d'une classe unique appelée *Metaclass*.
En SMALLTALK, une classe est dynamique et a une existence pendant l'exécution. Toute classe peut être créée en envoyant le message *new* à sa métaclasse. Le seul mécanisme de contrôle est l'envoi de message.

Les "LISPIENS"

Les concepts orienté-objet s'implantent bien en LISP grâce aux caractéristiques que ce langage possède: ramasse-miettes, représentation uniforme des programmes et des données, langage interprété et non typé. On distingue deux approches: les langages utilisant l'interprète et l'environnement LISP pour réaliser un nouveau langage (LOOPS), et les extensions aux différents dialectes de LISP (CLOS, FLAVORS, CEYX).
LOOPS est bâti sur l'interprète InterLISP-D [Bobrow 83]. Il permet plusieurs styles de programmation: programmation objet, programmation logique et programmation dirigée par les accès (data oriented programming) dans laquelle des appels fonctionnels peuvent être déclenchés sur des événements, comme par exemple l'accès ou la modification d'une donnée[1]. Il offre l'héritage multiple, la composition d'objet, des outils de gestion de base de connaissances. Les conflits d'héritage sont résolus en utilisant une liste de priorité. LOOPS n'est disponible que sur les machines XEROX.
CEYX est un sur ensemble de LE-LISP [Hullot 85]. Les objets sont typés et les types sont organisés en une hiérarchie de types/sous-types. L'héritage est linéaire.
FLAVORS puis NEW-FLAVORS a été conçu pour rédiger le système d'exploitation des machines Symbolics [Moon 86]. Une classe s'appelle un *flavor*. L'héritage est multiple avec résolution de conflit par liste de priorité établie selon la position de la classe parente dans la déclaration d'héritage.
OBJVLISP est une généralisation du modèle de SMALLTALK 76. Classe, métaclasse et objet sont unifiés et tout objet est instance d'une classe [Cointe 87]. OBJVLISP n'est cependant pas un langage mais un modèle permettant d'expérimenter les modèles objets. CLOS et SMALLTALK ont fait l'objet d'une étude en OBJVLISP.
Le langage CLOS pour Common LISP Object System constituera la norme objet de COMMON-LISP. Il est en cours de spécification [Keene 89].

---

[1] La programmation par *trigger* -litteralement *déclencheur* ou *gachette*- se développe surtout dans les bases de données.

OBJECTIVE-C est un préprocesseur C offrant une véritable couche objet sur C et calqué sur SMALLTALK [Cox 86]. C'est le seul langage hybride de l'école SMALLTALK basé sur un langage procédural. En OBJECTIVE-C, la classe est une notion déclarative statique mais elle possède un équivalent dynamique appelé *Factory*, véritable machine à créer des instances. Une "Factory" est créée à la compilation de la classe correspondante. Le typage est complètement dynamique. Toute entité est de type prédéfini *ID*. L'héritage est linéaire, les routines sont exportées et les attributs sont privés comme en SMALLTALK. La librairie a été reprise de SMALLTALK. Un outil a même été développé pour traduire un programme SMALLTALK en OBJECTIVE-C. En conclusion, OBJECTIVE-C est un langage plus efficace mais moins homogène que SMALLTALK car la couche C reste accessible.

3.1.3.3. Comparaison des deux écoles

Au delà du typage statique et dynamique, ces approches diffèrent par le paradigme de programmation qu'elles proposent.
L'approche dynamique exploite complètement le paradigme objet et propose une nouvelle manière d'aborder la résolution de problèmes algorithmiques. On peut en effet classifier les langages de programmation selon le paradigme qu'ils proposent : procédural, fonctionnel, déclaratif ou orienté-objet. Dans le premier, on range des langages comme PASCAL, FORTRAN ou ADA qui privilégie l'aspect traitement et considère un programme comme une succession d'états. Le second s'appuie sur le mécanisme de l'évaluation fonctionnelle où tout programme est une fonction qui rend un résultat. LISP en est le représentant. Dans l'approche déclarative, on s'attache à décrire les relations mises en cause pour résoudre le problème, le système se chargeant de donner les solutions correspondantes. Il s'agit par exemple de la programmation logique de PROLOG ou des langages de construction de systèmes experts comme OPS-5.
Dans le paradigme orienté-objet de SMALLTALK, on considère que tout est objet et le seul mécanisme de contrôle est l'envoi de message entre objets. Les entiers sont eux-mêmes des objets et une opération telle que 3+2 sera traduite par un envoi d'un message dont le sélecteur est "+", le destinataire l'objet "3" et le paramètre l'objet "2".

Dans l'approche statique du modèle objet, on considère la décomposition d'un système en objets (plus généralement en classes) comme primordiale car elle permet de mettre en place l'architecture du programme. On s'attache particulièrement aux propriétés de spécification, d'abstraction de données, de masquage d'informations et d'autonomie des objets. Les méthodes (algorithmes) des classes, après spécification, seront implantées selon un paradigme classique généralement procédural. Ceci explique en partie le succès des langages hybrides comme C++ ou TURBO-PASCAL 5.5 qui greffent, avec plus ou moins de réussite, le modèle objet sur les constructeurs habituels du langage. Cependant, l'approche d'EIFFEL qui propose un langage totalement nouveau pour implanter le modèle objet, apporte une cohérence des concepts et de la notation allant dans le sens de l'élaboration de programmes de qualité. Le langage est de fait plus homogène et oblige le programmeur à utiliser le paradigme objet, ce qui n'est pas vrai de C++ qui reste utilisable comme une version statiquement typée de C.

Notre contribution se situe dans l'approche typage statique. Il est bien évident que, en considérant l'aspect conception au sein du développement par objet, nous sommes plus naturellement associés à cette approche permettant de décrire **à priori** les modèles (classes) intervenant dans le système. Pour la définition des notions de la programmation objet qui va suivre, nous considérons généralement l'approche du typage statique. Ceci permettra de privilégier certaines notions clés de cette école (généricité, polymorphisme, masquage d'informations, assertions) sans s'égarer dans les discussions subtiles concernant les classes, métaclasses et autres méta-métaclasses de l'école SMALLTALK.

3.2. Les Types Abstraits de données: modèle sous-jacent

Les types abstraits de données constituent une théorie en informatique au même titre que l'étude de la complexité ou des langages [Liskov 74], [Guttag 77]. Les types abstraits permettent de décrire les données d'un point de vue externe -synthétique-. On parle aussi d'abstraction de types par opposition aux constructeurs de types. La figure 11 montre les différentes approches des types selon les langages. Contrairement aux types concrets qui décrivent les structures de données par leur implantation en machine comme dans PASCAL, le type abstrait s'attache à spécifier le comportement d'une donnée en définissant des ensembles mathématiques, des opérations et une axiomatisation. Une grande activité de recherche est déployée sur ce thème afin de mettre en place des langages de spécification algébrique et des systèmes de programmation automatique. Dans l'état actuel des travaux, ces systèmes adaptés pour des cas précis, ne peuvent pas subvenir à la majeure partie des besoins des développements logiciels. Aussi, l'approche objet en offrant un intermédiaire satisfaisant entre, d'un côté, l'idéal de tout programmeur -spécifier sans programmer- et la réalité brutale -écrire des lignes de codes et vérifier qu'elles répondent aux spécifications-, permet d'aménager une transition en douceur et sans doute durable vers un âge d'or informatique encore lointain. En effet, la philosophie de langages comme EIFFEL peut se résumer ainsi :

> *"Ecrivez les spécifications de vos objets (types abstrait); implantez ensuite ces spécifications à l'intérieur des objets; le système essayera de garantir le respect des spécifications"*

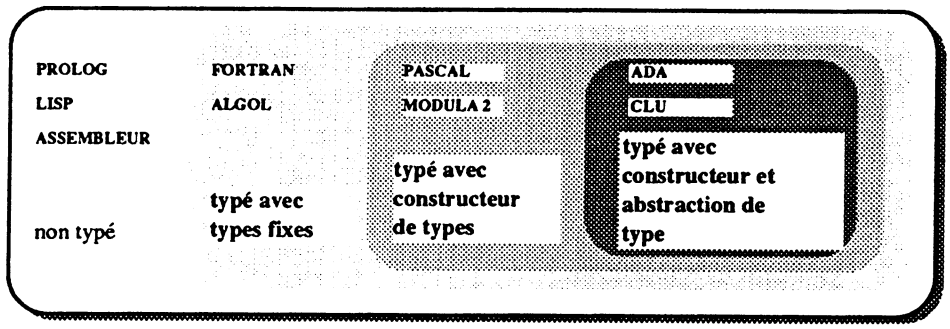


figure 11 : typage dans les langages

Nous rappelons dans la suite des éléments du modèle des types abstraits. Ceci nous servira pour présenter la transformation type abstrait -> objet de la section §3.4. Le lecteur familiarisé avec ce modèle pourra passer directement au §3.3.

3.2.1. Spécification d'un type abstrait de donnée

Le type abstrait de donnée se distingue du type concret par le point de vue qu'il offre d'une donnée ou d'une structure de données. Le type concret considère la donnée sous l'angle de l'implantation (représentation de donnée), par exemple :

```
type PILE1 =
      tableau + indice
      {l'indice indique à tout moment l'élément au sommet de pile}
type PILE2 =
      pointeur sur la structure (élément, pointeur_suivant)
      {le sommet est le premier élément de la liste}
```

Le type abstrait se place du coté de l'utilisateur du type. Pour décrire un type abstrait, on utilise une spécification algébrique qui décrit le nom du domaine de valeur appelé "sorte" et les opérateurs agissant sur le type.
Exemple:

```
type PILE3
opérateurs
      empiler:  (pile,naturel)  -> pile
      pile_vide:                -> pile
      sommet : pile             -> naturel
      etc...
```

En plus, la spécification définit la sémantique des opérateurs au moyen d'axiomes.

```
axiomes
      (1) dépiler(empiler(p,x)) == p
      (2) sommet(empiler(p,x)) == x
```

Une spécification de type abstrait est donc constituée:
- d'un **nom**
- des types **importés**
- de **sortes**
- d'un ensemble d'**opérateurs**
- d'un ensemble de **pré-conditions**
- d'un ensemble d'**axiomes**

Les types importés correspondent à des spécifications extérieures ajoutées à la spécification courante. Les sortes sont les domaines de valeurs. En général, il n'y a qu'une sorte de même nom que le type abstrait en cours de définition, mais la théorie des types abstraits autorise une spécification avec plusieurs sortes. Les opérateurs sont des fonctions dont les paramètres et les résultats prennent leurs valeurs dans des sortes. Les opérateurs se classifient en trois genres:

- **constructeur** ou générateur:
  C: s -> st
  où **st** est la sorte du type en cours de définition
  le constructeur rend (construit) une valeur du type à partir d'une valeur de sorte **s**
  s peut être absent (constructeur constante)

- **accès** ou observateur:
  A: st -> s
  où **st** est la sorte du type en cours de définition
  l'accès rend une valeur de sorte **s**

- **transformateur**:
  T: st -> st

où **st** est la sorte du type en cours de définition
transforme une valeur du type en une autre valeur du type

Remarque:
Cette classification est étendue au fonctions n-aires en considérant la présence ou non de **st** dans le n-uplet des sortes d'entrée.
Certains opérateurs sont **partiels** i.e. il existe des éléments du type pour lesquels ils ne sont pas définis. Pour ces opérateurs, une **pré-condition** définissant logiquement le sous-ensemble de départ est établie.
La définition du nom, des sortes et des opérateurs s'appelle la **signature** du type abstrait. La signature défini l'aspect **syntaxique** du type.
Les axiomes permettent de spécifier les opérateurs, ils doivent être vérifiés à tout moment par tous les éléments du type. Les axiomes définissent la **sémantique** du type (le comportement). Par exemple, l'axiomatisation permettra de définir des politiques d'insertions et d'accès différentes (LIFO ou FIFO) pour les types PILE et FILE dont les signatures sont pourtant identiques[1].

Nous donnons à titre d'exemple, le type abstrait décrivant les Naturels.

```
type  Naturel
sorte
     naturel
importe
     booléen
opérateur
     constructeur
          0:   -> naturel
          1:   -> naturel
     accès
          pair: naturel -> booléen
     transformateur
          +:  naturel x naturel     ->    naturel
axiome
     (1)  pair(0) == vrai
     (2)  pair(1) == faux
     (3)  pair(x + y) == pair(x) et pair(y) ou non pair(x) et non pair(y)
     (4)  0+x ==  x
     (5)  x+0 ==  x                {0 élément neutre de l'addition}
     (6)  (x + y) == (y + x)       {commutativité}
     (7)  (x + y) + z == x + (y + z)   {associativité}
fin  type  Naturel
```

3.2.2. Paramétrisation de type

On peut, pour des raisons de généralisation, **paramétrer** un type abstrait. Dans la spécification du type, on emploie alors un **paramètre formel** qui représente un type (sorte) inconnu. Ce type sera instancié lors de l'utilisation du type paramétré pour définir des opérateurs.
Par exemple, on peut spécifier une pile d'éléments de type quelconque:

```
type  Pile(T)
sorte
     pile(T)
```

[1] au nom des opérateurs près, mais les noms ne constituent pas une information formelle.

```
opérateur
    constructeur
        pilevide:                    -> pile(T)
    accès
        estvide:        pile(T)      -> booléen
        sommet:         pile(T)      ->T
    transformateur
        empiler:        pile(T) x T -> pile(T)
        dépiler:        pile(T)      -> pile(T)
pré-condition
    (1) precond(dépiler(p))      =    non  estvide(p)
    (2) precond(sommet(p))       =    non  estvide(p)
axiome
    (1) estvide(pilevide)            == vrai
    (2) estvide(empiler(p,n))        == faux
    (3) dépiler(empiler(p,n))        =p
    (4) sommet(empiler(p,n))         ==n
fin  type  Pile(T)
```

En instanciant le paramètre formel générique T avec un type effectif Te (par exemple Naturel), on obtient un nouveau type abstrait permettant de manipuler une pile d'élément de type Te (Naturel).

3.2.3. Les propriétés

La généricité, telle qu'elle a été définie plus haut, convient la plupart du temps pour construire des structures de données contenant des éléments dont le type est paramétré. Dans ce cas, les éléments du type formel ne sont utilisés que comme paramètres ou résultats des opérateurs du type. Il arrive cependant que la spécification de type ait besoin de fonctions plus spécifiques sur l'élément de type générique [Jacquet 78].
Par exemple, si l'on veut définir le type vecteur avec l'opération d'addition de deux vecteurs:

```
type  Vecteur(T)
opérateur
    accès
    valeur:   entier x vecteur -> T
    valeur(i,v): rend le ième élément du vecteur v
    transformateur
    +: vecteur(T) x vecteur(T) -> vecteur(T)
    effectue la somme de deux vecteurs
    ...
axiome
    (1) valeur(i,v1 + v2) == valeur(i,v1) + valeur(i,v2)
    ...
fin  type  Vecteur(T)
```

L'opérateur + est un opérateur sur le type T car *valeur(i,v)* rend un élément de type T. Dans ce cas, le type formel ne peut pas être remplacé par n'importe quel type. Le type effectif devra impérativement fournir un opérateur d'addition. Pour caractériser l'ensemble des types pouvant remplacer le paramètre formel on définit une **propriété.** La propriété est nommée et spécifie un ensemble d'opérateurs avec une axiomatisation adéquate sur un type formel.
On dira qu'un type générique **exige** une propriété sur un type formel.

Un type **possède** la propriéte P si sa définition contient les opérateurs de P avec l'axiomatisation associée.
Un type peut être substitué à un type formel s'il possède au moins la propriété exigée par le type formel[1].
Dans l'exemple précédent, la propriété exigée sur T est *monoide(T)* définie par:

```
propriété  monoide(T)
     opérateur
          + : T x T -> T;
          0 : -> T;
     axiomes
          (1) 0 + x == x
          (2) x + 0 == x
          (3) (x + y) + z == x + (y + z)
fin  propriété  monoide
```

le type vecteur se réécrit maintenant:

```
type  Vecteur(T)
exige  monoide(T)
opérateur
     accès
     valeur:   entier x vecteur -> T
     valeur(i,v): rend le ième élément du vecteur v
     transformateur
     +: vecteur(T) x vecteur(T) -> vecteur(T)
     effectue la somme de deux vecteurs
     ...
axiome
     (1) valeur(i,v1 + v2) == valeur(i,v1) + valeur(i,v2) ...
fin  type  Vecteur(T)
```

Le type abstrait **Naturel** possède la propriété **monoide** (axiomes (4),(5) et (7)) et peut par conséquent être candidat comme type effectif de vecteur(T). Par contre, le type Pile(T) ne peut pas être le type d'un élément de vecteur puisque l'addition n'est pas définie sur les PILES.
Les fonctions des propriétés peuvent être réparties en deux catégories: les fonctions **fondamentales** et les fonctions **dérivées**. Les fonctions fondamentales sont celles qui devront être définies obligatoirement dans le type. Les fonctions dérivées sont celles qui peuvent être définies (axiomatisées) dans la propriété.

```
Propriété ORDRE(T)
     importe BOOLEEN
     opérateur   fondamental
          <=: T x T -> booléen
     opérateur   dérive
          =: T x T -> booléen
          <>: T x T -> booléen
          <=: T x T -> booléen
          >: T x T -> booléen
```

[1] On ne confondra pas type abstrait générique et propriété. Le premier définit un ensemble d'objets caractérisés par les fonctions qui les manipulent (généricité structurale) alors que la propriété définit un ensemble de fonctions caractérisant une classe de types (généricité incrémentale) [Jacquet 78].

```
            >=: T x T -> booléen

axiome
            (1) x <= x == vrai                                réflexivité
            (2) x <= y et y <= z et non x <= z == faux         transitivité
            (3) x = y == x <= y et y <= x                     définition de =
            (4) x <> y == non x<=y ou non y<= x               définition de <>
            (5) x > y == non x <= y                           définition de >
            (6) x >= y == non x <= y ou (x <= y et y <= x)    définition de >=
fin propriété ORDRE(T)[1]
```

Un type possède une propriété s'il définit une axiomatisation des opérateurs fondamentaux.

3.3. Le modèle objet

Si l'on considère la définition du Larousse[2], on est surpris de voir à quel point le concept d'Objet s'applique au terme informatique choisi pour définir les "composants" des programmes orienté-objet:

"Ce qui se présente à la vue..."
On caractérise avant tout un objet par son aspect externe, par la vue qu'on en a en tant qu'observateur. Ceci rejoint bien le soucis de spécification et d'abstraction de donnée présent dans une approche basée sur la théorie des types abstraits.

"Chose matérielle façonnée en vue d'un usage précis... "
Une "*chose*": c'est l'aspect "donnée" de l'objet, "*matérielle*": l'objet sous-entend une implantation physique, "*façonnée en vue d'un usage précis*": l'objet offre un service et possède un comportement (aspect fonctionnel).

"Monde extérieur, connu immédiatement par nos sens..."
Les objets se composent et s'assemblent pour former un monde; leur compréhension est totale et immédiate.
Dans la suite, nous introduisons les notions de base et les notions optionnelles du modèle objet. Ces notions seront présentées plus en détail au Chapitre 3, lors de la définition des concepts présents dans la méthode MECANO.

3.3.1. Les notions de base

Les notions de base regroupent les concepts présents dans tous les langages ou modèles dits "orienté-objet" .

3.3.1.1. La classe, l'objet

Une **classe** est la description d'une famille d'**instances** ou **objets** ayant le même comportement et la même structure.
- Les objets ont le même **comportement**: la classe définit un type abstrait.
- Les objets ont la même **structure**: la classe définit une représentation (type concret) du type abstrait.

---

[1] Cette propriété pourra par exemple être exigée par le type abstrait ARBRE_BINAIRE_ORDONNE(T).
[2] Voir première page du chapitre.

Le comportement d'un objet est défini par un ensemble de méthodes (procédures ou fonctions). La structure est composée d'attributs (les données). Tous les objets possèdent le même comportement mais se différencient par leur **état**. L'état d'un objet est défini par les valeurs associées aux différents attributs.
Les attributs et les méthodes d'une classe sont regroupés sous le terme générique de **caractéristiques**.

```
CLASSE =
        ENSEMBLE DE DONNEES  (composante statique)
+       ENSEMBLE DE METHODES (composante dynamique)

CLASSE = ATTRIBUTS + METHODES

OBJET = ETAT + COMPORTEMENT
```

Exemple :

```
Classe ARTICLE
    attribut
        référence, désignation, prixHT, quantité
    méthode
        prixTTC        = 1.186 * prixHT
        prixTransport  = 0.05 * prixHT
        retirer(q) :   quantité <- quantité - q
        ajouter(q) :   quantité <- quantité + q
```

Instanciation
La classe constitue un moule à partir duquel les instances sont produites. Dans les langages de l'école SMALLTALK, la création d'instance est une opération du niveau classe. On envoie un message de création (new) à la classe considérée comme instance d'une métaclasse. Ceci permet d'uniformiser le modèle. Ce n'est pas le cas d'EIFFEL par exemple, où la méthode *Create* fait partie de la définition de classe et est envoyée à une référence *ref* de type "*void*" afin de créer un nouvel objet référencé par *ref*.
Exemple d'instanciation:

INSTANCE <------	CLASSE	-----> INSTANCE
12003	référence	16867
lave-linge	désignation	camescope
2320	prixHT	9450
17	quantité	25

3.3.1.2. Envoi de message

Les objets communiquent en s'envoyant des messages. La terminologie objet a choisi le terme de message pour remplacer le classique appel de procédure. Ce choix est fondé sur les raisons suivantes.
Dans un envoi de message on identifie toujours un paramètre particulier comme étant le destinataire du message. C'est au destinataire qu'incombe la responsabilité de traiter le message de la manière qu'il lui convient. De ce fait, l'appel (nom + paramètre) est dissocié du code qui doit être exécuté. Par la liaison dynamique, on peut même retarder jusqu'à l'exécution le moment de cette association. La richesse des objets vient justement du fait que différents codes peuvent être exécutés pour le même nom de procédure selon le type du destinataire du message (surcharge et polymorphisme).

Un message est composé d'un objet destinataire, d'un sélecteur (nom de la méthode) et d'une liste de paramètres effectifs.
Exemple:

```
Classe TRUCMUCHE
    ...
    attribut camescope-VHS : ARTICLE
    méthode gerer_stock:
        ...; message(camescope-VHS, ajouter, 15); ...
```

L'objet référencé par camescope-VHS va modifier son état (quantité <- quantité + 15)

3.3.1.3. Masquage d'information et interface

Pour l'instant, les méthodes et attributs d'une classe ne sont pas masqués et sont par conséquent utilisables par n'importe quelle autre classe. En considérant une classe comme une implantation de type abstrait, on peut classifier les méthodes et attributs qu'elle possède en deux ensembles:

- les méthodes et attributs correspondant à la clause *opérateur* du type abstrait,
- les méthodes et attributs utilisés pour implanter le type abstrait.

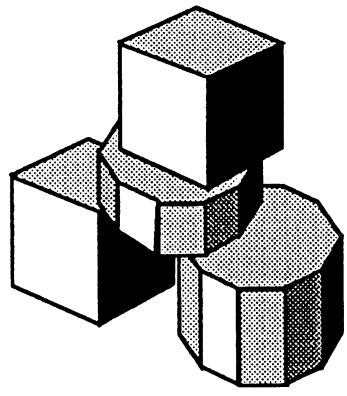
Pour des raisons évidentes de protection, il convient de masquer les éléments du deuxième ensemble. C'est le rôle de l'interface (ou export) qui définit les méthodes visibles de l'extérieur donc utilisables pour des envois de message. En ce qui concerne les attributs, certains langages les considèrent comme automatiquement masqués (SMALLTALK). D'autres permettent l'export d'attributs (EIFFEL), dans ce cas, l'attribut se comporte comme une fonction 0-aire et ne peut pas être modifié directement par une classe extérieure.
L'interface permet de réaliser l'abstraction de donnée en empêchant l'accès direct aux structures. L'implantation peut alors être modifiée sous réserve qu'elle respecte les spécifications, sans incidence sur les clients actuels de la classe.

3.3.1.4. Héritage et polymorphisme

L'héritage est un moyen incrémental de définition de classe. C'est une relation inter-classe qui permet de définir une classe (classe fille) par rapport à une autre classe (classe mère). La classe héritière dispose du "patrimoine génétique" de la classe parente. Ce patrimoine est constitué des méthodes et des attributs qui sont considérés comme propres à la classe fille.
En général, les mécanismes d'héritage autorisent une certaine adaptation des méthodes et attributs hérités dans la classe fille, comme la redéfinition, le renommage et le blocage.
La **redéfinition** permet d'hériter du nom et parfois de la spécification (paramètres et axiomatisation) d'une méthode sans hériter de la réalisation. Cela permet de proposer une nouvelle réalisation plus adaptée au contexte de la classe fille.
Le **renommage** est une facilité syntaxique pour changer le nom d'une caractéristique héritée. Il permet d'adapter un nom à un contexte et de conserver l'accès à une méthode redéfinie. On notera que certains langages proposent la spécialisation de méthodes (BETA, CLOS) qui permet de composer directement une méthode à partir de la méthode héritée (mécanismes "before / after").
Le **blocage** permet de masquer certaines caractéristiques dans une sous-classe. Cette propriété est surtout utilisée dans les langages de représentation de connaissances pour exprimer des cas particuliers, comme par exemple le fait qu'une autruche est un oiseau

PARTIE 1:

L'APPROCHE OBJET POUR LA CONSTRUCTION DES LOGICIELS



qui ne vole pas. Cependant si l'on dispose d'un modèle avec masquage d'information, on peut très bien gérer ce cas en n'exportant pas les méthodes correspondantes.

Le **polymorphisme** est la possibilité pour une entité d'avoir plusieurs formes ou plus précisément d'être vue sous différentes formes. Concrètement, cela signifie qu'un envoi de message:

message(destinataire, sélecteur, paramètres)

peut déclencher l'exécution de méthodes de même nom (sélecteur) mais d'implantations différentes selon le type (la forme) de l'objet référencé par le destinataire. Cette propriété permet d'écrire des programmes généraux s'appliquant à des objets différents. Cela conduit à éviter les interminables instructions de choix telles que:

```
selon type de E faire
     type = t1 : instruction1
     type = t2 : instruction2
     etc ...
```

De plus le programme général devient évolutif: il est possible de rajouter de nouveaux types dans le système sans modifier le programme. Le code écrit reste valide pour ce nouveau type, ce qui n'est pas vrai avec la structure de choix précédente.
Prenons par exemple le cas d'une procèdure *VOIR* chargée d'afficher une liste d'objets à l'écran:

```
VOIR(liste_aff : LISTE(AFFICHABLE)) :
    Répéter
        liste_aff.premier.afficher
        liste_aff.reste
    jusqu'à liste_aff.est_vide
```

La méthode *afficher* appartient à la classe AFFICHABLE; tout objet désirant avoir la propriété d'être affichable doit hériter de cette classe. Chaque objet peut redéfinir la méthode *afficher* pour l'adapter à son propre mode d'affichage. Selon l'objet, la méthode *afficher* utilisée sera celle liée au type de l'objets (icône, fenêtre, menu, figure géométrique etc..). La souplesse est donc apportée à deux niveaux:

- un seul appel pour des codes exécutés différents,
- possibilité d'extension à d'autres types d'objets sans changer l'appel.

Il faut cependant bien distinguer le polymorphisme non contrôlé des langages comme SMALLTALK, du polymorphisme réglementé des langages à typage statique. Dans le premier cas, aucun contrôle de type n'étant fait statiquement, il est possible d'envoyer à un objet un message de sélecteur quelconque. C'est à l'exécution que l'objet détermine grâce à son dictionnaire de méthodes s'il peut ou non répondre au message[1].
Dans la deuxième approche, on ne peut envoyer un message à un objet ne possédant pas ou n'héritant pas d'une méthode correspondant au message (sélecteur et paramètres). Le polymorphisme est ainsi contrôlé statiquement. Bien que ne connaissant pas le type réel de l'objet destinataire du message, on s'assure qu'il sera un descendant du type destinataire. De plus, si le langage propose des mécanismes d'assertions, une méthode redéfinie doit assurer les assertions de la méthode héritée. Cette technique permet de garantir l'usage du polymorphisme à des situations où l'objet est réellement polymorphe et évite qu'une méthode soit détournée de sa spécification initiale.

---

[1] En SMALLTALK, un message incompris déclenche l'envoi d'un message *doesNotUnderstand* permettant à l'emetteur de récupérer l'erreur.

Nous verrons au chapitre 3 que le polymorphisme est un concept privilégié de la méthode MECANO.

Les différents points de vue de l'héritage

L'héritage est vu de différentes façons selon le langage, le domaine d'application ou la "philosophie objet" adoptés.
Soient deux classes A et B telles que **A HERITE DE B**

• point de vue **ensembliste** (approche des types)
$\forall x, x \in A \Rightarrow x \in B$ (A est inclus dans B)
Les éléments de A sont aussi des éléments de B.
Ce point de vue est celui de la compatibilité de types dans l'approche à typage statique.

• point de vue **logique** (prédicats)
$\forall x, A(x) \Rightarrow B(x)$
Si x est un A alors x est aussi un B.
x est un objet polymorphe et peut être vu comme "un B".

• point de vue **conceptuel** (classification)
Le concept A est plus spécifique que le concept B ou A est une sorte de B.
L'arbre d'héritage constitue une classification de concepts du plus général au plus particulier. Cette approche est utilisée pour représenter des connaissances classifiables (Animaux, Végétaux, etc ...).

• point de vue **modulaire**
Les méthodes applicables aux instances de B sont un sous-ensemble des méthodes applicables aux instances de A.
Cette approche considère l'héritage comme un moyen de s'approprier les caractéristiques d'une autre classe.

• point de vue **structurel**
Les attributs de B sont aussi des attributs de A.

Ces différentes approches peuvent parfois s'opposer. Le point de vue structurel peut conduire à des héritages contre nature sous l'aspect logique. Par exemple, il est possible de faire hériter RECTANGLE de CARRE puisque RECTANGLE possède un attribut de plus: *largeur*. Cependant le point de vue logique (et la géométrie) nous dit qu'un carré est un rectangle particulier. C'est en effet l'approche structurelle qui est inexacte, il faut faire hériter CARRE de RECTANGLE en ajoutant un invariant de classe décrivant la propriété bien connue des carrés: *largeur* = *longueur*[1].
Par ailleurs, l'héritage est souvent décrit comme une extension alors que d'autres parlent de spécialisation. Ce n'est pas l'héritage qui est en cause mais bien l'approche qu'on en a. La spécialisation correspond à l'approche ensembliste pour laquelle la classe fille représente un ensemble d'instances potentielles inclus dans celui de la classe mère. Par contre, le point de vue modulaire considère que le nombre de méthodes offertes par la classe fille est supérieur à celui de la classe mère, et par conséquent qu'il s'agit d'une extension. Cette extension concerne en fait l'enrichissement du service offert par l'objet: l'objet est plus spécifique donc on a plus d'information sur lui. Les diagrammes de Venn de la méthode de Wirfs-Brock (cf chapitre 2 et [Wirfs-Brock 90b]) sont une représentation de l'héritage modulaire.

---

[1] On pourra de plus renommer *longueur* en *coté*.

Héritage d'implantation / héritage de spécification

L'héritage tel que nous l'avons défini concerne l'implantation d'une classe. Que se passe-t-il pour l'interface? Selon les langages, l'interface est directement héritée comme en TREILLIS/OWL (héritage de spécification) ou est indépendante de l'héritage comme en d'EIFFEL[1]. Dans ce cas, la classe héritière choisit d'exporter ou non les méthodes héritées. Cette approche est plus souple, mais il y a un risque d'utiliser le polymorphisme sur des objets non polymorphes. Une solution est de répartir les situations d'héritage en différentes classes (implantation, spécialisation ...) et de n'autoriser le polymorphisme que sur les héritages de spécialisation (Cf Chapitre 3).

3.3.2. Les autres concepts

Nous passons brièvement en revue les autres concepts du modèle objet. Ces notions ne sont pas implantées dans tous les langages, mais sont présentes dans EIFFEL et dans le modèle MECANO.

3.3.2.1. Classe virtuelle

Le concept de classe virtuelle est présent dans EIFFEL, BETA ou SIMULA. Une classe **virtuelle** est une classe dans laquelle l'implantation d'au moins une méthode n'est pas définie. On parle de méthode **virtuelle** ou **retardée**[2]. L'intérêt de cette notion est la possibilité de spécifier une classe sans la réaliser. La responsabilité de l'implantation est transmise aux classes descendantes. Cela permet de mettre en place des conceptions à partir de types abstraits tout en retardant le moment de la réalisation. Les classes virtuelles sont aussi utilisées pour la mise en place de propriétés (Cf §3.2.2. et §3.4.6.).

3.3.2.2. Assertion

La possibilité de définir des assertions sur les méthodes contribue à enrichir le modèle. Les assertions peuvent prendre la forme de pré- et post-conditions de méthodes, d'expressions booléennes vérifiées par tous les objets d'une classe (invariants de classe) ou positionnées dans le code des méthodes (invariants de boucle par exemple). Les assertions vont servir à traduire les axiomes et pré-conditions du type abstrait correspondant (Cf §4.6.). Elles servent aussi à décrire des invariants de la représentation, par exemple une pile implantée par un tableau doit vérifier que le nombre d'éléments qu'elle contient ne dépasse pas la taille du tableau.
Les assertions couplées à l'héritage permettent de mettre un garde-fou aux possibilités de redéfinition des méthodes héritées. Si les assertions sont héritées, les redéfinitions doivent les respecter et conservent ainsi la sémantique de la méthode initiale. Une redéfinition peut affaiblir le pré-condition et enrichir la post-condition. Le polymorphisme est alors complètement sécurisé: un client est assuré que la pré-condition reste valide et que la méthode exécutée garantira la post-condition.

3.3.2.3. Généricité

---

[1] La clause *repeat* permet néanmoins de préciser que l'interface est héritée sans avoir à réénoncer les méthodes exportées.

[2] A ne pas confondre avec la notion de méthode virtuelle de C++ ou TURBO-PASCAL 5.5 qui signifie que la méthode pourra être redéfinie dans les classes descendantes (candidates à la liaison dynamique). Par défaut, les méthodes sont liées statiquement.

La généricité, comme dans les types abstraits, permet de définir dans une seule classe générique un ensemble de classes en paramétrant la définition par un ou plusieurs **types génériques formels**. Cette notion n'a d'intérêt que dans les modèles à typage statique. Un langage à typage dynamique est générique par définition: une classe LISTE définit le comportement de toutes les listes d'éléments puisqu'aucun contrôle statique n'est fait sur le type des éléments.
Il est à noter que les notions d'héritage et de généricité sont rarement présentes dans le même langage. ADA propose la généricité sans l'héritage, C++ propose l'héritage mais pas la généricité. EIFFEL et HYBRID [Nierstrasz 87] proposent les deux notions.
De la même façon que dans les types abstraits, la généricité simple est limitée puisqu'aucune opération ne peut être appliquée aux entités du type formel. Pour lever cette limitation, EIFFEL introduit la généricité contrainte. Le problème est le suivant: on désire appliquer des opérations à des objets du type formel tout en s'assurant que le type effectif qui remplacera le type formel possèdera ces opérations (sinon le typage statique est contourné). La meilleure façon de s'assurer qu'un type possède un ensemble d'opérations est de le contraindre à hériter d'une classe fournissant ces opérations. Une telle classe est appelée **contrainte**[1]. On va donc être conduit à créer des classes particulières agissant comme contraintes sur des types formels, paramètres de types génériques.
Voici à titre d'exemple, l'entête de la classe HTABLE de la librairie EIFFEL. Cette classe décrit les tables stockant de l'information par une méthode de hashcoding.

```
class HTABLE [ T, U -> HASHABLE ]
```

T est le type des éléments stockés dans la table, U est le type des clés associées aux éléments de type T. Les clés doivent fournir une fonction de hashcode: c'est l'objet de la classe HASHABLE. Dans la même librairie, on trouve la classe STRING décrivant les chaînes de caractères. STRING hérite entre autre de HASHABLE et fournit une fonction standard de hashcode.

3.3.2.4. Héritage multiple

L'héritage multiple est la possibilité pour une classe d'avoir plus d'une classe parente (au premier degré). L'intérêt direct est que la classe héritière reçoit l'ensemble des caractéristiques de toutes les classes parentes. Cependant, plusieurs problèmes techniques et conceptuels se posent.

• problèmes conceptuels
Le graphe d'héritage n'est plus un arbre (ou une forêt); que devient dans ces conditions le point de vue ensembliste ou conceptuel de l'héritage? On imagine mal la classification animale par exemple possédant une structure de treillis[2]. Nous verrons au Chapitre 3 que cette question est un faux problème car les situations d'héritage multiple ne correspondent pas en général au point de vue conceptuel.

• problèmes techniques
Le premier problème à résoudre dans un héritage non linéaire est la détermination de l'attribut ou de la méthode à appliquer quand ceux-ci sont hérités de plusieurs classes.

---

1 L'héritage étant réflexif, le type effectif peut être la contrainte elle-même.

2 La méthode OMT (cf chapitre 2) permet des spécialisations "recouvrantes". Une classe peut alors hériter de deux spécialisations, et correspond à un sous-ensemble de leur intersection au sens ensembliste.

Les langages de l'école SMALLTALK proposent généralement des mécanismes automatiques se basant sur la liste de priorité obtenue par le parcours du graphe d'héritage. EIFFEL rejette statiquement les conflits d'héritage. Le programmeur doit alors régler les conflits à la main en renommant certaines caractéristiques. La clause de renommage est associée à la déclaration d'héritage, ce qui permet de faire un renommage sélectif.
Le second problème vient du fait qu'une même classe peut être héritée deux fois par des chemins d'héritage différents. Dans ce cas, on peut se demander si les caractéristiques héritées deux fois sont identiques ou si elles doivent être dupliquées. En général, les langages considèrent les caractéristiques comme identiques. Le renommage permet d'être plus précis: les caractéristiques à dupliquer seront renommées pour les distinguer dans le contexte de la classe héritière.

3.4. Des Types Abstraits aux Objets

Nous avons effectué un rappel sur les types abstraits au §3.2. La section précédente a présenté les concepts du modèle objet. Les deux modèles ont beaucoup de similitudes mais divergent cependant quant à l'utilisation qui en est faite. Les types abstraits sont adaptés à la spécification et la description des données du point de vue externe. Le modèle objet est plus opérationnel et fournit les moyens de réaliser des systèmes conçus comme des collections d'objets s'échangeant des messages. Nous pensons néanmoins que la démarche des types abstraits peut et doit être utilisée dans la conception par objet. Cette approche pourra garantir la rigueur des applications développées. Nous verrons au Chapitre 3 que la méthode MECANO tout comme EIFFEL favorise une telle démarche.

Dans cette section, nous étudions le rapport existant entre les types abstraits et le paradigme objet. Nous montrons comment les concepts des types abstraits s'expriment dans le modèle objet. Ces indications permettrons à un concepteur de conserver l'approche "type abstrait" pendant la conception par objets.
Un exemple complet de traduction des types abstraits vers EIFFEL est proposé en Annexe A.

3.4.1. Passage d'un modèle déclaratif à un modèle opérationnel

B. Meyer propose la définition suivante pour la conception orienté-objet [Meyer 88]:

> *"Object Oriented Design is the construction of software systems as structured collections of abstract data types implementations"*[1].

Cette définition décrit de manière condensée le passage du type abstrait au modèle objet:

- Une classe est une **implantation** de type abstrait: on réalise un logiciel, on va donc plus loin que la spécification. En particulier, les fonctions (constructeurs et transformateurs) deviennent des actions agissant par effets de bord. L'aspect spécification des types abstraits est néanmoins conservé par l'interface, les axiomes sur les méthodes et les classes virtuelles.
- Un programme est une **collection**: on privilégie l'autonomie des composants ainsi que l'aspect "bottom/up" de la conception. Les classes sont conçues de manière indépendante et peuvent ensuite être associées.

[1] La Conception Orienté-Objet se définit comme la construction de systèmes logiciels au moyen de collections structurées d'implantations de types abstraits de données.

• Un programme est une collection **structurée**: les classes sont composées par les relations prédéfinies d'héritage et de délégation (client-fournisseur)[1]. Ces relations permettent de mettre en place des architectures logicielles.

Les sections suivantes exposent comment le modèle objet permet d'implanter les types abstraits. Nous détaillons les différentes rubriques d'un type abstrait et leur traduction dans le modèle objet.

3.4.2. Classe virtuelle ou classe concrète?

"Un type = une classe"
La classe va tout naturellement jouer le rôle du type abstrait du point de vue de l'extérieur (ses clients). A se stade, on peut choisir de spécifier et représenter le type dans une seule et même classe, ou bien de définir d'abord le type abstrait dans une classe à part et réaliser les différentes représentations du type dans des classes descendantes. Dans ce dernier cas, la première classe sera virtuelle. Les classes descendantes seront concrètes et (re)définiront les méthodes. Cette deuxième approche offre l'intérêt d'être plus souple et évolutive, elle sera discutée au Chapitre 3 (§3.2.3).

3.4.3. Opérations

Le passage d'un modèle fonctionnel à un modèle opérationnel s'accompagne de la transformation de certaines fonctions en actions avec effets de bord. Dans le modèle objet, les méthodes s'appliquent toujours à l'objet courant (le destinataire du message) et les méthodes agissent directement sur l'état de cet objet. Les types abstraits sont définis par des fonctions produisant un résultat à partir d'éléments en entrée.
Les règles de transformation des opérateurs du type abstrait en méthodes de classes sont donc les suivantes:

Règles de transformation des opérateurs		
transformateur	->	*action sur l'objet*
accès: x -> t	->	*fonction ou attribut de type t*
constructeur	->	*procédure de création d'objet de la classe*

Les paramètres sont évidemment conservés sauf le paramètre du type de définition (cas des transformateurs). S'il y a plusieurs paramètres du type de définition, on choisira un paramètre particulier comme destinataire, les autres seront maintenus.
Si le type possède plusieurs constructeurs, on mettra en place une méthode de création sans paramètres et on fournira des méthodes pour initialiser l'objet selon les différents constructeurs. Par exemple, soit le type CARRE possédant les deux constructeurs:

```
carre1: point x point x point x point -> carre
     définition par les quatres sommets
carre2: point x vecteur -> carre
     définition par un sommet et le vecteur du premier côté
```

La classe correspondante offrira les méthodes:

```
Créer      création de l'objet carré, initialisation par défaut
Init1(P1,P2,P3,P4 : POINT) ...
Init2(P: POINT, V: VECTEUR) ...
```

[1] Dans la suite nous nommerons cette relation indifféremment "délégation", "client" ou "utilisation".

L'appel de l'opérateur *carre1(p1,p2,p3,p4)* sera remplacé par la séquence: *Carre.Créer; Carre.Init1.*

3.4.4. Pré-conditions et axiomes

Si le modèle disponible fournit des moyens pour exprimer (et vérifier) des assertions, les axiomes pourront eux aussi être intégrés à la conception. En général, quand il est présent, le langage d'assertions est un langage de logique d'ordre 0. Une logique d'ordre supérieur signifierait que le système de vérification est un démonstrateur de théorème sur le langage hôte, ce qui à l'heure actuelle n'existe pas[1].
Si la traduction est possible, les règles de transformation sont les suivantes:

Règles de transformation des pré-conditions et axiomes
- *pré-condition(op)*
  -> *pré-condition de méthode op*
- *axiome mettant en jeu un transformateur*
  -> *post-condition de l'action correspondante*
- *axiome sur un accès*
  -> *post-condition de la fonction correspondante*
- *axiome sur plusieurs accès*
  -> *post-conditions sur les fonctions correspondantes ou invariant de classe*
- *axiome sur un constructeur*
  -> *post-condition de la méthode de création de la classe*

La traduction de l'axiome en post-condition se fait en considérant la partie gauche comme réécriture (définition) de la partie droite. Réécriture signifie qu'on simplifie un terme en remplaçant un sous-terme équivalent à la partie gauche d'une équation, par la partie droite de l'équation. Un exemple de réécriture est donné en Annexe A.

Prenons le cas d'un axiome mettant en jeu la définition d'un accès sur un transformateur:

équation

A(T(t,u1,u2,...),v1,v2,...) ==
si EB(t,u1,u2,...,v1,v2,...) alors F(t,u1,u2,...,v1,v2,...)

- *t est une variable du type en cours de définition TD*
- *u1,u2,...,v1,v2,... sont des variables d'autres types que TD*
- *A est un accès*
- *T est le transformateur (ou constructeur) candidat à la post-condition*
- *EB est une expression booléenne*
- *F est une expression dont le type est le même que le résultat de A contenant éventuellement des accès (réduction du transformateur).*

La post-condition pourra être exprimée si les paramètres de EB et F ne mettent en jeu que les variables u1,u2,... ou des variables s'y ramenant (logique d'ordre 0).

---

1 Une étude sur l'utilisation de PROLOG comme langage d'assertion d'EIFFEL est en cours à l'Université de Marseille-Lumigny.

La formulation de la post-condition de l'opérateur T devient:

post-condition

postcond(T) = EB(t-avant,u1,u2,...,v1,v2,...) =>
A(t_apres,v1,v2,...) = F(t-avant,u1,u2,...,v1,v2,...)

La variable *t-avant* désigne la valeur de l'objet avant la transformation.
La variable *t-après* désigne la valeur de l'objet après la transformation[1].
Se reporter à l'exemple de l'Annexe A pour une illustration complète.

3.4.5. La généricité

La généricité se traduit directement si le modèle objet possède cette notion. Les paramètres formels sont généralement déclarés avec le nom de la classe. Ils peuvent ensuite apparaître comme type des paramètres de méthodes, type d'attribut, type de résultat de fonction, ou encore comme type de variable locale. La construction est purement syntaxique, le paramètre étant instancié à la compilation dans une classe utilisatrice.
Si le modèle n'offre pas la généricité, il existe des moyens pour la simuler [Meyer 88 pp 410-421].

3.4.6. Les propriétés

Une propriété va être implantée par une classe généralement virtuelle (ce n'est pas un type abstrait de données mais la description d'un comportement). Cette classe exporte les opérateurs de la propriété. Les opérateurs fondamentaux ne sont pas réalisés (méthode *deferred* d'EIFFEL ou méthode *virtuelle* de SIMULA). Les opérateurs dérivés sont implantés en terme des opérateurs fondamentaux. Les axiomes correspondants sont ajoutés en post-condition ou invariant si c'est possible.

- **Point de vue du type générique**

Le type générique exige une propriété sur le type formel pour utiliser les opérateurs de cette propriété. Dans la classe correspondante, on indique que le type formel est contraint par la propriété (il hérite de la classe implantant la propriété).

- **Point de vue du type effectif**

Le type effectif doit possèder la propriété. En conséquence, les opérateurs de la propriété sont disponibles dans le type effectif avec l'axiomatisation correspondante. Pour s'assurer de ces deux contraintes, une solution est de faire hériter le type effectif de la classe implantant la propriété[2]. De cette façon, l'axiomatisation est directement héritée. Les méthodes correspondant aux opérateurs fondamentaux sont virtuelles dans la propriété, le type effectif doit les implanter dans son contexte. Les méthodes correspondant aux opérateurs dérivés sont réalisées dans la classe propriété.

---

[1] En général l'objet n'interviendra pas comme paramètre d'opérateur mais comme destinataire du message correspondant. Par exemple en EIFFEL, la valeur d'un objet avant l'exécution peut être obtenue par ***old*** *ref-objet* et l'objet courant est désigné par **Current**.

[2] Ce type d'héritage est qualifié de **Comportement** dans MECANO (Cf Chapitre 3).

Une autre solution serait de vérifier que le type effectif exporte bien les mêmes opérateurs que la propriété, mais il faudrait alors s'assurer que l'axiomatisation est équivalente (démonstration de théorèmes).

- **Inclusion de propriété**

On définit l'inclusion de propriété par [Jacquet 78]:

soient P et P'' deux spécifications de propriété,
P P' <=> les fonctions de P sont définies dans P'
les axiomes de P sont présents dans Pp' (ou déductibles)

Cette notion permet de définir des propriétés par rapport à d'autres. Par exemple, la propriété **Monoide** est incluse dans **Groupe**, **Groupe** est inclus dans **Anneau**.
L'inclusion de propriété sera naturellement implantée par héritage entre les classes virtuelles correspondantes (spécialisation).

Un exemple complet de passage des types abstraits au modèle objet est donné dans l'Annexe A.

3.5. Conclusion

Cette partie a permis de présenter le lien entre le modèle objet et le modèle des types abstraits. Nous avons insisté sur l'approche à typage statique qui est celui que nous adoptons dans la suite. Ce modèle oblige à bien séparer la description du programme (les classes) du modèle exécutif (les objets) et se prête à l'approche "implantation de types abstraits". Les types abstraits fournissent un critère de cohésion très fort aux modules (classes) de l'approche objet. Si le passage des types abstraits aux classes est relativement aisé, il peut cependant donner lieu à une perte de sémantique (axiomes inexprimables). Ceci est incontournable puisqu'un langage objet n'est pas un système de programmation automatique.
Mais le modèle objet apporte une notion absente des types abstraits: l'Héritage. Cette notion est importante pour assurer certaines qualités au logiciel.

4. CONCLUSION

4.1. L'approche objet: un modèle fédérateur

Dans ce premier chapitre, nous avons essayé de placer notre étude dans le champ très vaste de la recherche en génie logiciel. L'étude des facteurs et des critères de qualité permettent de mieux situer la contribution du modèle objet en général et de notre étude en particulier. Cet apport concerne la mise en place de techniques de développement apportant qualité et rigueur à la production de logiciels, activité encore très souvent empirique. Dans la suite, nous nous attacherons toujours à rattacher les concepts présentés au point de vue de la qualité et du génie logiciel. En cela, nous adoptons l'approche de B. Meyer et de l'école dite "scandinave" des langages statiquement (et fortement) typés [Brien 87], [Kristensen 87b], [Meyer 89a] en nous démarquant d'approches plus "dynamiques" comme celles de SMALLTALK [Goldberg 83] ou LOOPS [Bobrow 83], langages privilégiant l'un le modèle objet en tant que paradigme unique de programmation, l'autre le modèle objet en tant que représentation des connaissances en Intelligence Artificielle.

Comme l'a montré J. Bezivin [Bezivin 90], le modèle objet apparait comme élément fédérateur de disciplines informatiques variées. L'approche objet est appliquée dans les Bases de Données [Adiba 89a], [Adiba 89b], l'Intelligence Artificielle [Malverti 88], le Parallélisme [Papathomas 89] et [De Mare 90], les Systèmes d'Exploitation [Krakoviak 87], les Langages de Programmation [Stroustrup 87], la Conception [Booch 83], les Ateliers de Génie Logiciel [Estublier 88]. De là à penser que ce modèle est en train de devenir, au delà des modes, le concept central de l'informatique de demain, il y a encore un grand pas à franchir ne serait-ce que parce que si tous se réclament du **"modèle objet"**, la sémantique et la finalité associées aux concepts sont souvent radicalement différentes. En effet, les chercheurs en Bases de Données voient dans l'objet un moyen de représenter une information rémanente et sont souvent embarrassés par les fonctions non élémentaires que tout objet est en droit de posséder ainsi que par le concept de classe.
Dans le modèle centré-objets, l'Intelligence Artificielle considère l'héritage comme un moyen de classification permettant de représenter des connaissances en privilégiant l'aspect déclaratif et ne considérant pas ou très peu l'aspect opérationnel des objets.
Dans le domaine du parallélisme les objets sont des processus ou des acteurs envoyant et recevant des messages.
Les systèmes d'exploitations gèrent des objets correspondant aux ressources physiques dans un environnement distribué.
Les langages de programmation encapsulent les données et les fonctions dans les classes et assurent le passage du flux de contrôle inter-objet par des envois de messages en autorisant une édition de lien dynamique.
Les structures d'accueil d'atelier logiciel gèrent des objets tels que les programmes, les documents, les versions et les configurations en tenant compte des problèmes d'évolution de ces objets pendant toute la durée de vie d'un système.
Enfin, la conception s'intéresse aux abstractions que représentent les classes, et à leur spécification, à leur organisation en graphe d'héritage et de délégations pour élaborer des logiciels évolutifs et réutilisables.

Notre approche dans cette étude sera l'utilisation du modèle objet dans le contexte de la construction des logiciels (programmation et conception). Nous verront que ce modèle permet de rapprocher le développement logiciel du développement matériel, par la mise en place de composants réutilisables et réutilisés de granularité suffisamment grande.

Nous nous attacherons à montrer comment les critères de modularité et de généralité sont produits par la conception par objets.

4.2. Approche objet et qualité du logiciel

Pour conclure cette partie, nous présentons l'impact de l'approche objet sur les qualités des logiciels produits. Les concepts objets ont été introduits au §3. de ce chapitre. Les critères énoncés au §1. sont repris en indiquant quelles notions orienté-objet permettent de les obtenir. Il faut cependant rester réaliste et l'approche objet n'entraîne pas automatiquement les critères de qualité même si elle offre des notions qui aident à les atteindre. Comme tout paradigme de programmation, l'approche objet peut être mal utilisée et conduire à des produits non fiables, non réutilisables ou impossibles à maintenir. C'est la finalité d'une méthode que d'indiquer la manière de mettre en oeuvre les concepts pour atteindre la qualité. Nous pensons que la méthode proposée aux chapitres 3 et 4 apporte une réponse à ces besoins.

Les critères de qualités du §1. sont repris. Pour chacun d'eux, nous précisons les concepts du paradigme objet qui tendent à les favoriser.

• La Complétude

L'espace des solutions est proche de l'espace du problème. Le passage de l'un à l'autre (la conception) en est simplifié car les objets du monde réel ont une existence dans la solution informatique. Une méthode systématique d'extraction des classes et des opérations telle que celle proposée par Booch (Cf Chapitre 2) contribue à la complétude de la solution.
D'autre part, les aspects fonctionnels, les aspects "contrôle" du système et l'interface utilisateur sont différés le plus tard possible dans la conception. Leurs modifications étant facilitées, la solution pourra être rendue complète plus facilement car les oublis de fonctionnalités ne remettent pas en cause la structure du logiciel.

• La Consistance

Le formalisme et les concepts étant unifiés entre la conception et la réalisation, la consistance est directement obtenue. Que l'on mette en place une architecture générale d'un système de surveillance aérienne (RADAR, PISTE, AVION) ou que l'on programme le stockage d'informations élémentaires dans une table de hashcoding (TABLE, HASHABLE), le formalisme et les notions utilisées sont les mêmes (Classe, Méthodes, Héritage, Généricité, Utilisation). Les objets de haut-niveau (proches de l'analyse) cohabitent avec les objets de bas niveau (proches de l'implantation).
Les classes virtuelles permettent aussi d'assurer la consistance. Une classe virtuelle peut être vue comme une spécification de type abstrait (classe GRAPHE en Annexe A), les classes filles concrètes sont alors des représentations de ce type abstrait à partir desquelles il est possible de créer des instances qui seront actives à l'exécution. Par cette méthode, on assure la consistance entre la spécification et la réalisation par le lien d'héritage. Dans ce cas, il conviendra cependant de restreindre le lien à l'héritage de réalisation (Cf Chapitre 3) pour assurer que le service offert reste le même.

• L'Auto-documentation

Le modèle objet unifie spécification et réalisation dans un même objet: la Classe. En EIFFEL par exemple, les assertions, l'interface, l'invariant ou l'indexation font partie du code source [Nerson 91]. Ces informations peuvent être extraites automatiquement par un outil (SHORT). De même, les relations (clientèle ou héritage) sont calculées (outil GOOD et FLAT).
De la même façon, le Poste de Conception MECANO intègre la documentation dans les descripteurs de classe (spécification, interface, mot-clés, invariant, pre- et post-conditions).

• **La Lisibilité**
La lecture d'un programme objet est aisée. Les objets du programme représentent la plupart du temps des objets du monde réel concrets ou abstraits (Tortue, Ordre) ou bien des objets informatiques bien connus (Table, Liste, Graphe, Dictionnaire). La cohésion modulaire d'une classe est naturelle et permet de comprendre une classe indépendamment du reste du système. Les relations inter-classes sont bien définies et conduisent généralement à un couplage minimum (1 ou 2 parents en moyenne et moins d'une dizaine de fournisseurs). Les niveaux d'abstraction offerts par l'héritage ou la composition de classes apporte une lecture incrémentale. Enfin, le nombre restreint de concepts du modèle permet une appropriation rapide de ceux-ci.

• **La Modularité**
L'approche objet remplit les sous-critères de modularité:

- Décomposition modulaire: la classe, implantation de type abstrait est le module. Le couplage est bas: les échanges sont limités à l'utilisation (attributs d'une classe) et à l'héritage. Même en cas d'héritage multiple, les parents d'une classe ne sont rarement plus de 2 ou 3. La cohésion est assurée par la modularité naturelle des types abstraits implantés.
- Composition modulaire: on peut combiner des éléments par généricité, héritage et utilisation.
- Compréhension modulaire: l'interface des classes spécifie l'abstraction représentée. Les classes sont autonomes et indépendantes. En EIFFEL, le fait que le service doit être redéfini pour chaque classe et non hérité conduit à une meilleure compréhension de la classe fille.
- Continuité modulaire: l'approche orienté-objet organise les logiciels sur les composants les plus stables dans le temps : les (abstractions de) données. L'héritage et la généricité apporte une certaine évolutivité à travers la généralisation.
- Protection modulaire: la philosophie du contrat passé entre classe cliente et fournisseur permet de déterminer les devoirs et responsabilité. EIFFEL propose un mécanisme d'exceptions discipliné basé sur la notion de contrat.

• **La Réutilisation**
- Variante dans les types: généricité et simple et contrainte assurent la paramétrisation des types.
- Variante dans les structures de données et d'algorithmes: l'héritage de réalisation et les classes virtuelles permettent de mettre en place des schémas de variantes.
- Indépendance de la représentation: le masquage d'information (interface) et les assertions permettent de spécifier un type abstrait indépendamment d'une réalisation concrète. Le polymorphisme autorise la liaison dynamique et l'exécution du code d'une méthode en fonction du type dynamique de l'objet concerné.
- Mise en commun dans les sous-groupes: l'héritage de spécialisation et les classes virtuelles apportent les moyens de capturer les informations au niveau d'abstraction voulu, et assurent une définition incrémentale des concepts.

4.3. MECANO et son poste de conception: approche informelle

Les chapitres 3 et 4 présentent la méthode que nous avons développée. Le besoin d'une nouvelle méthode s'est fait sentir par l'absence de méthode réellement orienté-objet adaptée à une implantation dans un langage tel qu'EIFFEL. La programmation orienté-objet est une activité relativement jeune et les méthodes sont toujours développées après les langages. Cet état de fait est en train de changer, la publication d'articles et d'ouvrages récents sur l'analyse et la conception par objets en atteste.

L'émergence d'EIFFEL a fait prendre conscience à de nombreux programmeurs, enseignants ou chercheurs qu'au delà des modes, l'approche orienté-objet constitue un véritable pari pour les années à venir. Avec EIFFEL, la programmation orienté-objet devient une évolution de la programmation classique vers la rigueur, la fiabilité, la réutilisabilité et la maintenabilité [Meyer 87a]. Les types abstraits sont sous-jacents et la rigueur du langage oblige à un style de programmation plus propre et plus pur. Ce n'est pas un hasard si en France, le langage EIFFEL commence a être utilisé pour l'enseignement de la programmation [Michel 88].

Conquis par ce langage, nous avons décidé de pousser plus loin les concepts proposés. Cette recherche a donné naissance à la méthode MECANO. La méthode offre une plus grande richesse dans la manipulation de l'héritage et de la délégation, et fournit des concepts permettant de structurer les applications en tenant compte de l'évolution et de la réutilisation. Au delà des langages, le concepteur a besoin de méthodes fournissant un cadre sémantique plus riche. Des règles et un processus doivent lui permettre d'utiliser les concepts le mieux possible afin d'obtenir les qualités qu'il souhaite.

Convaincu qu'il ne peut guère exister de méthode sans environnement de support, un poste de conception associé à la méthode MECANO a été défini. Nous avons voulu le rendre le plus convivial possible. Pour cela, il s'appuie sur la richesse de l'interface des stations de travail actuelles, combinant édition graphique, mode de travail ouvert, multi-fenêtrage, menus fugitifs et manipulation d'icônes.

Nous espérons que le lecteur sera intéressé par cette contribution et qu'il trouvera matière à réflexion sur sa propre expérience de concepteur d'applications.

CHAPITRE 2:

CONCEPTION PAR OBJET

Intuitively, it is clear that the closer the solution space maps to our concept of the problem space, the better we can achieve our goals of modifiability, efficiency, reliability, and understandability[1]. *[Booch 83]*

ETAT DE L'ART

---

[1] "Intuitivement, il est clair que plus l'espace de la solution s'adapte aux concepts de l'espace du problème, plus nous pourrons facilement atteindre les buts de modifiabilité, d'efficacité, de fiabilité et de compréhensibilité".

1. INTRODUCTION

1.1. But de l'étude

Ce chapitre se propose d'offrir un panorama des méthodes existantes dans le domaine du développement de logiciels utilisant une approche objet. A travers une présentation et un bilan critique des méthodes, nous nous efforçons de montrer les évolutions actuelles. La recherche dans ce domaine est très productive et ces deux dernières années ont vu la naissance de nouvelles méthodes s'adressant aussi bien à l'analyse qu'à la conception.
Nous montrons les apports de ces nouvelles méthodes ainsi que leurs lacunes. Ceci nous conduira à situer nôtre contribution, la méthode MECANO, parmi les autres approches afin d'en distinguer l'originalité.

1.2. Analyse, Conception et Implantation

Une méthode qu'elle soit d'analyse ou de conception est constituée d'un modèle et d'un processus. Le modèle définit un formalisme généralement graphique et un ensemble de règles. Les règles garantissent le respect des concepts de la méthode et assurent la cohérence d'un "état" du développement. L'activité de développement de logiciel peut être vue comme un processus constitué d'actions élémentaires [Girod 88] assurant le passage d'un état du développement à un état plus détaillé et plus proche de l'implantation. L'état initial est généralement constitué d'une spécification des besoins (énoncé du problème). Le modèle permet de définir un état de conception, alors que le processus définit la dynamique de conception.

1.2.1. Conception et Analyse

En général, les méthodes d'analyse et les méthodes de conception ne reposent pas sur le même modèle. Par exemple, SADT [Ross 85] est un modèle d'analyse par décomposition fonctionnelle et MACH [IGL 84] une méthode de conception par machines abstraites.
L'approche par objets réconcilie analyse et conception à travers un modèle unifié: le modèle objet. Il est maintenant établi que celui-ci est générateur de qualités logicielles telles que la consistance et la maintenabilité. Le modèle étant le même, une méthode d'analyse se réduit donc à un processus d'analyse dans lequel on indique au concepteur comment trouver les objets, les opérations ou les interactions entre objets. La conception est souvent alors vue comme un enrichissement du modèle d'analyse pendant lequel les objets définis à l'étape d'analyse sont réalisés, y compris au moyen de nouveaux objets plus proche de l'implantation [Rumbaugh 91]. Dans cette optique, nous exposons les méthodes d'analyse qui peuvent être vues comme des méthodes de conception préliminaire.
La phase d'implantation si elle utilise un langage orienté-objet se réduit à une traduction de la conception détaillée dans le langage de programmation choisi. Le développement par objets peut donc être vu comme un processus continu, dans lequel un modèle d'analyse est peu à peu enrichi (figure 1). La frontière entre les étapes du cycle de vie s'estompe. La seule différence réside dans les objectifs poursuivis dans chaque phase. L'analyse sera une phase descriptive (propriétés des entités) tandis que la conception apportera des solutions techniques en tenant compte de critères tels que la réutilisabilité et la flexibilité. La définition de Coad et Yourdon illustre bien cette approche:

" L'analyse orienté-objet consiste à identifier et définir les classes et objets qui reflètent directement l'espace du problème et les responsabilités du système,
la conception orienté-objet consiste à identifier et définir de nouvelles classes reflètant une implantation des besoins, ces classes permettent de réaliser le dialogue, la gestion des tâches et la gestion des données" [Coad 91]

1.2.2. Conception et Implantation

La puissance conceptuelle de certains langages à objets permet d'exprimer à la fois l'architecture, les composants et les programmes exécutables. De plus, la généralisation qui est privilégiée permet la mise en place de modèles de systèmes directement codables et déclinables à volonté. Nous pensons que, malgré tout, l'aspect langage et l'aspect conception doivent rester sur des niveaux conceptuels différents. Ce choix est dicté par les raisons suivantes:

• Soucis de généralité et d'universalité:
Une méthode doit être le plus possible indépendante d'un langage et doit pouvoir s'adapter à différents langages.

• Granularité des objets:
Une méthode manipule des entités de granularité forte (module, sous-systèmes) et permet de mettre en place une conception globale. Le langage n'intervient qu'en phase de conception détaillée, la granularité descend à l'instruction.

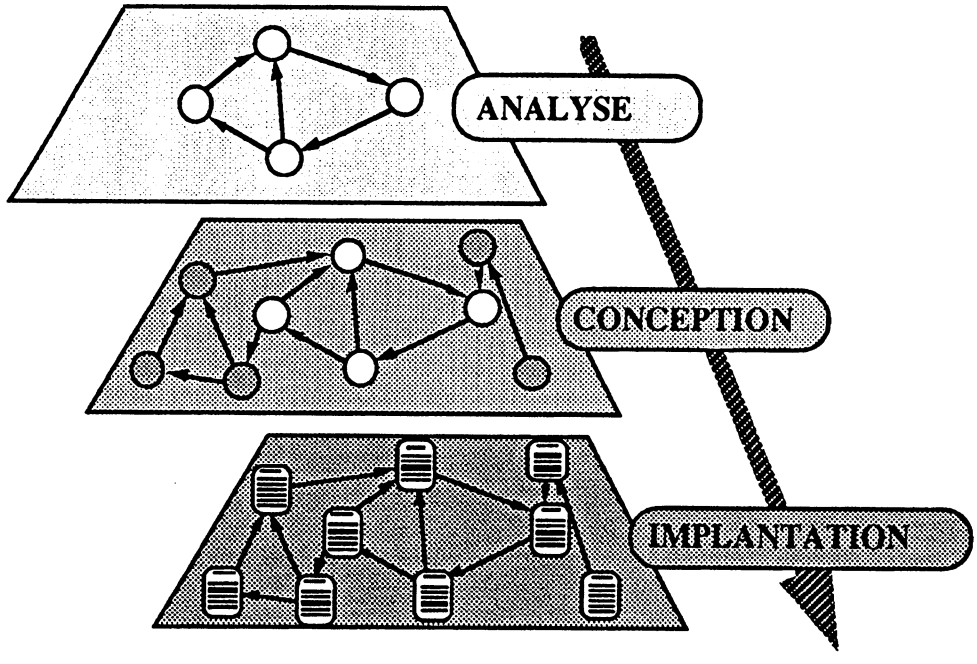


figure 1: Le cycle continu du développement par objets

• Aspect technique des langages:
Au niveau langage, on parle d'implantation, de traduction, de compilation et d'exécution. La conception, elle, est vue sous un aspect créatif plus informel. Elle s'intéresse aux

choix du concepteur, à la signification des objets. La finalité est différente: la conception résout un problème en créant une structure logicielle alors que l'implantation réalise un programme exécutable. La conception est une phase amont plus proche de l'homme.

Notre contribution concerne cette phase de conception par objets. Les concepts utilisés sont ceux du domaine de la programmation orienté-objet que nous avons étendus au cadre de la conception. Nous n'avons pas pris en compte la phase d'analyse en tant que telle, c'est à dire que nous ne donnons pas de guide pour le choix des objets à définir. Par contre le modèle objet de MECANO offre un cadre sémantique fort pour la mise en place d'architecture de conception privilégiant les qualités de réutilisabilité et d'évolutivité. Notre démarche est comparable à celle de J.M Nerson [Nerson 91] dont nous présentons l'approche dans la suite (§3.5.).

2. L'APPROCHE "OBJECT ORIENTED DESIGN"

2.1. Introduction

Nous présentons dans cette partie une étude des différentes méthodes se réclamant de la l'approche OOD. Historiquement, la méthode OOD est la première méthode préconisant une approche orienté-objet pour la construction des applications [Booch 83]. Elle propose une analyse classique (SADT par exemple), une conception par objets et une réalisation dans un langage classique. La méthode OOD et les méthodes dérivées telles que HOOD et MACH2 ont pour langage d'implantation cible ADA ou LTR3. Ceci empêche de considérer des concepts clés tels que les modèles d'objets (classe) ou l'héritage, présents dans les langages à objets.
Nous présentons pour commencer la méthode Object Oriented Design de G. Booch. On peut considérer cette méthode comme la base des autres méthodes présentées dans cette partie.

2.2. La méthode BOOCH ou "Object Oriented Design"

2.2.1. Présentation

Dans la méthode OOD de G. Booch [Booch 83], [Booch 86], l'axiome de base est le suivant : "Tout objet du système représente un module". Chaque objet est défini comme une abstraction de donnée. Cette méthodologie est indépendante du langage bien que fortement axée sur ADA et permet à travers un ensemble d'étapes de conduire à des applications réalisées dans un "style" orienté-objet. La méthode préconise la démarche suivante en 3 étapes:

- **Etape 1 : Définir le problème**

C'est la phase d'analyse traditionnelle menée par exemple par une méthode telle que SADT [Ross 85]. Elle permet de définir le problème à résoudre de manière formelle.

- **Etape 2 : Développer une stratégie informelle**[1]

Au moyen de la langue naturelle, le concepteur doit s'efforcer de décrire une première solution, appelée stratégie informelle, au problème posé à l'étape 1.

- **Etape 3 : Formaliser la stratégie**

Au moyen de règles simples, le concepteur identifie les objets et leurs attributs. Pour cela, Booch propose de repérer les noms dans le texte de la stratégie informelle et de les relier aux objets en employant les règles suivantes :

---

[1] NDLA: il s'agit d'une stratégie **de résolution.**

NOM COMMUN
-> nom d'une classe d'objets
(Capteur, Table, ...)
NOM PROPRE ou nom de référence directe
-> nom d'une entité spécifique (objet)
(ma table, le capteur de la porte ...).
ADJECTIF
-> attribut associé à un objet

Puis le concepteur identifie de la même façon les opérations sur les objets :

VERBE
-> action ou opération à associer aux objets identifiés
ADVERBE
-> attribut d'opération
(contrainte de temps, séquence de contrôle, boucles)

Enfin, le concepteur établit les interfaces des objets en déterminant pour chaque objet quelles seront les opérations proposées aux autres objets puis les traduit dans le langage de programmation (interface du package ADA). Les opérations sont ensuite réalisées dans le langage de programmation en tenant compte des contraintes détectées dans la stratégie informelle (body du package ADA).
Le processus peut être itéré pour décomposer les modules ainsi construits et obtenir un niveau d'abstraction suffisamment bas.

2.2.2. Commentaires

La méthode de Booch apparait comme une alternative séduisante à l'approche fonctionnelle modulaire ou non. Elle prend comme langage cible un langage modulaire et procédural classique : ADA. La méthode insiste surtout sur le processus de conception et propose essentiellement des règles permettant de passer d'une spécification informelle à une conception objet. Le processus pour passer des spécifications à la conception a le mérite d'être simple et clair mais reste incomplet, en particulier parce que le mécanisme d'héritage n'est pas pris en compte et qu'aucun cadre formel à part celui du langage ADA n'est proposé. Cette approche ne propose pas non plus de nouveaux concepts. Plus qu'une méthode, on parlera d'une "école" puisque cette approche est à la base de nombreuses méthodes qui introduisent de nouvelles notions pour structurer la conception. Nous en présentons deux dans la suite de ce chapitre.

2.3. HOOD : Une méthode hiérarchique

2.3.1. Définition

HOOD signifie "Hierarchical Object Oriented Design" ou conception orienté-objet hiérarchisée ([Heitz 87], [CCM 87] et [Gendre 90]). C'est une méthode de conception descendante orienté-objet inspirée de la méthode OOD de Booch et de la méthode par machines abstraites MACH [IGL 84]. Elle est plus particulièrement destinée au temps réel, ainsi qu'aux gros logiciels scientifiques et techniques avec développement réparti, réalisés en ADA. Elle reprend les principes de la méthode Booch en ajoutant des concepts de structuration et d'expression du contrôle (temps réel) propre aux machines abstraites. Les principes de structuration sont les suivants :

- *Principe d'encapsulation* : composition interne cachée et interface externe publique.

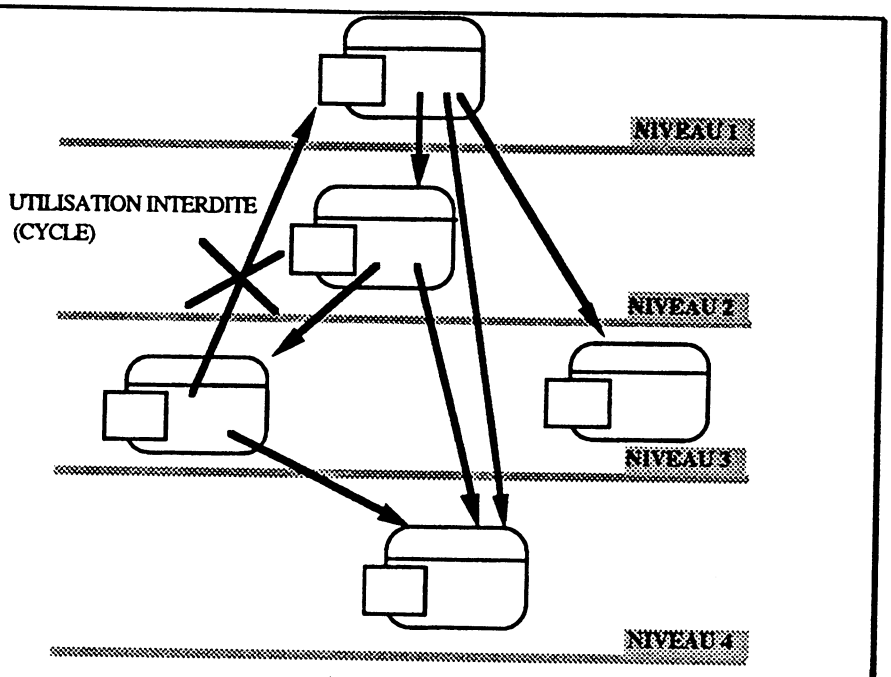


figure 3: relation "USE" ou "junior-sénior"

Les notions de classe, de généricité et d'héritage ne sont pas proposées.

• **L'Objet** (figure 4)

Il est composé d'une partie publique - l'interface - et d'une partie privée - le corps -.

- Interface : spécification des opérations et des ressources en import et export, paramètres et types. On remarquera que sont exportables non seulement les opérations mais aussi les données et les définitions de types de données.
- Corps : implantation de l'interface, objets enfants, ressources et opérations internes déterminant l'état de l'objet. Pour les objets actifs, la synchronisation et les interactions entre opérations sont définies au moyen de la clause OBCS (OBject Control Structure).

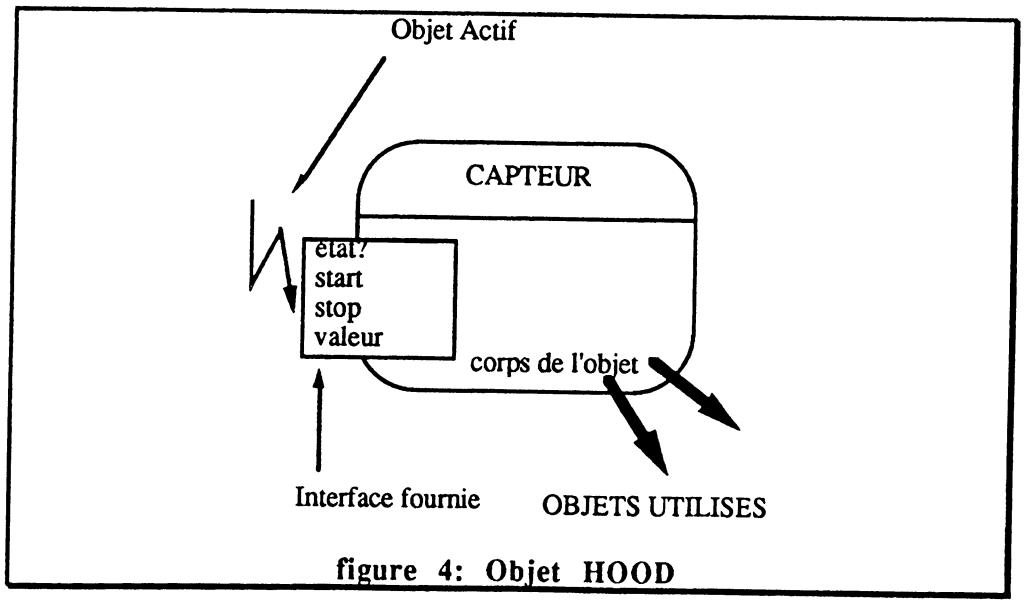


figure 4: Objet HOOD

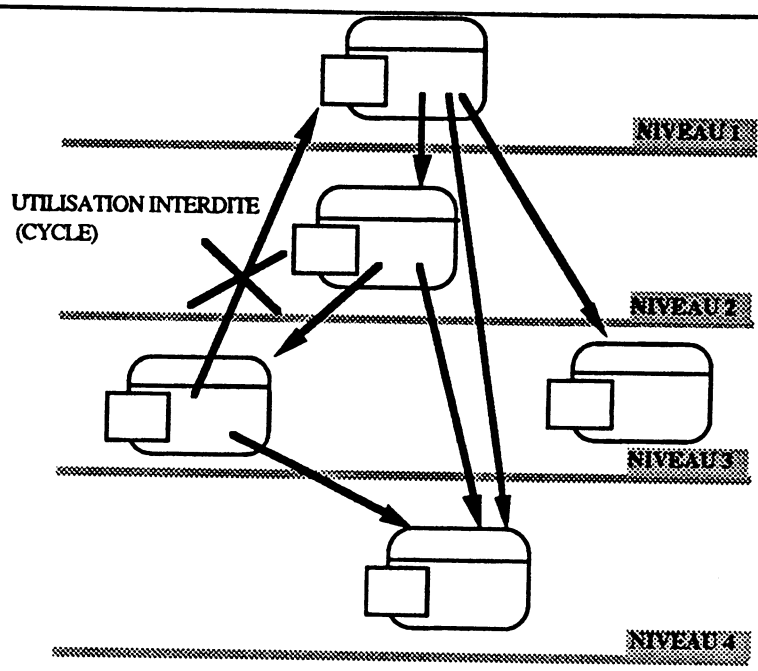


figure 3: relation "USE" ou "junior-sénior"

Les notions de classe, de généricité et d'héritage ne sont pas proposées.

• **L'Objet** (figure 4)

Il est composé d'une partie publique - l'interface - et d'une partie privée - le corps -.

- Interface : spécification des opérations et des ressources en import et export, paramètres et types. On remarquera que sont exportables non seulement les opérations mais aussi les données et les définitions de types de données.
- Corps : implantation de l'interface, objets enfants, ressources et opérations internes déterminant l'état de l'objet. Pour les objets actifs, la synchronisation et les interactions entre opérations sont définies au moyen de la clause OBCS (OBject Control Structure).

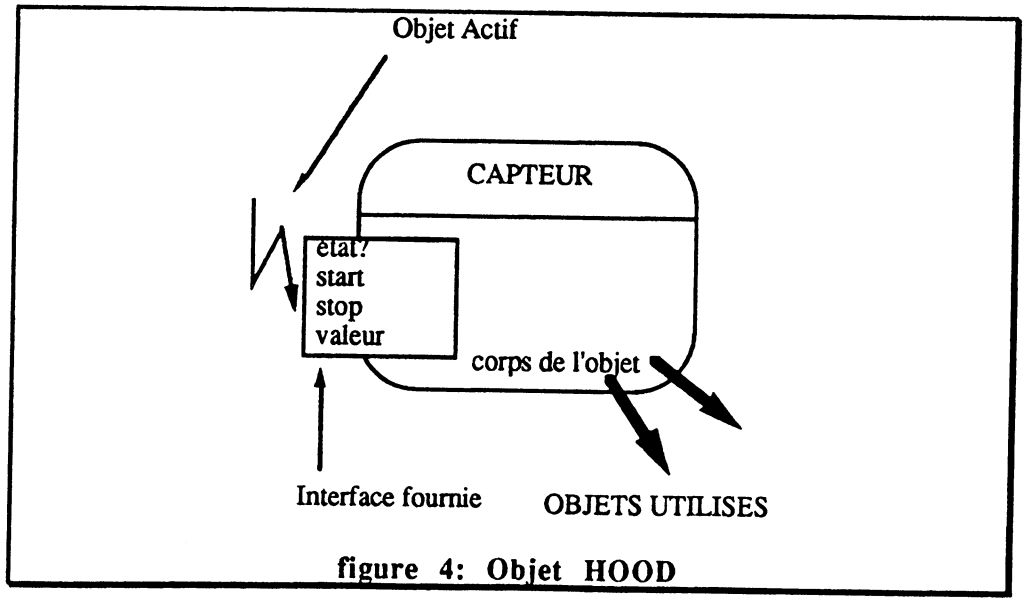


figure 4: Objet HOOD

• **Expression du contrôle :**
Pour pouvoir traiter les applications parallèles, les notions d'objet passif et d'objet actif ont été introduites. Cette distinction dépend des propriétés dynamiques de l'objet :
- exécution séquentielle classique : l'objet est dit passif (machine). L'objet appelant attend la fin de l'exécution.
- exécution concurrente : l'objet est dit actif (processus), le contrôle dépend de l'état interne et de l'interaction des autres flots de contrôle à l'intérieur de l'objet. Il est décrit dans l'OBCS associé à l'objet.

2.3.3. Stratégie de conception

Une conception est une arborescence Parent/Enfant d'objets actifs et passifs.
HOOD reprend à la fois les principes de l'OOD en les adaptant au formalisme proposé et la stratégie de conception descendante (par raffinements successifs) ayant fait ses preuves dans la méthode par machines abstraites MACH. Les objets actifs se retrouvent au sommet et se décomposent en objets passifs aux niveaux inférieurs.
Bien que l'identification des objets soit généralement menée en utilisant l'OOD, le formalisme et la méthode autorisent d'autres approches. Le concepteur peut décrire en HOOD des modèles de systèmes plus classiques: machines en couche, BD partagée, structuration par la communication ou par les fonctions etc ...

2.3.4. Commentaires

HOOD propose une dimension structurante (INCLUDE) à la méthode OOD. Les concepts d'objets actifs et passifs permettent de mettre en oeuvre des conceptions temps-réel. La distinction entre les relations USE et INCLUDE nous paraît importante au niveau de la conception. Elle permet de faire la part des choses entre les composants d'un objet et les objets offrant simplement un service aux autres objets. Cette distinction est généralement absente des langages à objets qui implantent ces deux types de relation par l'envoi de message. Une certaine uniformité des concepts est présente en particulier en ce qui concerne les objets actifs et passifs. Cependant comme dans l'OOD, l'instanciation et l'héritage de classe sont absents.

2.4. MACH2 : De la machine abstraite à l'objet

2.4.1. Définition

MACH2 [Henry 89] est une méthode de conception graphique et textuelle orienté-objet inspirée de l'OOD et de la méthode MACH. Elle s'adresse à la conception globale et là a conception détaillée des applications implantées en ADA ou LTR3. MACH2 utilise cinq types d'objets prédéfinis permettant de prendre en compte les problèmes du temps-réel: sous-système, point de communication, processus, service et type de donnée. La mise en place de ces objets constitue la phase de conception architecturale. Les objets sont structurées au moyen de deux relations : composition et utilisation. La stratégie de base utilise un cycle de conception permettant d'identifier et de structurer les objets. La conception détaillée est effectuée dans un PDL-ADA[1].

2.4.2. Modèle objet utilisé

Mach utilise le modèle habituel de OOD en ajoutant des types d'objets prédéfinis. La notion de modèle/instance est présente bien que non obligatoire: le concepteur peut

[1] Langage de conception détaillée utilisant un pseudo code proche d'ADA et de LTR3

définir des "types" de sous-systèmes, de processus ou de services. L'héritage et la généricité ne sont pas pris en compte.

• **L'Objet**
Tout type d'objet comprend deux parties : l'interface qui contient les informations visibles des autres types d'objets; le corps qui contient les informations cachées aux autres types. Les cinq catégories de type d'objet sont les suivantes :

- le **Sous-système** est une sous-application communicante parallèle ou séquentielle.
- le **Point de communication** correspond à une description des échanges entre sous-systèmes et/ou processus. Il permet de contraindre les accès entre objets.
- le **Processus** est une entité active parallèle synchronisée par rendez-vous. Il peut échanger des messages et faire appel à un service.
- le **Service** est un ensemble de ressources propres (données + opérations) gérées séquentiellement de manière interne. Il peut faire appel à d'autres services. Ce concept est proche des machines abstraites de MACH.
- le **Type de donnée** est local à un processus ou à un service. Il représente un type abstrait.

Les objets sont finalement structurés selon deux axes :
• *Axe structurel ou "Est défini par":* un type de sous-système se décompose en types de sous-système et en instances de processus et services; un type de processus/service est composé de types de données. Les types de données s'appuient eux-mêmes sur d'autres types de données.
• *Axe fonctionnel ou "Utilise" :* au plus haut niveau de cet axe, on retrouve les architectures de processus communicants éventuellement via les sous-systèmes[1]. Le deuxième niveau est constitué d'architectures de services (une architecture par processus). Ces architectures peuvent avoir des intersections non vides dans le cas de services partagés donnant lieu à une synchronisation indirecte. Le dernier niveau correspond aux modules de types de données permettant de décrire les données internes aux processus et services.

2.4.3. Stratégie de conception

Un état de conception est constitué par une représentation graphique comportant des types d'objets appartenant aux cinq catégories citées, structurés par les deux relations de définition et d'utilisation.

La méthode MACH2 préconise une stratégie de conception en cinq étapes.

• *phase 1) définition de l'interface avec l'environnement*
Il s'agit, à partir du document de spécification, de décrire l'interaction du système avec l'extérieur.

• *phase 2) détection des objets ressources*
A partir d'une lecture exhaustive des spécifications, le concepteur définit les objets ressources qu'il devra manipuler. Pour chacun d'eux, les caractéristiques (actions et attributs), les contraintes et les interactions sont dégagées. Cette phase reprend la stratégie proposée par Booch.

• *phase 3) identification des objets de structuration*

---

[1] Cet axe est très proche de l'architecture proposée dans la méthode AMPHI [Girod 87b].

A partir de la mise en place du parallélisme, les actions sont structurées en (type de) sous-système et processus. Les ressources sont implantées dans des services ou types de données. Les corps de sous-système (liens entre objets composant, points de connexion) sont produits à cette étape.

• *phase 4) abstraction des objets introduits.*
Il s'agit de "remonter" le niveau d'abstraction des objets en vue de rassembler des objets de même nature, généraliser un objet pour une réutilisation future ou rattacher une instance d'objet à un type existant.

• *phase 5) choix du nouveau type d'objets à décrire*
Le cycle est réarmé pour un nouveau type d'objet en commençant par les sous-systèmes. Quand tous les sous-systèmes sont réalisés, le concepteur doit alors choisir parmi les processus et les services non réalisés (en conception préliminaire). Enfin les types de données seront décrits.
En dernier lieu la conception détaillée des entités de deuxième et troisième niveau est réalisée.

2.4.4. Commentaires

MACH2 fournit une bonne approche pour le temps-réel en proposant des concepts (sous-systèmes, point de connexion) non présents dans ADA. Cependant le modèle objet est trop réduit et les concepts "câblés" sont trop nombreux voir redondants. Les types de données sont très voisins des services. La seule différence réside dans le fait qu'un type de donnée est local à une entité (service ou processus) alors qu'un service fait partie de la décomposition d'un sous-système et peut à ce titre participer à la synchronisation indirecte des processus.
L'orientation objet est à notre avis déviée de sa nature en ce sens que les moyens de structuration du premier niveau s'appuient sur le traitement: sous-systèmes et processus. Les véritables objets que sont les services et types de données n'apparaissent qu'aux niveaux inférieurs.
Enfin, bien que la stratégie prévoie une phase d'abstraction permettant de réutiliser et de généraliser les objets, les techniques telles que l'héritage ou la généricité permettant de supporter cette phase d'abstraction font défaut.

2.5. Bilan sur l'approche OOD

2.5.1. OOD et modèle objet

L'acronyme OOD (Object Oriented Design) est devenu synonyme de conception par objets pour une réalisation classique. Le modèle objet est adopté pour la modularité intrinsèque des objets qui permet d'encapsuler dans un même package[1] données et traitements. L'OOD, parce qu'il est destiné à des langages classiques tels qu'ADA, ne prend pas en compte toute la richesse du modèle objet. En particulier les concepts de classe, d'instanciation, de polymorphisme et d'héritage ne sont pas utilisés. De ce fait l'OOD peut être considérée comme une bonne méthode modulaire mais elle n'est pas adaptée à l'implantation dans un langage réellement orienté-objet.
Les méthodes dérivées de l'OOD telles que HOOD ou MACH2 n'offrent pas plus les concepts manquant. De plus ces méthodes n'ont pas su rompre avec le modèle des machines abstraites et en conserve de ce fait les inconvénients: structuration en couches

---

[1] le mot package - paquetage en français - vient du langage ADA. Un package est une unité logicielle contenant des données et des fonctions, munie de points d'entrée (interface) et masquant des informations dans sa partie privée (body).

contraignante, exportation directe de données, absence de concept de modèle (classe). Cependant, les nouvelles notions telles que les entités structurantes et les relations inter-entités montrent qu'il est important au niveau conception (donc architectural) de proposer des moyens pour structurer les objets.

2.5.2. Autres méthodes

Dans la lignée des méthodes issues de l'OOD on peut citer la méthode MAC_ADAM [Riguidel 87]. MAC_ADAM est très voisine de MACH2, elle propose une architecture de l'application en "machinogrammes", ensemble structuré de diagrammes permettant de décrire un objet de décomposition (programme abstrait, processus ou machine).

Le projet de recherche européen ESPRIT "DRAGOON" (Distributed Reusable Ada Generated From an Object Oriented Notation) et sa méthode associée DREAM [Di Maio 89] est prometteur. Il tente de développer une approche tirée de HOOD mais proposant les concepts de classe et d'héritage de classes.

La méthode DOCASE [Mühlhäuser 88] est à l'état de projet et s'adresse aux concepteurs d'applications distribuées.

La méthode MACAO [Bois 89] est une Méthode d'Analyse et de Conception ADA Orienté-objet. Elle propose une conception fonctionnelle descendante suivie d'une conception orienté-objet ascendante pour les données (types abstrait). Ce compromis risque de maintenir les inconvénients de la conception fonctionnelle descendante (cf Chapitre 1).

2.5.3. La nouvelle génération de méthodes

Le modèle objet présent dans les langages de programmation est plus riche que celui proposé par l'OOD. Il convient de disposer de méthodes prenant en compte ces concepts. Certains langages comme EIFFEL ou TRELLIS/OWL se déclarent à finalité "génie logiciel". L'utilisation du même modèle en phase de conception et de programmation permet d'améliorer la qualité. Ces qualités sont d'ailleurs largement dues aux notions du modèle objet ignorées par l'OOD (Classe, Héritage, Polymorphisme). EIFFEL est l'un des langages ayant été le plus loin dans ce sens en proposant l'héritage multiple, la généricité contrainte, les classes virtuelles et les mécanismes d'assertion et d'exception. Cependant, peu de concepts sont fournis pour structurer le logiciel, guider le concepteur dans ses choix et organiser un ensemble de classes dont le volume croît très vite, même pour des programmes de taille raisonnable.
Les méthodes que nous présentons dans la suite appartiennent à une nouvelle génération. Elles supportent toutes un ensemble plus riche de concepts que l'OOD et exploitent les notions issues des langages orienté-objet. C'est aussi cette philosophie que nous avons adoptée pour notre propre méthode.

3. LES NOUVELLES APPROCHES

Les méthodes présentées ici ont toutes en commun leur récente apparition (89-91). Elles se démarquent de l'OOD par leur souci de préserver le modèle objet issu des langages. En particulier, elles proposent l'héritage et indiquent comment les structures de spécialisations peuvent être mises en place[1].

[1] Le lecteur pourra trouver dans [Wirfs-Brock 90a] une présentation des recherches actuelles sur la conception orienté-objet, dont certains travaux non exposés dans ce chapitre.

3.1. Object Modeling Technique

La méthode OMT (Object Modeling Technique) est une méthode ayant fait l'objet d'un livre publié récemment [Rumbaugh 91], qui fait suite à une publication plus ancienne mais partielle [Loomis 87]. Elle se propose de guider le concepteur à travers les trois premières phases du cycle de vie du logiciel, à savoir l'analyse des besoins, la conception d'une solution, et l'implantation de la solution dans un langage de programmation ou dans un SGBD. L'accent est surtout mis sur la phase d'analyse pour laquelle les auteurs de proposent une modélisation par objets en tenant compte de tous les concepts du modèle objet. L'apport essentiel concerne la définition de règles de recherche des classes et des caractéristiques, ainsi que de règles de recherche des relations d'association et d'héritage.

3.1.1. Le modèle objet

La méthode s'appuie sur les 4 thèmes fondamentaux du modèle objet selon les auteurs:

Identité : une donnée est quantifiée dans une entité distinguable appelée Objet.
Exemple: un "paragraphe de document" ou la "reine blanche du jeu d'echec".

Classification : les objets ayant la même structure (attributs) et le même comportement (opérations) sont regroupés en **classes**.
Exemple: PARAGRAPHE, PIECE D'ECHEC.

Polymorphisme : la même opération se comporte différemment selon les classes.
Exemple: *Déplacer* appliqué à FENETRE ou appliqué à PIECE D'ECHEC.
Cela permet d'ajouter de nouvelles classes sans changer le code appelant.

Héritage : partage d'attributs et d'opérations parmi des classes à partir d'une relation hiérarchique. Cela conduit à réduire les répétitions.
Exemple: PION hérite de PIECE D'ECHEC.

3.1.2. Les quatre étapes du cycle de vie OMT

OMT est constituée d'une notation graphique et d'une méthodologie. Le principe de la méthodologie est de construire un modèle du domaine de l'application (Analyse) et de l'enrichir ensuite par des détails d'implantation pendant la conception du système. La notation graphique restant valide pour les deux étapes, la frontière entre analyse et conception s'efface au profit d'un processus de raffinement incrémental: les objets issus de l'analyse sont réalisés pendant la conception, éventuellement au moyen de nouveaux objets.
La méthodologie se décompose en quatre grandes étapes:

1) Analyse
L'analyse part de l'énoncé du problème et construit un modèle du monde réel. Les objets que l'on crée proviennent de l'espace du problème (domaine de l'application). Cette étape fournit un premier niveau appelé modèle d'analyse. Par exemple, une classe Fenêtre sera décrite en terme d'attributs et d'opérations visibles par l'utilisateur (taille, déplacer, position).

2) Conception système
Cette étape n'est pas spécifiquement orienté-objet. Il s'agit de prendre des décisions sur l'architecture du système et de définir des stratégies d'implantation. Le système est organisé en sous-systèmes. Un sous-système est un ensemble cohérent de classes,

d'associations, d'opérations, d'événements et de contraintes interdépendants qui possède une interface réduite avec les autres sous-systèmes. Un sous-système offre un service, ensemble de fonctions partageant un objectif commun: entrées/sorties, graphique, arithmétique. Cette notion est comparable au concept de bloc système de ObjectOry.
L'organisation des sous-systèmes peut être horizontale (niveaux ou couches) ou verticale (partition).
Dans cette étape, le concepteur doit aussi identifier le parallélisme, allouer les sous-systèmes aux processeurs, et définir la gestion du stockage des données.
Enfin, le concepteur doit choisir le style de contrôle: dirigé par les procédures, dirigé par les évènements ou concurrent.

3) Conception objet
Vient ensuite la construction du modèle de conception basé sur le modèle d'analyse. Dans ce modèle sont mis en évidence les structures de données et les algorithmes permettant d'implanter chaque classes du modèle d'analyse. De nouvelles classes spécifiques à ce modèle peuvent être créées.

4) Implantation
Enfin, la traduction est effectuée dans un langage de programmation, une base de donnée ou en forme cablée .

L'avantage de cette approche dans l'utilisation d'un le même modèle tout au long des étapes assurant ainsi la cohérence du développement. Une même classe va évoluer du modèle d'analyse au modèle de conception sans changement de notation. Si, de plus, l'implantation est faite dans un langage ou une BD objet, on assure la validité des quatre thèmes énoncés plus haut de l'étape 1 à l'étape 4.

3.1.3. Les trois modèles de OMT

La modélisation du système à réaliser est conduite sur trois axes complémentaires: axe descriptif, axe dynamique et axe fonctionnel. L'analyste doit construire trois modèles correspondant à ces trois axes:

1) Modèle objet: il supporte l'axe descriptif. Dans ce modèle, les objets et leurs relations sont décrits. Les diagrammes d'objets sont élaborés.
2) Modèle dynamique: ce modèle décrit l'interaction (axe dynamique) entre les objets au moyen de diagrammes d'états/transitions.
3) Modèle fonctionnel: ce modèle décrit les transformations de données au moyen de diagrammes de flots de données, c'est l'axe fonctionnel.

Les auteurs donnent une place prépondérante au premier modèle. Celui-ci assure en effet une décomposition stable et facilement évolutive. Les deux autres modèles sont classiques et proviennent de méthodes déjà éprouvées: ils interviennent à titre de complément du premier modèle.

3.1.4. Le modèle objet

Nous décrivons le modèle objet proposé par OMT qui est la partie la plus intéressante pour notre étude. Le modèle objet utilise différentes notions empruntées à la fois aux modéles entité-relation et à l'approche objet. Du monde orienté-objets sont repris les concepts de classe, objet, attribut, opération et méthode, généralisation et héritage (simple ou multiple). Le modèle entité-relation apporte les notions d'association et de lien, de nom de rôle, d'agrégation, de propagation d'opération et de contrainte.

La Classe est le composant de base des diagrammes de classes (figure 5). Des diagrammes d'instances (objet) peuvent aussi être définis pour exprimer des cas particuliers et des exemples.
Les Attributs de classe possèdent des valeurs qui ne sont pas des objets mais des constantes de base (entier, string ...).
Les Opérations agissent sur les objets pour les transformer. Les paramètres et les résultats de fonctions sont typés par des noms de classe.

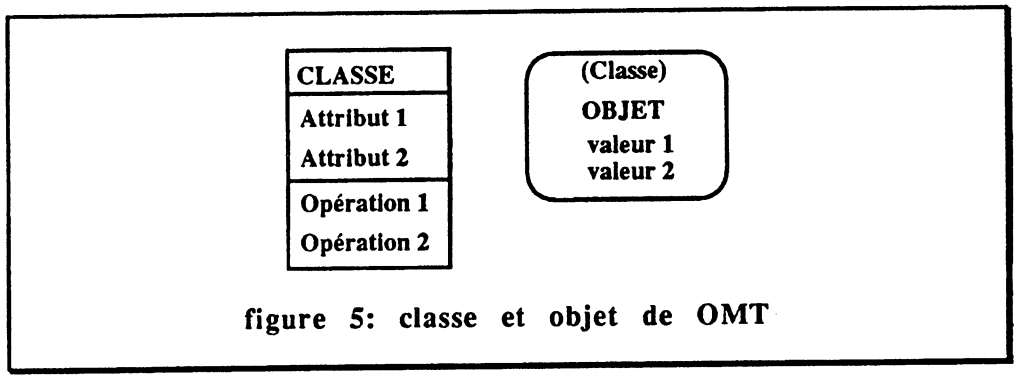


figure 5: classe et objet de OMT

Les Liens sont des connections entre instances d'objet (Tuples).
Une Association est un ensemble de liens (bidirectionnelle).
Les Attributs de liens permettent de positionner des informations directement sur les liens (cf exemple, figure 6)
Un Nom de Rôle permet de préciser, sur une association, les rôles que jouent les classes associées. Exemple:

PERSONNE employé ----- travaille_pour ------ employeur SOCIETE

Les noms de rôles sont surtout utilisés pour distinguer les différents rôles joués par une même classe dans plusieurs association (sinon le nom de la classe suffit).

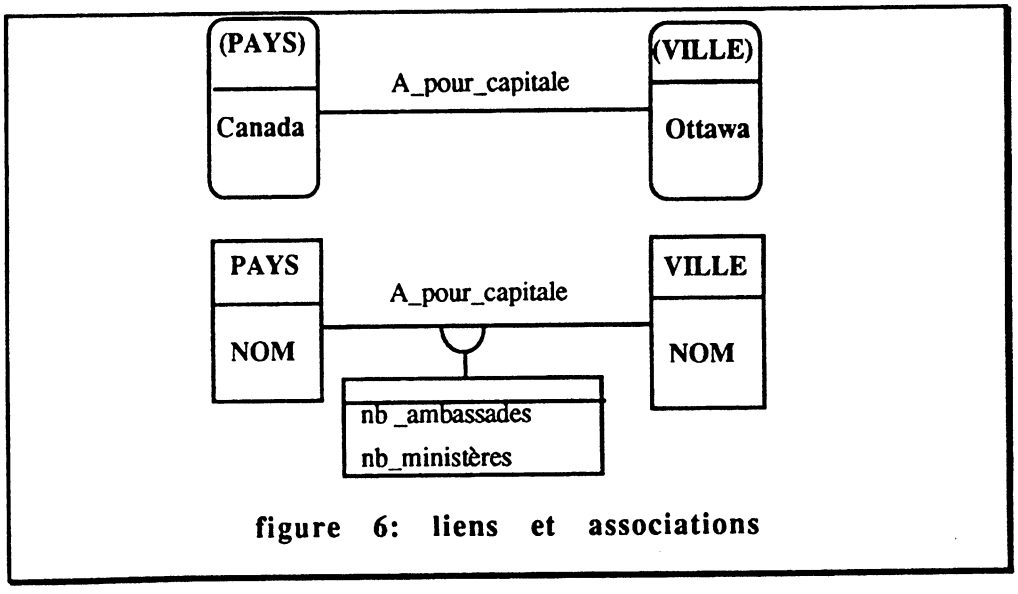


figure 6: liens et associations

L'Agrégation est une forme particulière d'association dans laquelle les classes associées sont couplées étroitement par un lien à forte sémantique. La sémantique de ce lien est de

type "partie de" ou relation "tout-partie". Cette relation est transitive et antisymétrique. Le choix d'une agrégation plutôt qu'une association est laissé à l'appréciation de l'analyste. Dans le doute, les auteurs préconisent l'association.
Des propagations d'opérations et de contraintes peuvent être associées aux agrégations et aux associations. Une opération est propagée entre un objet et ses composants si son exécution sur l'objet déclenche l'exécution de chacune des opérations équivalentes sur les composants.

La Généralisation est une relation entre une classe (A) et une ou plusieurs classes (Bi). Chaque Bi **hérite** de A et constituent des versions plus raffinées de A. La généralisation apparait donc comme une relation orientée et n-aire. Une généralisation peut avoir un discriminant indiquant le critère de classification. Par exemple, la spécialisation de Mammifère en Carnivore et Ruminant est établie sur le discriminant "type de nourriture". Les spécialisations peuvent être disjointes ou non. Une spécialisation disjointe assure que les classes descendantes forment une partition de la classe héritée.
L'héritage multiple n'est possible que sur des classes appartenant à une spécialisation non-disjointe, c'est à dire que certaines spécialisations peuvent se recouvrir. Cette vue de l'héritage est ensembliste: une classe est vue comme un ensemble d'objets potentiels, les spécialisations sont des sous-ensembles de l'ensemble initial. L'aspect opérationnel est plutôt défini dans les modèles dynamique et fonctionnel.
Afin d'utiliser l'héritage à bon escient, les règles suivantes sont proposées:

- *Toutes les opérations de requête sont héritées (fonctions).*
- *Toutes les opérations de mise à jour sont héritées par les extensions.*
- *Les opérations de mise à jour d'attributs contraints ne sont pas autorisées dans la classe fille. Exemple: L'opération homothétie héritée de Ellipse par la classe Cercle est bloquée.*
- *Les opérations réécrites ne doivent pas avoir un comportement différent et doivent avoir le même protocole.*
- *Les opérations héritées peuvent être redéfinies en ajoutant un comportement additionnel.*
- *L'arbre de généralisation ne doit pas excéder 2 ou 3 niveaux.*

Le Module est un ensemble de classes reliées par des associations et/ou des généralisations.

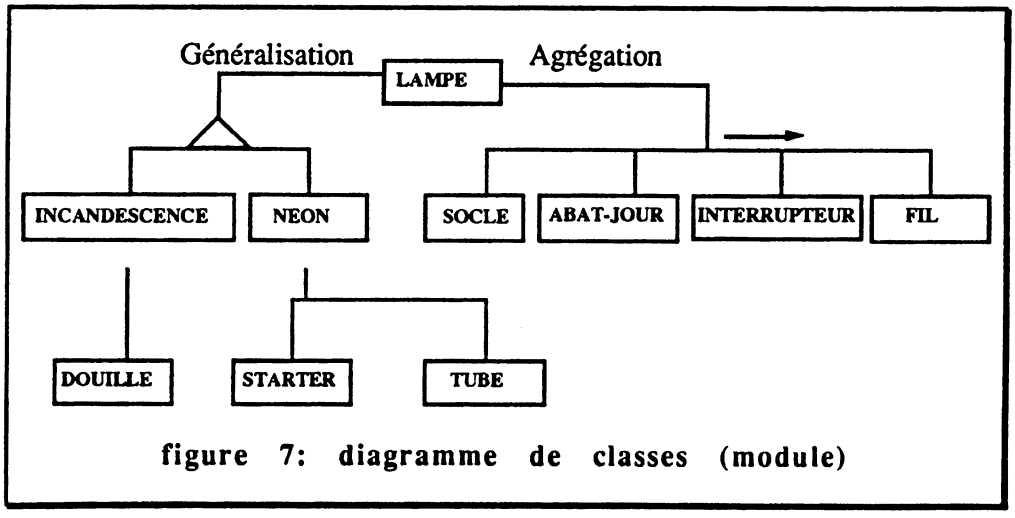


figure 7: diagramme de classes (module)

La figure 7 présente un exemple de module comportant des classes reliées par la relation d'agrégation et la relation de généralisation.

3.1.5. Processus de développement

La méthode OMT propose un processus de développement adapté à l'analyse et la conception. Nous décrivons un résumé de ces processus.

3.1.5.1. Processus d'analyse

1) Identifier les classes d'objets:
Les objets sont recherchés en examinant les noms dans l'énoncé du problème. Il peut s'agir d'entités physiques ou de concepts abstraits.

2) Garder les bonnes classes:
Dans cette étape, il s'agit d'éliminer un certain nombre de classes inutiles. On supprimera les classes redondantes ou ne relevant pas du problème, ainsi que les classes trop vagues. Certaines entités peuvent devenir des attributs d'objet ou des opérations (nom en "tion" ou "ement").
Chaque nom de classe est ensuite répertorié dans un dictionnaire de données avec sa définition.

3) Identifier les associations:
Les associations entre classes sont définies en examinant les verbes d'état ou phrases verbales: près de, contenu dans, travaille pour, conduit, parle à, possède ...

4) Garder les bonnes associations:
Les associations entre classes éliminées à l'étape 2) sont supprimées. De même, les associations hors problème ou non structurelles (désignant des actions) seront éliminées. Les relations ternaires seront binarisées quand cela est possible.

5) Identifier les attributs:
Les attributs sont des propriétés individuelles des objets. On examine les phrases possessives (ex: "la couleur de la voiture") et les adjectifs (ex: rouge, vendu).

6) Garder les bons attributs:
Les attributs ne doivent pas être des objets, sinon une association doit être utilisée. La distinction se fait selon le contexte: par exemple, une ville est un attribut dans une liste d'adresses, elle devient un objet dans un recensement. Les attributs qui qualifient une association ne doivent pas apparaître dans l'objet. Les attributs d'implantation ne doivent pas apparaître dans l'analyse.

7) Raffiner au moyen de l'héritage:
L'héritage est utilisé dans les deux sens: généralisation d'aspect communs entre classes existantes en une super-classe commune ou bien raffinage de classes existantes en sous-classes.
Pour trouver les généralisations, on examinera les classes ayant des attributs, des opérations ou des associations similaires. Les spécialisations peuvent être découvertes dans les substantifs composés d'adjectifs variés. Exemple: lampe fluorescente, lampe incandescente; menu fixe, menu pop-up.

8) Itérer la modélisation:
Cette itération permet d'affiner le modèle. Plusieurs axes sont étudiés:
- asymétries dans les associations et généralisations: ajout de classes par analogie,
- attributs et opérations disparates dans une même classe: découpe en plusieurs classes,

- rôles fortement sémantiques: conversion d'associations en classes,
- classes non connectées: ajout d'associations.

9) Grouper les classes en modules:
Un module est un ensemble de classe qui capture un sous-ensemble logique du modèle. Les modules seront définis de manière à minimiser les relations inter-modules.

Ce processus d'analyse est ensuite suivi d'une modélisation de la dynamique puis d'un modèle fonctionnel. Les trois modèles ainsi obtenus constituent la base de l'étape de conception.

3.1.5.2. Processus de conception objet

La conception consiste à définir complètement les entités du modèle d'analyse (classes, associations, interfaces et algorithmes des méthodes) en suivant la stratégie définie par la conception système (cf 3.1.2.).

1) Combinaison des trois modèles: les trois modèles d'analyse (objet, fonctionnel et dynamique) sont combinés pour obtenir les opérations. Le modèle objet est privilégié et sert de squelette au modèle de conception.

2) Concevoir les algorithmes: choix des algorithmes et des structures de données, définition des classes internes et des opérations.

3) Optimiser: l'optimisation peut être faite sur les accès par ajout d'associations redondantes ou par la sauvegarde d'attributs dérivés pour éviter les calculs.

4) Implanter le contrôle: on utilise le type de contrôle défini dans la conception système pour réaliser le modèle dynamique d'analyse.

5) Ajuster l'héritage: cet ajustement peut être fait par réarrangement des classes et des opérations. Les opérations identiques sont regroupées dans un ancêtre commun. Les comportements peuvent être factorisés pour favoriser une réutilisation potentielle, améliorer l'extensibilité ou pour gérer des configurations différentes du système. Il est conseillé d'utiliser la délégation plutôt que l'héritage pour partager une implantation.

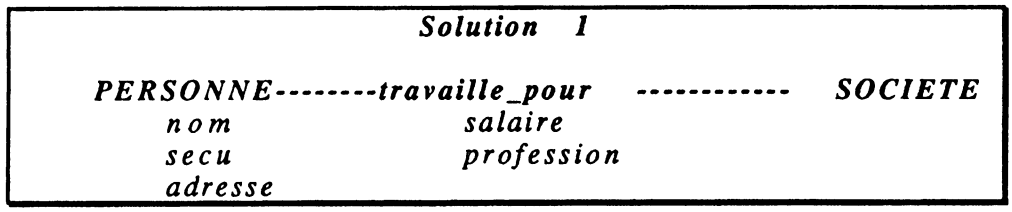
6) Concevoir les associations: en général une association unidirectionnelle sera réalisée par une référence sur un objet. Les associations bidirectionnelles peuvent être réalisées par un attribut dans un sens et une recherche dans l'autre sens, ou bien par un attribut dans chaque sens, ou encore en créant un objet spécifique représentant l'association. Les attributs de lien seront réalisés dans les objets respectifs.

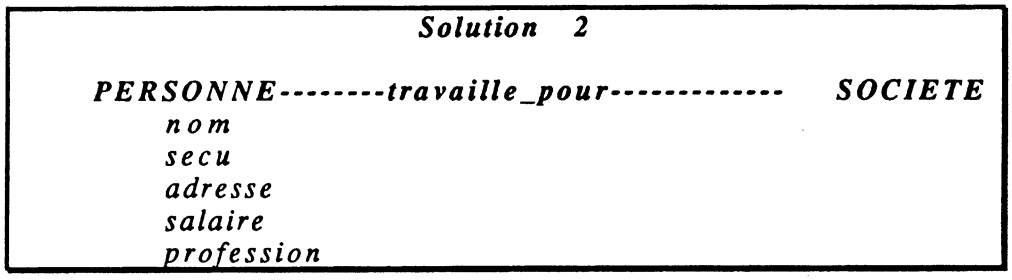
7) Paquetage et documentation: les interfaces des classes sont définies. Des modules, éventuellement différents de ceux de l'analyse, sont constitués afin de repartir l'ensemble des classes. Les décisions de conception doivent être enregistrées et documentées tout au long du processus.

3.1.6. Commentaires

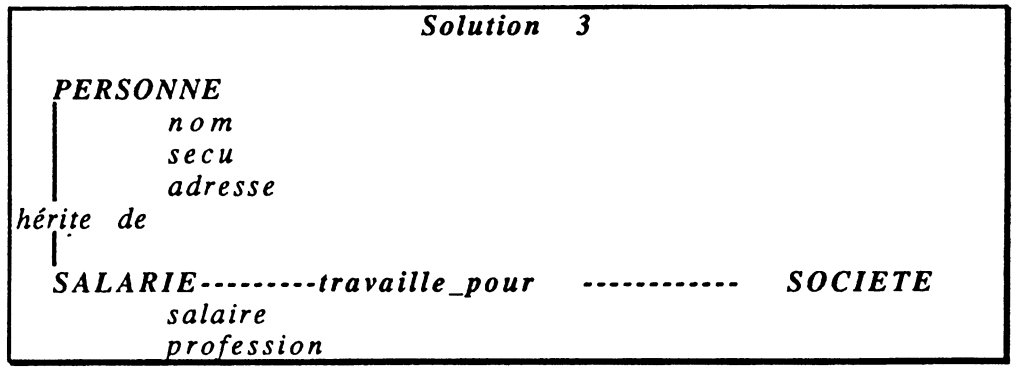
La méthode OMT applique le paradigme objet à tout le cycle de vie du logiciel et propose de ce fait un processus homogène et cohérent pour le développement orienté-objet. Le modèle d'analyse offert est très riche. Cette richesse peut cependant obscurcir le modèle, certains concepts étant redondants. Le concept d'attribut de liens est superflu. Prenons l'exemple d'une personne travaillant pour une société, les auteurs préconisent la

solution 1 avec attributs de lien plutôt que la solution 2 obtenue par dépliage sur la classe personne:





Il existe une troisième solution non proposée, qui consiste à spécialiser PERSONNE en SALARIE qui conduit au schéma suivant:



Cette solution n'utilise que le concept de l'héritage et permet d'appliquer le polymorphisme (un salarié peut être vu comme une personne).
Le concept de nom de rôle que propose la méthode subit la même critique et fait double emploi avec l'héritage. Ceci risque de conduire les analystes à se poser des choix là où il n'y en a pas: faut-il hériter ou mettre une association avec attribut ou nom de rôle ?

Il semble arbitraire de contraindre les attributs à prendre leur valeurs dans des types de base et de ranger tout autre cas dans des situations d'associations. En effet, si une Personne à un attribut *profession* de type *string*, cet attribut ne sera plus raffinable car il faudrait le transformer en association. Ceci va à l'encontre des qualités d'encapsulation et d'évolutivité du modèle. Il est préjudiciable de fixer dès l'analyse l'implantation des attributs dans des types de base.

La règle des 3 niveaux de généralisation semble trop rigide et ne tient pas compte des cas d'héritage multiple. Les auteurs imposent cette règle par soucis de ne pas complexifier le graphe d'héritage des classes (problème de compréhension). Ceci ne se justifie plus s'il l'on dispose d'une taxonomie de l'héritage et d'un outil permettant d'avoir des vues ne montrant que tel ou tel type de relation et masquant les autres.

3.2. Object Oriented Analysis

La méthode OOA ou méthode de Coad-Yourdon [Coad 91] est une méthode d'analyse orienté-objet.
Elle repose sur les concepts d'abstraction, d'encapsulation, d'héritage et d'association. Elle fournit des concepts pour organiser et structurer les objets. Les relations disponibles sont la Généralisation/Spécialisation et la relation Assemblage/Partie. La notion de "sujet" est proposée pour regrouper des ensembles de classes appartenant à la même structure (tout/parties ou héritage). Le processus d'analyse préconisé se décompose en 5 étapes:

1) Trouver les classes et les objets,
2) Identifier les structures (Héritage et Tout/Partie),
3) Identifier les sujets,
4) Définir les attributs,
5) Définir les services (comportement)

Les auteurs préconisent une conception orienté-objet après l'analyse. De la même façon, si un concepteur veux effectuer une conception orienté-objet à partir d'une analyse classique, on lui conseille de réaliser d'abord une analyse orienté-objet avant de commencer la conception.
La méthode OOA n'apporte rien de spécifique par rapport à la méthode OMT dont elle est très proche. Blaha [Rumbaugh 91, p273] considère d'ailleurs que cette méthode est incluse dans la méthode OMT.

3.3. ObjectOry

ObjectOry -"Object Factory for Software Development"- est une méthode d'analyse et de conception créée par Jacobson, préconisant une approche objet pour les gros systèmes [Jacobson 88], [Jacobson 89], [Jacobson 91a]. ObjectOry a été conçue pour répondre aux exigences du développement de logiciel pour les télécommunications et a été généralisée afin de prendre en compte d'autres domaines (gestion, temps réel).
Le modèle objet proposé reprend les concepts d'instance d'objets, de classe d'objets et d'héritage. Les objets communiquent en s'envoyant des "stimuli" (messages) qui déclenchent des opérations (méthodes). Une classe correspond à un regroupement d'objets sur des critères structurels (mêmes attributs et opérations). L'héritage est vu comme un moyen de définir une classe par rapport à une autre, ce qui correspond à une approche plutôt technique.
L'originalité de la méthode tient au fait que le concept d'objet (de classe) a été raffiné en plusieurs sous-concepts correspondant à des types de classe (le mot métaclasse n'est pas employé). ObjectOry propose les types de classe de la figure 8.
La méthode couvre les phases d'analyse, de conception et de test. Bien que l'approche objet soit adoptée dans toutes les phases, des modèles différents doivent être utilisés selon que l'on développe l'analyse ou la conception.

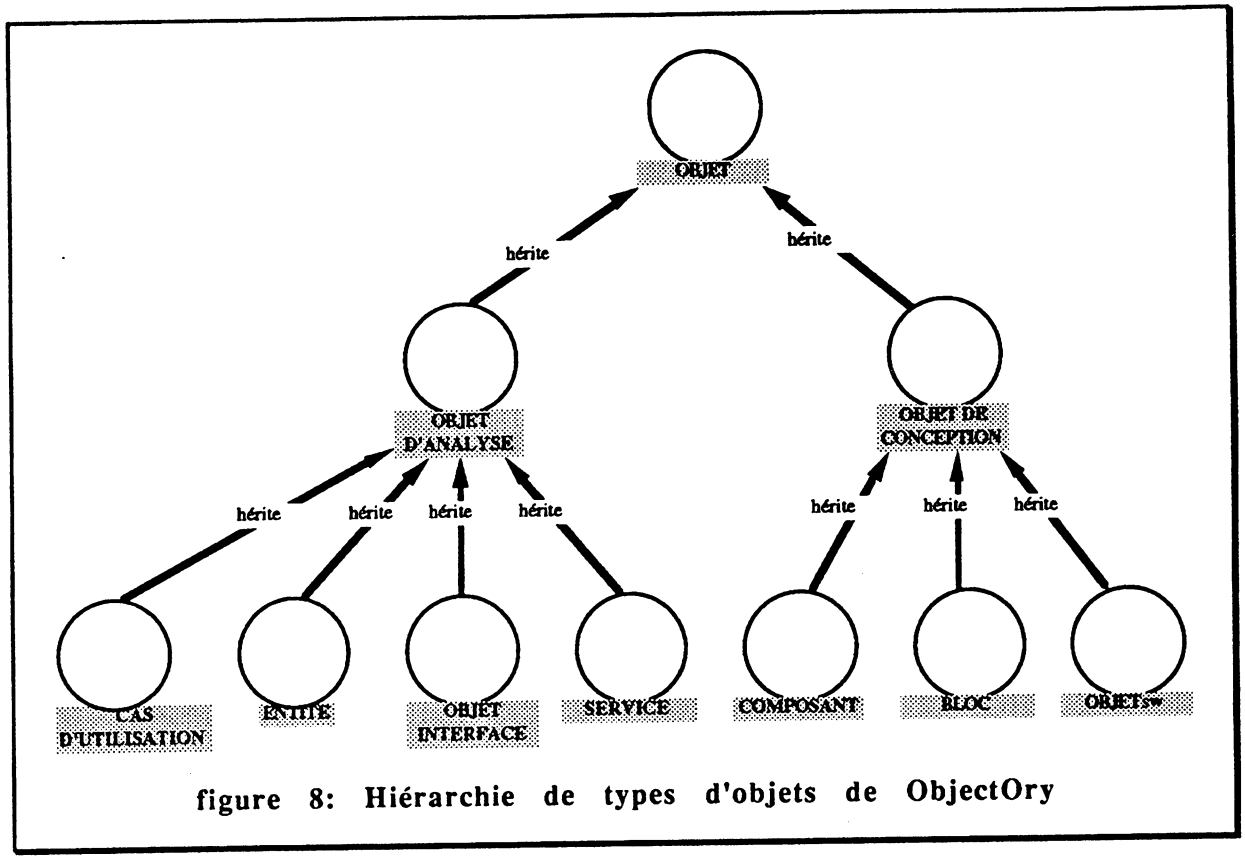


figure 8: Hiérarchie de types d'objets de ObjectOry

3.3.1. L'Analyse

Le modèle d'analyse s'exprime au moyen des objets suivants: Cas d'utilisation, Entités, Objet Interface, Service.

• Les **Cas d'utilisation** aussi nommés "responsabilités du système", modélisent le comportement dynamique du système. Un cas d'utilisation décrit une séquence de transactions pendant laquelle l'utilisateur envoie des stimuli au système et reçoit d'autres stimuli en retour. L'ensemble des cas d'utilisation décrit toutes les façons possibles d'utilisation du système. En prenant en compte de façon prioritaire les responsabilités du système pour les redistribuer sur des unités plus petites, la méthode assure que les spécifications seront bien réalisées. De ce point de vue Objectory est une méthode dirigée par les cas d'utilisation ("use case driven"). Elle est à rapprocher de la méthode "class responsabilities" dirigée par les responsabilités (cf §3.4.).

• Les **Entités** modélisent les informations persistantes du système. Ces informations sont encapsulées et ne peuvent être atteintes que par les opérations. Les diagrammes d'entités permettent d'exprimer les relations entre entités De telles relations sont dites "associations de connaissance". Les associations sont nommées: constitué de, existe dans, (cf Figure 9)

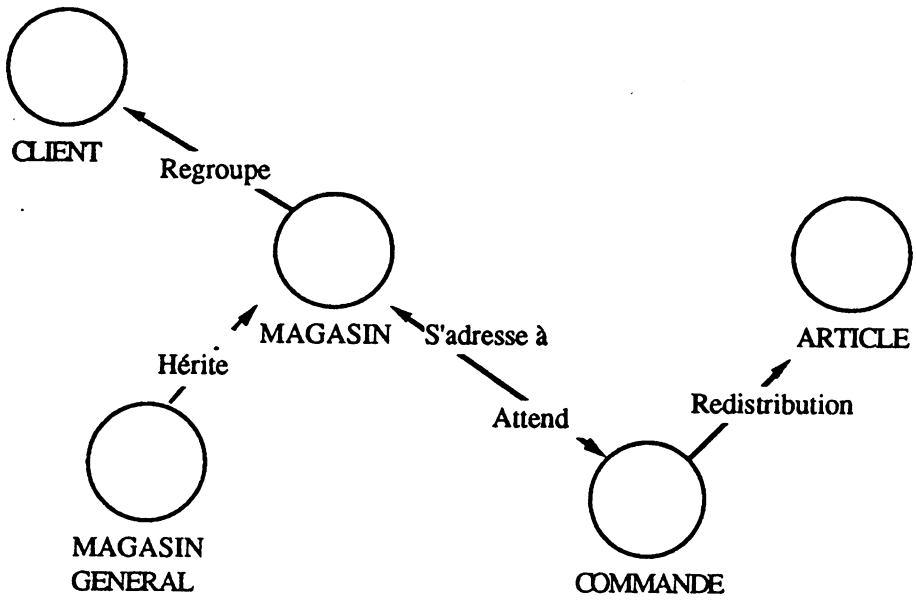


figure 9: Diagramme d'entité

• Les **Objets Interface** représentent les connexions entre l'utilisateur et les cas d'utilisation. L'avantage principal de dissocier l'interface des cas d'utilisation est de pouvoir remplacer l'interface sans changer l'application. Les Objets Interface correspondent à la vue utilisateur du système (Fenêtre, Bouton, etc).

• Les **Services** sont des paquettages contenant des objets fortement connectés des trois types précédents. Le Service offre un ensemble cohérent de fonctionnalités qui peut être commandé par un client désirant un système spécifique. Il peut s'agir de services optionnels, de services interchangeables en vue d'une configuration particulière du système ou bien de services de base obligatoires nécessaires à toute configuration du système. Un système particulier est constitué d'un agencement de services facilement remplaçables et interchangeables pour configurer le système. Les auteurs établissent une comparaison avec l'industrie automobile, dans laquelle les voitures sont conçues de façon modulaire permettant de décliner très facilement des gammes et des options.

L'existence de types d'objets permet d'améliorer la phase de recherche des objets. Cette phase qui est traditionnellement la première étape du processus d'analyse par objets devient plus aisée puisque l'analyste doit rechercher les objets du système pour chacun des types d'objets et dispose ainsi de critères variés pour ce faire.
Le modèle d'analyse est schématisé dans la figure 10. Le modèle se développe sur trois axes complémentaires; Comportement, Présentation et Information. Selon les auteurs de ObjectOry, cette représentation en trois dimensions apporte richesse et souplesse au modèle comparativement à un document textuel linéaire.

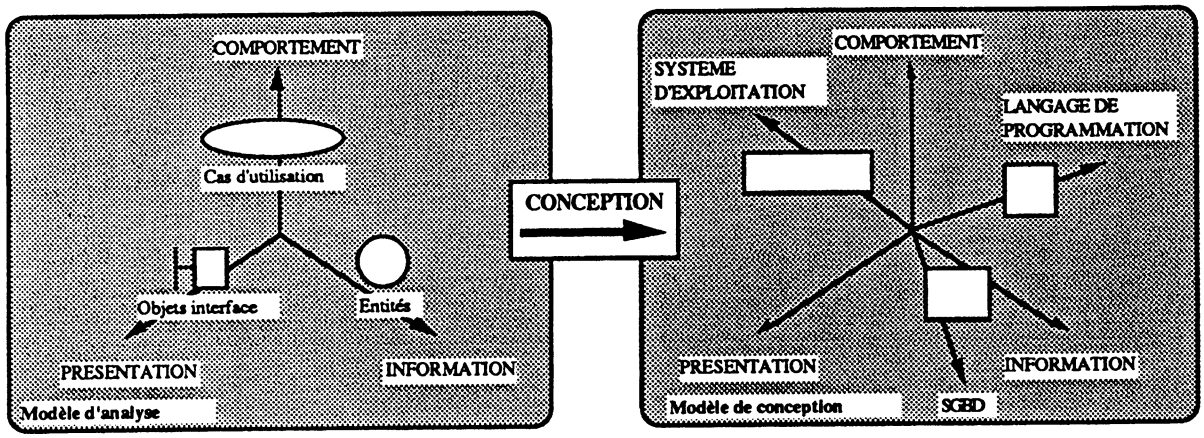


figure 10: Modèles d'Analyse et de Conception

3.3.2. La Conception

Le modèle de conception repose sur des entités appelées "blocs" et "composants" destinés à l'implantation du système. Les composants sont des modules standards réutilisables, indépendants d'une application spécifique. Ils proviennent de librairies (gestion de fichier, mathématiques, systèmes de gestion de fenêtre, gestion de buffers etc...).
Les blocs sont des modules d'application peu réutilisables entre des systèmes de natures différentes, mais fortement réutilisables entre les différentes "instanciations" du même système. Les instanciations d'un système correspondent à des applications dédiées à des clients particuliers, il s'agit de déclinaisons (ou version) du système initial. A une configuration particulière du système correspond un agencement de blocs. Les blocs sont constitués de composants et sont reliés par des "canaux" de communication (envoi de message ou appel procédural).
Les blocs et les composants sont des objets au sens large (cf figure 8). L'instanciation d'un bloc signifie que le code correspondant au bloc du modèle de conception est installé dans le système du client (instance de bloc).

3.3.3. Passage de l'Analyse à la Conception

Selon les auteurs, le passage direct idéal "un concept de l'analyse = un concept dans la réalisation" n'est pas possible du fait des contraintes d'implantation (langage de programmation, SGBD, machine, produit existants). Le modèle d'analyse doit donc être "spécialisé" pour l'environnement courant d'implantation. Pour mettre en place cette spécialisation, les modèles de conception ont une structure hiérarchique construite sur trois types de blocs: les blocs systèmes, les blocs fonctions et les blocs objets. Ces trois sous-types sont hiérarchisés par une relation d'inclusion comme le montre la figure 11.

Un bloc système correspond à une réalisation d'un cas d'utilisation. Les différents blocs systèmes sont rassemblés dans un bloc système supérieur unique définissant les interconnexions (héritage et communication) parmi les sous-blocs.
Un bloc système est constitué de blocs fonctions.
Un bloc fonction réalise un service. Il est lui-même divisé en blocs plus petits: les blocs Objet.

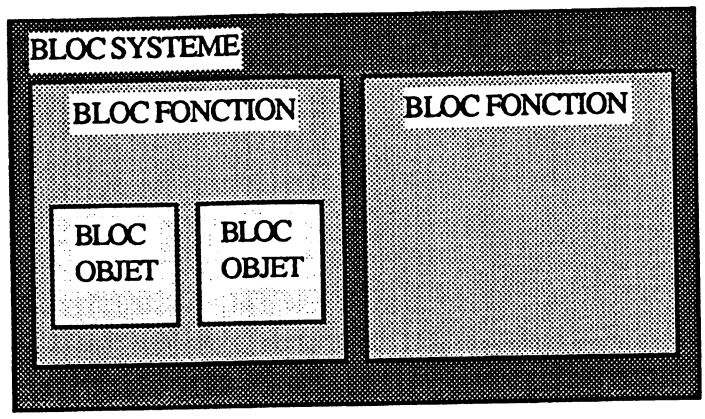


figure 11:
Types de blocs dans le modèle de conception

Chaque bloc objet contient les descriptions d'un certain nombre de classes. Les classes (appelées classe logicielle ou *classSW*) sont les classes du langage de programmation. Dans le langage ADA ce sera un package. Un bloc objet implante une entité, un objet interface ou une partie de cas d'utilisation du modèle d'analyse.
Les blocs communiquent au moyen de "canaux". Un canal est l'extension au niveau des blocs d'un ou plusieurs chemins de communication supportant les stimuli entre objets des différents blocs.

3.3.4. Commentaires

La méthode ObjectOry parait surtout dédiée au très gros logiciels. Elle fournit des concepts de structuration de granularité plus forte que l'objet: services et cas d'utilisation pour l'analyse, système, blocs et composants pour la conception. Elle privilégie la prise en compte prioritaire des cas d'utilisation afin d'apporter la complétude et d'assurer la réalisation d'un système correspondant aux exigences du client. Cependant on risque, comme dans toutes les approches favorisant les fonctions, d'établir des choix importants trop tôt, de défavoriser la généralité (être trop spécifique) et d'être incapable de réutiliser autre que chose que des objets de niveau bas tels que les structures de données ou les interfaces graphiques. La taxonomie établie sur les objets est intéressante pour l'analyse et fournit des critères de recherche des objets.
L'héritage est peu utilisé et le polymorphisme n'est pas mentionné. La seul apport de l'héritage semble être la possibilité de définition incrémentale des objets.
Enfin, si les concepts sont nombreux, leur sémantique est relativement vague[1]. La partie Analyse semble bien traitée dans une "philosophie objet" mais la partie Conception est moins fidèle aux concepts du modèle objet. Ceci est du en grande partie au fait que les auteurs ne font pas d'hypothèses sur l'environnement d'implantation qui peut être aussi bien un langage à objets, un langage classique ou un SGBD relationnel.

3.4. La méthode "Class responsabilities"

La METHODE "class responsabilities" de Wirfs-Brock, Wilkerson et Wiener fais partie de la nouvelle génération de méthode [Wirfs-Brock 90b]. Elle repose sur 6 grands thèmes: CLASSE, RESPONSABILITE, COLLABORATION, HIERARCHIE, SOUS-SYSTEME et PROTOCOLES.

---

1 Précisons que peu de documentation existe sur cette méthode et que le livre à paraître prochaînement permettra sans doute de combler cette lacune [Jacobson 91b].

L'accent est mis sur les "collaborations" (délégation) entre classes permettant d'assurer les responsabilités (service exporté par la classe). Des guides sont proposés pour trouver les responsabilités et les collaborations.

3.4.1. Modèle objet et concepts de base

Les concepts de base de la méthode sont les suivants:

Encapsulation: collection d'informations et de connaissances regroupées et vues comme une seule entité,

Masquage d'Information: distinction des informations précédentes en informations publiques et informations privées.

Message: seule voie de communication entre objets, un message est composé d'un nom, de paramètres et d'un destinataire.

Méthode: algorithme exécuté par un objet en réponse à un message. Une méthode appartient à la partie privée de l'objet.

Signature de méthode: nom de la méthode, type des paramètres et type du résultat.

Classe: ensemble d'objets possédant le même comportement. La classe est vue comme une spécification générique d'objets.

Polymorphisme: mécanisme d'abstraction permettant à plusieurs classes de répondre au même message (signature). Remarque : le polymorphisme n'est pas associé à l'héritage.

Héritage: autre mécanisme d'abstraction donnant la possibilité à une classe de définir le comportement et les structures de données de ses instances comme un sous-ensemble de la définition d'une ou plusieurs autres classes (héritage multiple).

Classe Abstraite: classe ne produisant pas d'instances, utilisée pour factoriser le comportement de plusieurs classes.

3.4.2. Processus de Conception orienté-objet

La conception est vue comme un processus par lequel les besoins d'un système sont transformés en spécifications détaillées d'objets. Le processus de conception est le suivant:

a) Exploration initiale
On utilise pour cette étape des "fiches" de classe (figure 12).

1)	*CLASSES: trouver les classes,*
2)	*RESPONSABILITE: déterminer quelles opérations chaque classe doit proposer (est responsable) et les informations à retenir,*
3)	*COLLABORATION: déterminer comment les objets collaborent pour assurer leurs responsabilités.*

Les notions de client et serveur sont définies. Le client émet une requête, le serveur reçoit la requête et assure le service.
Un contrat est un ensemble de responsabilités du serveur offertes au client (signatures).

b) Analyse détaillée

HIERARCHIES:	*1) examiner les relations d'héritage* *2) examiner les collaborations*

c) Sous-systèmes de classes

Regrouper les classes qui travaillent plus particulièrement ensembles en sous-systèmes. Un sous-système est un ensemble de classes offrant des responsabilités aux autres sous-systèmes.

CLASS: nom de classe	**abstraite ou concrete**
liste des superclasses	
liste des sousclasses	
Responsabilité	**Collaboration**

figure 12: fiche de classe

Nous détaillons les différentes étapes du processus de conception:

1) LES CLASSES (exploration initiale)

Trouver les classes

On s'intéresse aux objets physiques et aux entités conceptuelles formant une abstraction cohérente. On doit modéliser les catégories de classes, les interfaces connues de l'extérieur (utilisateur, programmes, environnements). Les types d'attributs seront définis comme des classes. Enfin les classes sont enregistrées dans des "fiches de classe" (figure 12)

Trouver les classes abstraites

On réexaminer la liste des classes pour trouver le plus possible de superclasses abstraites.

On crée des catégories (hiérarchie d'héritage). Les superclasses sont enregistrées sur des fiches de classes. Pour nommer un groupe de classe on pourra dériver le nom des attributs partagés ou utiliser des abstractions intermédiaires.

2) LES RESPONSABILITES

La responsabilité correspond à la connaissance maintenue dans l'objet et aux actions qu'il peut accomplir. La responsabilité d'un objet comprend tous les services qu'il

fournit pour tous les contrats. Pour déterminer les responsabilités, le processus suivant est proposé:

Identifier les Responsabilités
On recherche les verbes dans les spécifications ainsi que les informations maintenues et/ou manipulées par le système. On utilisera de même les classes trouvées à l'étape précédente dans lesquelles un certain nombre de responsabilités ont déjà été identifiées.

Assigner les responsabilités
Il s'agit de distribuer le plus possible l'intelligence (le contrôle) parmi les classes. On rendra les responsabilités très générales. Si un objet maintient une information, il possède dès opérations pour manipuler cette information. Une information sur un objet doit être dans un seul endroit. On peut partager une responsabilité quand celle-ci est manifestement composée.

Examiner les relations entre les classes
Cet examen utilise trois relations différentes "est une sorte de", "est analogue à" et "est une partie de"
- "est une sorte de" suppose une relation d'héritage: on positionnera les responsabilités dans la hiérarchie,
- "est analogue à": les deux classes possèdent des responsabilités en commun; il faut créér une superclasse commune et lui associer la responsabilité factorisée.
- "est une partie de": on recherche les responsabilités des parties en fonction de celles du tout.

Classes manquantes
Les responsabilités non assignées conduiront à créer de nouvelles classes pour les supporter.

Enregistrer les responsabilités
Sur la fiche de classe, on indique une responsabilité par ligne, dans la colonne de gauche (phrase en langue naturelle) (voir figure 13).

CLASS: TORTUEABSOLUE	abstraite
Tortue	
TortueEcran, TortueRobot	
Sait avancer et reculer d'un nombre de pas	
Sait tourner a gauche et a droite	
Sait rentrer a la maison	POSITION, CAP
Ecrit ou non sur l'ecran en se deplacant	PLUME

figure 13: exemple de fiche de classe

3) COLLABORATION

Les collaborations représentent des requêtes d'un client vers un serveur. Une collaboration supporte un flot de contrôle et d'information entre deux objets.
Un objet collabore avec un ou plusieurs autres objets si pour assumer une responsabilité il a besoin d'envoyer des messages aux autres objets.
L'identification des collaborations peut mettre en évidence des manques dans les responsabilités.

Trouver les Collaborations

• Pour chaque responsabilité d'une classe (client), les questions suivantes doivent être posées:

Est-ce que la classe est capable de remplir sa responsabilité seule ?
Sinon, de quoi a-t-elle besoin?
De quelle classe peut-elle obtenir ce dont elle a besoin?

Toutes les classes partageant une même responsabilité sont en collaboration.

• Pour chaque classe (serveur), les questions suivantes seront posées:

Que fait ou que sait la classe?
Quelle autre classe a besoin d'un resultat ou d'une information détenu par la classe?
Collabore-t-elle avec toutes les classes qui ont besoin d'elle?

Si elle n'a pas d'interaction, éliminer la classe.

• Examiner les relations "est une partie de" ("est composé de" et "container")

On crée des collaborations du "tout" vers ses parties.
On ne créera pas forcément de collaboration entre un container et ses éléments. Dans le cas d'un tableau par exemple, il n'y a pas de responsabilité pour maintenir une information sur ses éléments. Par contre une table de hashcode doit s'assurer que ses éléments sont "hashables". Remarque: ces notions sont à rapprocher de la généricité contrainte, non proposée par la méthode.

• Examiner les relations "à la connaissance de"

Les collaborations entre le demandeur et le fournisseur d'information sont mise en place.

• Examiner les relations "Dépend de" débouchant sur une collaboration de type précédent ou sur une collaboration tri-partite.

Enregistrer les collaborations

Il s'agit d'inscrire le nom de la classe serveur à droite de la responsabilité concernée (figures 12 et 13). Des collaborations peuvent être positionnées dans la superclasse: elles sont automatiquement héritées. On testera ensuite les collaborations avec des scénari d'entrées.

4) LES HIERARCHIES (analyse détaillée)

Graphes de hiérarchie d'héritage (figure 14)

Les classes sont représentées par des rectangles. Les liens d'héritage sont symbolisés par des segments. La direction de l'héritage est donnée par la position des classes: la classe du bas hérite de la classe du haut.
Les classes abstraites sont repérées par un coin supérieur gauche noirci. Le choix d'une classe concrète ou abstraite est établi en réponse à la question: est-ce qu'il existera une instance à l'exécution? si oui, c'est une classe concrète.

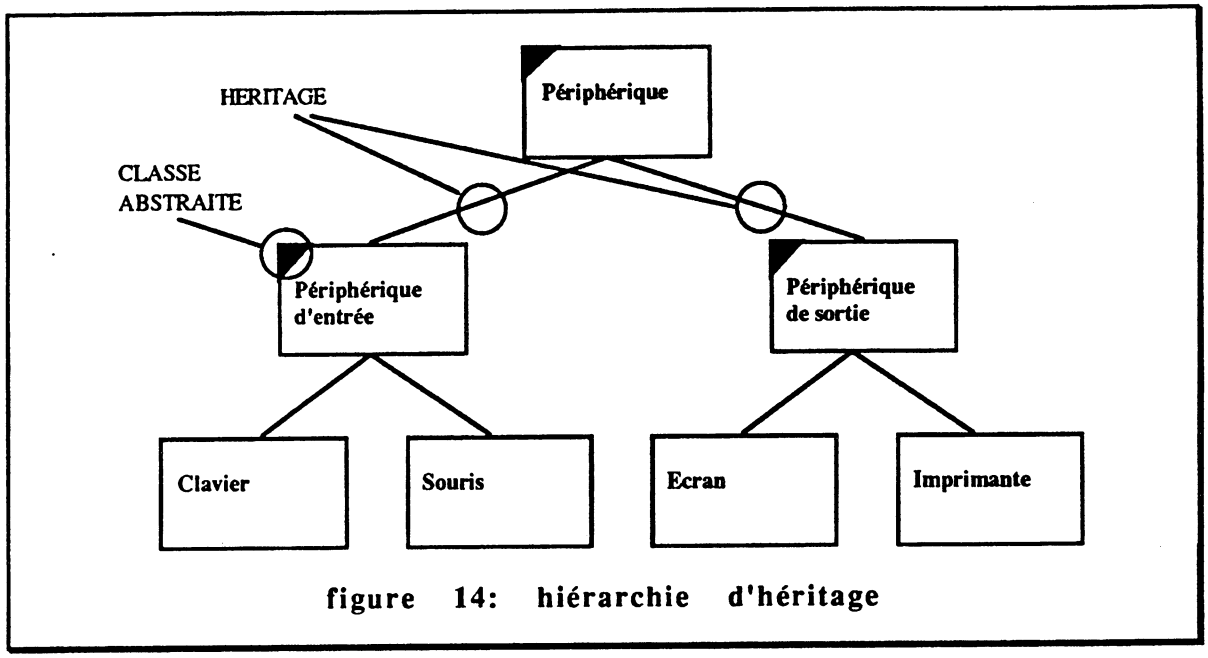


figure 14: hiérarchie d'héritage

Diagrammes de Venn (figure 15):
Un autre moyen de représenter les hiérarchies d'héritage sont les diagrammes de Venn. Une classe est vue comme un ensemble de responsabilités. Les intersections de ces ensembles sont des superclasses, le reste d'un ensemble n'appartenant pas à l'intersection constitue la partie spécifique à la sous-classe. Dans cette approche, les inclusions suivantes sont valides: Array Matrice, Array Chaine, Ordre Chaine.

Selon les auteurs, cette notation est complémentaire aux graphes d'héritage et permet d'avoir une vue d'ensemble des classes.

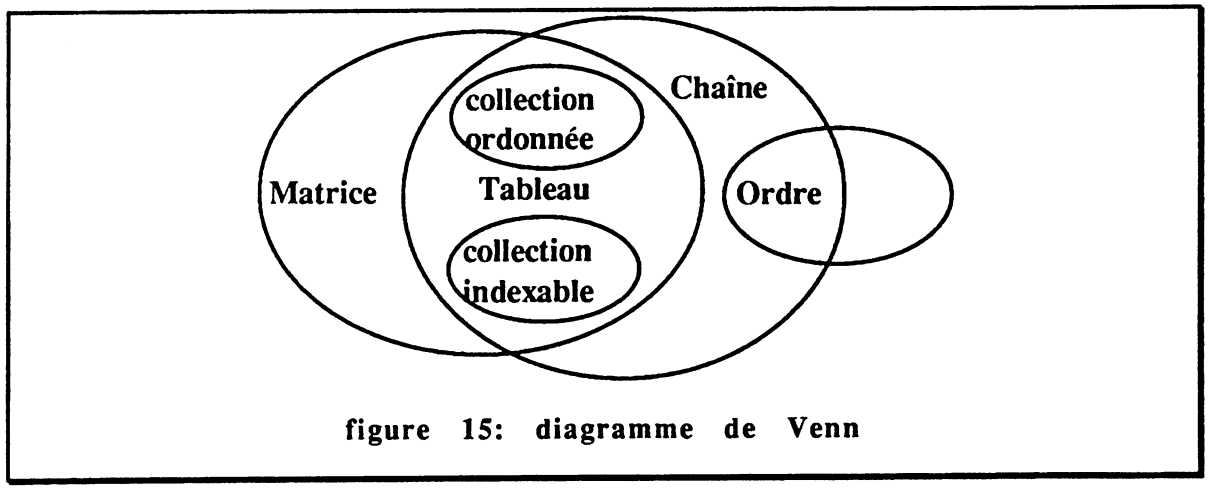


figure 15: diagramme de Venn

Construire des "bonnes" hiérarchies
On modélise la relation "est une sorte de": une sous-classe doit assurer toutes les responsabilités définies dans sa super-classe et si possible plus. Cette approche correspond à une spécialisation. Dans le cas de classes non conformes (la sous-classe n'assure qu'une partie de la super-classe), on restructurera la conception en faisant hériter les deux classes d'une même super-classe abstraite les factorisant.

On cherche à factoriser les responsabilités communes le plus haut possible. Pour cela, on se donnera un grand nombre de classes abstraites intermédiaires. Les classes abstraites ne doivent pas hériter de classes concrètes.
On élimine les classes abstraites qui n'ajoutent pas de fonctionnalités. Par contre, les classes concrètes qui n'ajoutent pas de fonctionnalités mais qui réalisent celles d'une classe abstraite seront maintenues.

Identifier les contrats
Un contrat est un regroupement logique de responsabilités. Il constitue un ensemble de requêtes qu'un client peut demander sur un serveur (ensemble de responsabilités). Le contrat fournit un moyen d'abstraction supplémentaire sur les responsabilités.
Exemple: [addition, soustraction, multiplication, division] est un contrat arithmétique. Si une classe offre uniquement l'addition elle ne respecte pas le contrat arithmétique.
Le critère de regroupement peut être les responsabilités utilisées par le(s) même(s) client(s).
On cherchera à maximiser la cohésion d'une classe: moins on a de contrats, plus on peut construire de nouvelles sous-classes. De même, on minimisera le nombre de contrats (idéal: un seul par niveau de la hiérarchie).

5) LES SOUS-SYSTEMES (analyse détaillée)

La création de sous-systèmes a pour objectif de diviser les responsabilités parmi des groupes de classes. Les sous-systèmes sont définis au moyen de graphes de collaboration et de fiches de sous-systèmes.

Graphes de collaboration
Il s'agit de représentations graphiques des collaborations parmi des classes et des sous-systèmes.
Les graphes de collaboration sont constitués de classes, de contrats, de collaborations et de relations super-classe/sous-classe
Un contrat est représenté par un demi-cercle à l'intérieur du carré de la classe serveur et les collaborations par une flèche dans le sens client -> serveur (voir figure 16).

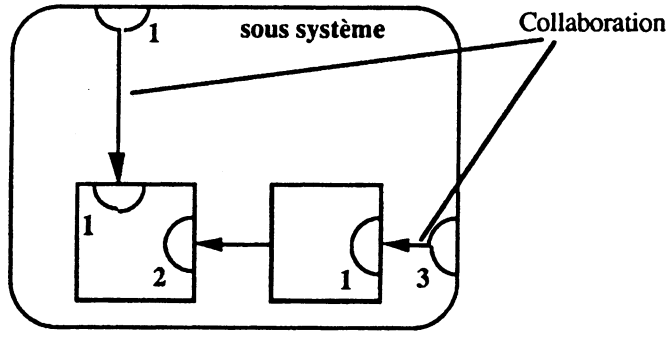


figure 16: sous-système et collaboration

Les Sous-sytèmes
Un sous-système contient un groupe de classes et de sous-systèmes qui collaborent pour offrir un ensemble de contrats. Il doit correspondre à une abstraction cohérente.
Un sous-système supporte un contrat représentant l'ensemble des contrats de toutes les classes qui fournissent quelque chose à l'extérieur.

On établit une fiche par sous-système, similaire aux fiches de classe sans l'héritage. Les fiches des classes clientes sont modifiées pour remplacer la classe serveur par le nom du sous-système auquel elle appartient (la classe serveur est interne au sous-système).

Identifier les sous-systèmes
A postériori: on établit les graphes de collaborations puis on découpe en sous-systèmes en recherchant les classes couplées fortement qui seront dans le même sous-système.
A priori: on construit des sous-systèmes avant de définir les classes contenues.

Simplifier les abstractions
Minimiser le nombre de collaborations qu'une classe a sur les autres classes du sous-système. Minimiser le nombre de classes et sous-systèmes auxquels est relié un sous-système. Minimiser le nombre de contrats différents supportés par une classe ou un sous-système.

6) LES PROTOCOLES (analyse détaillée)
Le but des protocoles est d'assurer que les responsabilités sont bien raffinées et les messages nommés.

Raffiner les responsabilités
Construire les protocoles pour chaque classe (signature des méthodes),
Rendre les protocoles le plus général possible (ajouter des méthodes),
Définir des valeurs et des situations par défaut.

Ecrire une spécification de conception pour chaque classe
On définit:
- le nom et le type (abstrait ou concret),
- les superclasses et sousclasses,
- les références aux graphes de collaboration et d'héritage (n° de page),
- la description (rôle),
- les contrats (ensemble de responsabilités: signature, collaboration, description),
- les responsabilités privées,
- les notes d'implantation.

Ecrire une spécification de conception pour chaque sous-système
On définit:
- le nom, les classes et les sous-systèmes encapsulés
- la référence dans le graphe de collaboration
- le Sujet du sous-système
- les contrats vus par le serveur
- pour chaque contrat, on définit les classe(s) déléguée(s)

Ecrire une spécification de conception pour chaque contrat

3.4.3. Commentaires:

La méthode "class responsabilities" privilégie les responsabilités et les collaborations inter-classes. C'est une véritable méthode de conception par objets qui a le mérite de pouvoir être facilement mise en oeuvre (fiche de classe "papier"). La notion de sous-système apporte un moyen de structuration des classes. La méthode apporte une réponse concrète aux aspects opératoires d'une conception tout en restant dans un univers de modélisation assez abstrait.

On peut cependant lui reprocher d'être trop informelle et de ne pas formaliser suffisamment les concepts qu'elle utilise, en particulier l'héritage. Un des exemples

présenté par les auteurs est particulièrement confus: les classes Matrice et String sont vues comme des sous-classes de Tableau; String et Date héritent tous deux de la classe Ordre (définissant une relation d'ordre abstraite). Dans le premier héritage Matrice constitue une spécialisation de Tableau mais ce n'est pas le cas de String (héritage d'implantation). Dans le second, la relation super-classe / sous-classe correspond à un héritage de propriété (cf Chapitre 3)
De même, les diagrammes de Venn sont générateurs de confusion: la super-classe apparait comme une intersection de sous-classes, ce qui donne l'impression qu'elle dépend des sous-classes. Enfin la règle énonçant que les spécialisations doivent offrir plus de responsabilités parait trop contraignante, notamment dans le cas ou seule des contraintes sont raffinées (Carré héritant de Rectangle par exemple).
L'absence de généricité est dommageable, d'autant plus que l'on s'en approche beaucoup avec les relations "container" et l'héritage de contrat (généricité contrainte).

3.5. La méthode Object Oriented Analysis and Design de Nerson

3.5.1. Présentation

Nous présentons dans cette partie une méthode très proche de nos préoccupations. Il s'agit d'une extension d'EIFFEL vers l'analyse et la conception. J.M. Nerson propose en effet une notation et une méthodologie tirées de son expérience du langage EIFFEL [Nerson 91]. Cette méthode a pour objectif d'apporter une représentation du logiciel plus adaptée à l'analyse et à la conception que les représentations textuelles destinées aux compilateurs. L'accent est mis sur l'uniformité du modèle objet dans les différentes phases du développement.

Le langage EIFFEL peut être considéré comme un formalisme adapté à la conception. En plus des concepts bien connus des langages à objets tels que le polymorphisme, la liaison dynamique, l'héritage multiple, l'envoi de message, les structures de classes et les instances d'objet, il possède des caractéristiques pour la conception: généricité, assertions, invariants de classe, indexation de classe, routines et classes retardées. L'expérience du langage a conduit l'auteur à dégager les principes de conception suivants:

PRINCIPES DE CONCEPTION

- Ecrire des classes "différées"[1] vues comme des types abstraits de données, en utilisant les invariants de classe,
- Ne s'intéresser qu'aux méthodes de l'interface en les spécifiant par les pré- et post-conditions,
- Prévoir les structures candidates au polymorphisme,
- Utiliser la généricité,
- Utiliser des classes de "haut niveau" pour assurer une meilleure flexibilité.

3.5.2. Formalisme

La méthode a deux représentations: un modèle statique et un modèle dynamique. Le modèle statique est produit en premier et représente les propriétés statiques. Le modèle dynamique représente les envois de messages et les instances d'objets.

[1] Les classes "deferred" d'Eiffel sont des classes virtuelles. Ce terme est traduit par "differée" dans la version française de l'ouvrage de Meyer. Nous préférons le terme "deferé" qui traduit bien un report de compétence (réalisation) aux classes héritières.

3.5.3. Modèle statique

Deux niveaux de description sont offerts: le niveau Classe et le niveau Cluster. Le niveau Classe offre des descriptions concises de classes (services proposés et propriétés internes). Le Cluster décrit des parties de l'architecture générale.

Les classes sont représentées par des ellipses contenant le nom de la classe. Différentes conventions graphiques permettent de représenter des informations supplémentaires sur les classes: classe racine de structure, générique, retardée, réutilisée ou classe dont les instances sont persistantes (cf figure 17).

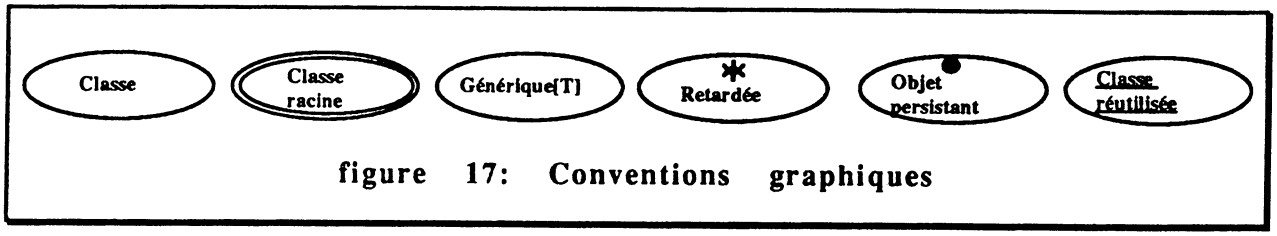


figure 17: Conventions graphiques

Un corps de classe est représenté dans une boite à coins arrondis (cf figure 18). La boite contient la partie indexation (informations optionnelles destinées au fouineur), les caractéristiques publiques disponibles aux autres classes, les caractéristiques à accès restreint (classes autorisées à accéder aux caractéristiques), les caractéristiques privées et l'invariant de classe.

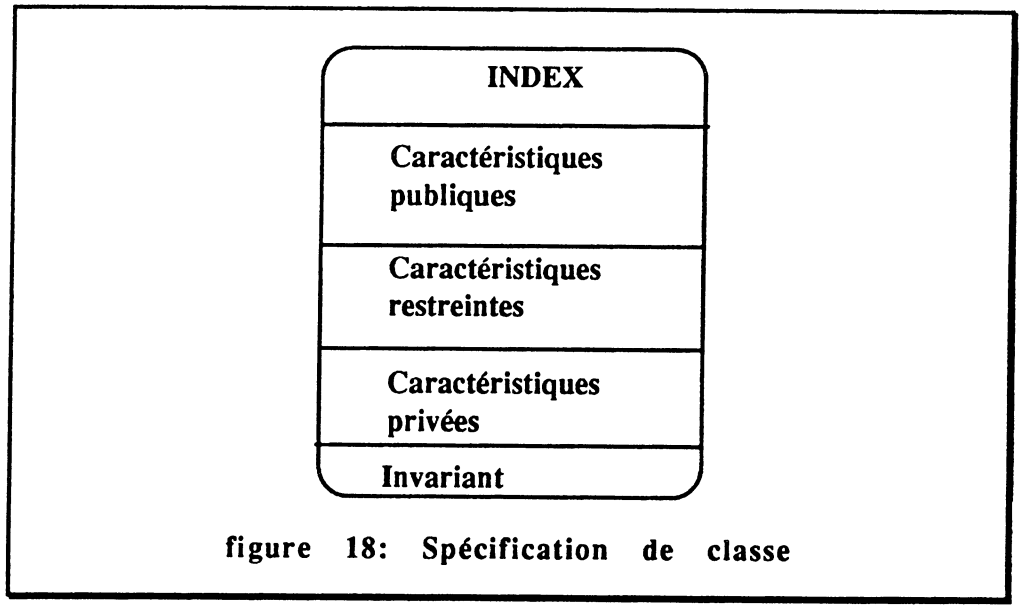


figure 18: Spécification de classe

Les relations entre classes sont de deux sortes: héritage et client-fournisseur.
L'héritage est représenté par une ligne simple dotée d'une flèche. Dans le cas d'héritage direct répété (une classe hérite directement plusieurs fois d'une autre), un losange contenant le nombre de liens d'héritage est ajouté sur la flèche.
La relation client-fournisseur est symbolisée par une flèche à double-trait. Une étiquette rappelant le nom de la caractéristique supportant la relation est positionnée sur la flèche. Une symbologie particulière permet d'exprimer la cardinalité de la relation. La cardinalité n-m indiquant que n instances du client peuvent référer à m instances du fournisseur est assurée par défaut. Le concepteur peut exprimer une relation 1-1, dans ce cas le fournisseur est créé automatiquement à la création du client.

La notation permet de plus d'exprimer une relation de partage (m-1) ou une situation de répartition des objets sur processeurs différents.
La méthode propose aussi une représentation de la généricité contrainte et de l'instanciation de paramètre générique[1].
Les clusters[2] permettent de regrouper des collections de classes répondant à un critère particulier: sous-système de fonctionalités, classes réutilisées, structures de données partagées (figure 19).

Les clusters peuvent avoir des relations avec d'autres clusters ou classes. Ces relations sont les mêmes que celles établies entre les classes. Un certains nombre de règles additionnelles sont énoncées:

- Toute classe apparaissant dans un cluster doit être connectée par une relation parent ou client-fournisseur à une classe du même cluster ou d'un autre cluster.
- Une classe peut être cliente d'un cluster.
- Un cluster peut être client d'un autre cluster.
- Si une classe hérite d'un cluster, alors c'est une classe descendante de toutes les classes de ce cluster.
- Un cluster héritant d'un autre cluster signifie que toutes les classes du premier héritent de toutes les classes du second.
- Une relation client-fournisseur impliquant un cluster indique qu'au moins une classe du cluster fait partie de la relation.
- Enfin, une classe ne doit apparaitre qu'une seule fois dans un modèle statique.

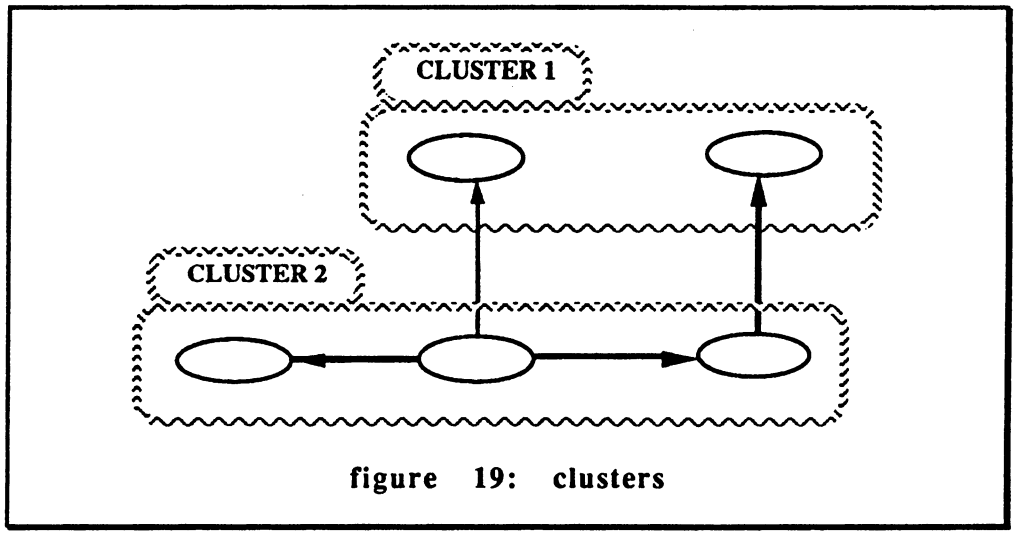


figure 19: clusters

3.5.4. Modèle dynamique

Le modèle dynamique permet d'exprimer les instances de classes et leur couplage. Il possède sa propre notation. Le mécanisme de couplage des objets n'est pas explicité lors de l'implantation, il pourra être traduit en références d'attributs, résultats de fonction, argument d'appel de routine ou accès à un objet partagé.

---

[1] On notera que ceci va dans le sens de ce qui a été proposé par MECANO (voir [Girod90b] et Chap 3).

[2] Littéralement: rassemblement, agglomérat.

Un objet est représenté dans un rectangle avec le nom de la classe d'origine inscrit en haut. Les attributs avec leurs valeurs sont ensuite définis. Les valeurs sont soit des valeurs de type de base (1,"toto", vrai...), soit des références à d'autres objets (->) (cf figure 20).

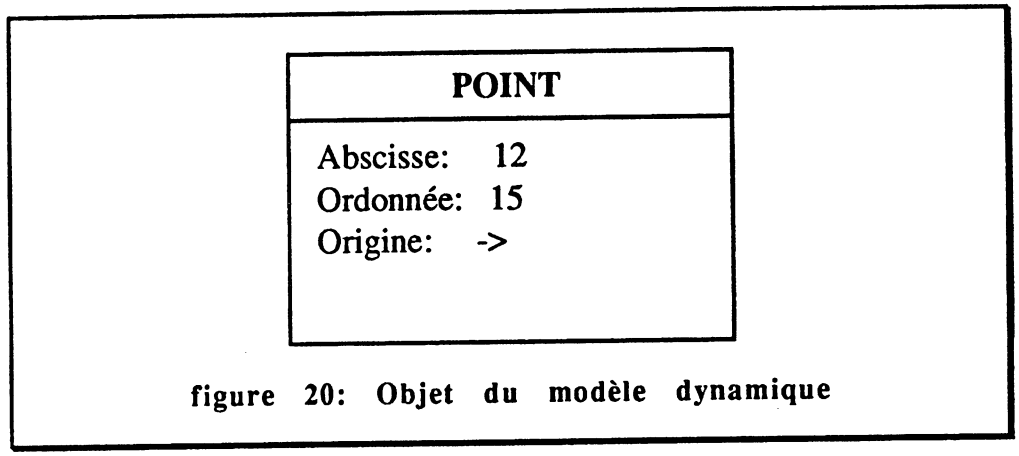


figure 20: Objet du modèle dynamique

Le couplage entre objets est représenté par une flèche grisée, éventuellement étiquetée d'un numéro (utilisé dans la documentation). Une telle flèche indique une voie de communication supportant des envois de messages entre les deux objets.

3.5.5. Processus de conception

La méthode fournit un cadre de conception en 8 étapes. L'auteur précise que ce processus doit être considéré comme un guide plutôt que comme un cadre rigoureux. Les étapes proposées sont les suivantes:

1) Trouver les classes du monde réel:
Les classes sont recherchées dans le domaine d'application. Pour chacune d'elle, on énoncera en plus du nom, les spécifications qu'elles remplissent et les caractéristiques qu'elles possèdent.

2) Définir les Clusters et élaborer les Contrats
Les clusters doivent être définis pour assurer des niveaux d'abstraction ou isoler les parties indépendantes et/ou critiques. De plus, les contrats doivent être établis au moyen des pré- et post-conditions en appliquant le principe de "programming as contracting" [Meyer 88].

3) Regrouper et supprimer les classes
De la même façon que dans la méthode OMT, on examine les classes identifiées. Certaines sont rejetées parce qu'elles sont en dehors du champs du projet. D'autres sont similaires et doivent être unifiées ou regroupées.

4) Résoudre les questions liées au modèle objet
Le concepteur doit se poser les questions suivantes: Comment trouver les bonnes couches d'abstraction? Comment obtenir des structures polymorphes? Comment isoler les informations dépendantes de l'implantation? Selon l'auteur, ces questions ne peuvent trouver de réponses qu'avec une certaine pratique de la conception orienté-objet. Des éléments de réponse sont donnés. Par exemple, en ce qui concerne le polymorphisme, on conseille de regarder les opérations qui doivent s'adapter à des cas particuliers. Dans ce cas, l'opération générale fait partie d'une classe de haut niveau, tandis que les cas particuliers seront réalisés dans des classes descendantes. Une

deuxième idée est d'envisager les modifications et extensions qui peuvent se produire. La conception pourra être rendue plus flexible.

5) Ajouter des classes "de second plan"
Des classes non connectées au monde réel peuvent apparaitre au fur et à mesure que la conception devient plus détaillée. Ces classes appelées "de second plan", par opposition aux classes initiales "de premier plan", assurent un certain nombre d'objectifs: compléter des classifications, découper les classes trop complexes ou trop grosses, assurer des responsabilités, servir d'informations partagées etc...

6) Typer les caractéristiques
On doit vérifier que toutes les caractéristiques listées dans les corps de classes, ainsi que les arguments et résultats d'opérations, sont typés par un nom de classe.

7) Connecter les classes: on vérifie que toutes les classes sont connectées. Il ne doit pas exister de classe isolée.

8) Itérer: Le processus précédent est itéré afin d'affiner de plus en plus l'analyse et la conception.

3.5.6. Commentaires

La démarche de J.M Nerson est comparable à la notre parce qu'elle part d'un langage (EIFFEL), reconnu pour la richesse de ses concepts, et propose une méthode d'analyse et conception les utilisant. On trouvera des similitudes entre OOAD et MECANO, en particulier dans la spécification des classes comme des types abstraits et dans le traitement graphique de la généricité. Les structures proposées (clusters) n'offrent cependant pas la richesse sémantique des domaines et des composites de MECANO. Ils constituent des ensembles de classes peu structurés, les règles de constitution se limitant à la connexité des classes contenues. Les règles permettant de mettre en relation des clusters et des classes ou d'autres clusters sont intéressantes parce qu'elle permettent de faire abstraction des classes internes à un cluster. En MECANO, nous verrons que cette abstraction est gérée au moyen des entités internes et externes.

4. SYNTHESE

Le tableau de la figure 21 présente une synthèse comparative de l'étude des méthodes orienté-objet. Les colonnes du tableau représentent les méthodes, les lignes indiquent les caractéristiques ou concepts présents dans les différents modèles, un point indique la présence du concept. Dans certain cas nous précisons le nom donné au concept par la méthode.

4.1. Présentation de la synthèse

Notre comparaison s'appuie sur les phases du cycle de vie traitées (lignes 1 à 3), les concepts du modèle objet proposés (ligne 4 à 10) et la prise en compte de notions propres à la conception (ligne 11 à 16).

La dernière colonne concerne la méthode que nous avons développée: MECANO. La lecture de cette colonne permet d'introduire cette méthode comparativement aux autres approches. Les termes utilisés seront définis dans la partie suivante (Chapitres 3 et 4). Nous invitons le lecteur, après avoir pris connaissance de ces chapitres, à revenir à ce tableau qui offre une vue globale de l'apport de notre démarche par rapport aux autres approches.

4.1.1. Couverture du cycle de vie

Analyse, Conception, Implantation
Nous rappelons les phases du cycle de vie couvertes par la méthode. La distinction entre analyse, conception et implantation est surtout sensible dans les méthodes dérivées de l'OOD. Pour les autres, les phases d'analyse et de conception sont moins distinctes, la seconde étant vue comme un enrichissement du modèle produit par la première. Cette distinction est encore moins sensible entre conception et implantation car les concepts de conception sont généralement directement exprimables dans le langage orienté-objet cible.

4.1.2. Concepts du modèle objet utilisés

Objet
Le concept d'objet est évidemment présent dans toutes les méthodes bien que celui-ci revêt une sémantique différente selon les cas. Certaines méthodes comme HOOD ou MACH2 considèrent l'objet comme une entité du monde conceptuel (un module) qui peut exister sans classe préalable. Les autres méthodes considère généralement l'objet comme une entité du monde opérationnel, c'est à dire une instance de classe dynamiquement créée à l'exécution.

Classe
Logiquement, le concept de classe est offert par toutes les méthodes sauf HOOD et MACH2. Il faut cependant préciser que MACH2 introduit la notion de type de sous-système, de processus et de service, mais le concepteur peut aussi travailler directement sur les objets réels sans créer de type au préalable.

Héritage
L'héritage, non proposé par les méthodes dérivées de l'OOD est présent dans les autres approches. Ce concept est plus ou moins bien décrit selon les approches. ObjectOry donne peu d'information sur les structures d'héritage, OMT et OOA ne considérent que l'héritage de spécialisation. Class Responsabilities propose le mécanisme de classe abstraite et plusieurs formalismes pour représenter les hiérarchies d'héritage. OOAD utilise un concept d'héritage proche du langage Eiffel: héritage indépendant de l'interface, classe abstraite, contrainte sur les assertions, polymorphisme sur les liens d'héritage, généricité contrainte cohérente avec l'héritage. MECANO s'appuie sur les mêmes concepts mais apporte en plus une taxonomie de la relation d'héritage. Cette taxonomie offre une panoplie de liens d'héritage pour des utilisations types adaptées aux phases d'analyse, de conception ou d'implantation.

Polymorphisme
Le polymorphisme est offert dans OMT et Class Responsabilities. Il est associé à la liaison dynamique et indépendant de l'héritage. OOAD offre un polymorphisme sur la relation d'héritage. MECANO intègre le polymorphisme dans le processus de développement et en fait un concept clé de l'évolutivité et du mécanisme de configuration. Les objets candidats au polymorphisme doivent vérifier des contraintes (héritage, interfaces compatibles ...).

Généricité
La généricité offerte par EIFFEL se retrouve dans OOAD et MECANO. Class Responsabilities propose des concepts proches: relation "container" -héritage de contrat sur des éléments de container- sans aller cependant jusqu'au concept de généricité.
On constatera que la généricité n'est pas proposée par les méthodes de l'OOD, alors qu'elle est présente dans le langage cible ADA.

METHODES / CONCEPTS	HOOD	MACH2	OMT	COA	ObjectOry	Class Responsab.	OOA&D	MECANO
Analyse			•	•	•	•	•	•
Conception globale	•	•	•		•	•	•	•
Conception détaillée	•	•	•		•	•	•	•
Objet	•	•	•	•	•	•	•	•
Classe		"Type" d'objet	•	•	•	•	•	•
Héritage			• (multiple)	•	•	• (multiple)	• (multiple répété)	• (multiple/ répété)
Polymorphisme			• (non lié à l'héritage)			• (non lié à l'héritage)	•	•
Classe Abstraite						•	•	•
Assertions							•	•
Généricité							•	•
Association Délégation	"USE"	"UTILISE"	association	association	association de connaissance	Collaboration	Client	Communication
Composition	"INCLUDE"		aggrégation	tout/ partie		•	Classe expansée	Composition
Entité structurante	"Parent/ Enfant"		Sous-système et Module	Sujet	Blocs Composants	Sous-système	Cluster	Domaine Composite Application
Notation graphique	•	•	•	•	•	•	•	•
Divers	objet actif	Types d'objets prédéfinis			types de classe	Notion de Contrat	EIFFEL	Taxonomie de l'héritage

figure 21: tableau de synthèse caractéristiques des méthodes orienté-objet

Assertion

Les assertions constituent un outil de spécification privilégié et sont à la base du principe de "programming as contracting" repris dans OOAD et MECANO. Ce concept, absent des autres méthodes, permet de mieux formaliser la relation de délégation

(obligations et devoirs du client et du fournisseur) et de contrôler l'arbre d'héritage (les redéfinitions doivent respecter le contrat hérité).

4.1.3. Notions propres à la conception

Délégation
La relation de délégation est présente dans toutes les approches. Les méthodes proches de l'analyse préféreront un concept plus déclaratif et parleront "d'association" entre classes ou objets (OMT, OOA, ObjectOry). Les méthodes de conception utilisent un concept plus opérationnel (client, collaboration, USE, communication).

Composition
La composition n'existe pas dans MACH2 et ObjectOry. En HOOD, la composition est réduite à une inclusion syntaxique. Dans d'autres approches (OMT,MECANO), la composition est vue comme un cas particulier de délégation avec des contraintes supplémentaires: création automatique des composants, propagation d'opérations, règles de visibilité ...

Entité structurante
Toutes les méthodes proposent des entités permettant de structurer l'ensemble des classes produites. Ces entités, de granularité plus forte que la classe, permettent d'appréhender l'architecture des conceptions en organisant l'ensemble des classes d'une application, d'un projet ou d'une librairie. Il s'agit dans la plupart des cas, de la notion de sous-système qui regroupe des classes logiquement liées ou connectées par une relation d'héritage ou de délégation.

Formalisme graphique
On notera que toutes les méthodes sans exceptions reposent sur un formalisme graphique. Ce formalisme est en effet le plus adapté pour la description d'architectures de classes. Des outils permettant la gestion de ce formalisme sont d'ailleurs proposé (ou vont l'être). On peut citer OOATool pour OOA, ObjectOrySE pour ObjectOry, MDS (Mecano Design Station) pour MECANO. Le projet Esprit "Business Class" va développer des outils supportant la méthode OOAD de Nerson. Des outils pour HOOD et MACH2 vont aussi être proposés.

Nous rappelons dans la dernière ligne un concept original apporté par la méthode.

4.2. Bilan: apports et limites des méthodes

Le tableau de la figure 20 met en évidence l'écart existant entre les méthodes dérivées de l'OOD et les méthodes de la nouvelle génération.
D'un côté (OOD), on propose une démarche (méthodologie) pour concevoir un programme ADA comme un ensemble d'objets coopérants. Le langage cible ne supportant pas tous les concepts du modèle objet, les méthodes ignorent volontairement ces concepts.
De l'autre côté, les nouvelles méthodes proposent une panoplie de concepts plus riche et un modèle uniforme de l'analyse à l'implantation. Cependant, les graphes de classes élaborés sur les relations d'héritage et de délégation sont complexes et difficiles à gérer. La réponse passe généralement par un mécanisme de sous-système permettant de répartir et de structurer l'ensemble des classes. Les contraintes portant sur la création de ces sous-systèmes sont peu nombreuses, ce qui confère à ce concept une faible sémantique en le réduisant la plupart du temps à un répertoire de classes. Nous verrons que notre démarche dans MECANO est plus affinée puisqu'elle propose trois entités structurantes permettant d'encapsuler des hiérarchies de spécialisation/réalisation (les domaines), des hiérarchies de composition (les composites) et des systèmes ou sous-

systèmes opérationnels (les applications). Chacune de ces trois structures possède des règles de construction et de cohérence qui devront être satisfaite par le concepteur.

4.3. MECANO: vers une construction orienté-objet

Notre approche est ascendante: elle part des concepts du niveau langage (classe, délégation, généricité, classe virtuelle, héritage, assertion) et les adapte au niveau conception (communication, composition, taxonomie de l'héritage). D'autre part, elle apporte des notions propres à la conception: relations inter-entités, structuration, stratégie, entités structurante.

MECANO fait partie des méthodes de la nouvelle génération. Elle est résolument dédiée à une réalisation dans un langage objet, adhérant totalement à la conviction de Korson et McGregor [Korson 90]: "*...High-level design should be language independent but it is not paradigm independent. A truly object-oriented design can be directly implemented only in an object-oriented langage...*"[1].

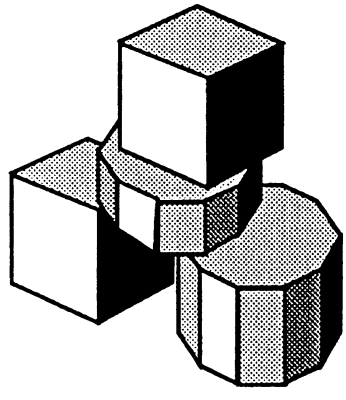
Les chapitres suivants sont consacrés à la présentation de la méthode MECANO. Le chapitre 3 présente les concepts. Le chapitre 4 expose le processus de conception et montre son application sur un exemple. Le chapitre 5 décrit brièvement le poste de conception MECANO.

---

[1] "... La conception doit être indépendante du langage mais ne peut pas être indépendante du paradigme. Une vraie conception orientée objet ne peut être directement implantée que dans un langage orienté objet ...". D'autres auteurs [Henderson-Sellers 90] précisent que si le cycle OOO (analyse, conception et implantation orienté objet) est naturel, le cycle OOF (analyse et conception orienté objet et implantation dans un langage procédural standard) est faisable mais implique un effort de mise en correspondance pouvant générer des erreurs. Le cycle FOO est aussi faisable mais comporte les mêmes inconvénients. L'approche de G. Booch peut avec ces conventions s'apparenter à un cycle FOF: analyse avec SADT, conception orientée objet et codage en ADA.

PARTIE 2:

MECANO METHODE ET ENVIRONNEMENT



CHAPITRE 3:

LA METHODE MECANO

MECCANO, *définition du Larousse encyclopédique [Larousse 77]*:
Jouet de construction en métal, à pièces interchangeables, fondé sur le système des trous équidistants, inventé par F. Hornby.

LES CONCEPTS

1. INTRODUCTION

1.1. Problématique et finalité

Dans cette partie, nous présentons la méthode MECANO [Girod 89a], [Girod 90b], [Girod 90c]. MECANO signifie "Méthode et Environnement de Construction d'ApplicatioNs par Objets", cet acronyme nous paraît correspondre totalement à une philosophie objet privilégiant l'assemblage de briques de bases (les classes) pour construire des structures complexes "de bas en haut"[1].
Le but de MECANO est d'appliquer et d'adapter les concepts du modèle objet bien décrit dans le cadre de la programmation, à l'activité de conception des logiciels. La méthode MECANO se propose de répondre au bilan dressé dans le chapitre précédent. L'idée est de fournir une méthode pour la conception offrant tous les concepts du modèle objet tels qu'ils ont été définis par exemple dans [Meyer 88] et s'attaquant aux problèmes typiques de la conception: à savoir comment structurer un logiciel, lui conférer des qualités de réutilisabilité et d'évolutivité, et utiliser l'héritage à bon escient. En fait, une méthode de conception par objets doit permettre d'exprimer les idées et les décisions du concepteur inexprimables dans un langage de programmation qui doit rester souple et général. Notre méthode va dans ce sens et propose de nouveaux concepts proches des préoccupations du concepteurs. Elle agit à la fois comme un moyen d'expression plus riche et aussi comme un catalyseur de l'activité créatrice voire un outil pédagogique permettant de mieux comprendre et appliquer le modèle objet. Enfin une méthode de conception par objet doit offrir à son utilisateur un niveau d'expression d'une granularité plus forte qu'un langage, et doit permettre de décrire les classes et leurs interconnexions sans être obligé de rédiger le code de chaque méthode.
Le poste de conception décrit au chapitre 5 illustre parfaitement cet objectif: il propose un langage graphique et un certain nombre de contrôles assurant la gestion d'une conception MECANO. Il permet au concepteur d'évoluer à la fois dans un cadre conceptuel précis mais suffisamment souple pour gérer les incohérences et incomplétudes passagères inévitables à toute activité créatrice.

1.2. Présentation

Pour répondre à cette problématique, la méthode va proposer une extension du modèle objet en apportant d'une part un affinement des deux relations inter-classes: héritage et délégation, et en fournissant d'autre part de nouveaux types d'entité structurante pour organiser l'ensemble des classes produites et/ou utilisées dans une conception. Les entités structurantes prendront en compte aussi bien les problèmes de réutilisabilité et d'évolutivité grâce aux **domaines** qui fournissent un moyen de généralisation sur la relation d'héritage, que les problèmes de hiérarchisation et de visibilité d'objets grâce aux **composites** qui proposent une structure d'emboîtement, sans oublier la construction de modèles opérationnels à travers la mise en place des **applications**.

Dans la première partie, les concepts objets qui constituent les notions de base de MECANO sont décrits. Cette partie peut paraître redondante avec le chapitre 1, mais permet d'établir une base aux autres concepts de MECANO et de définir le vocabulaire qui sera utilisé.
Ensuite, les relations inter-classes sont présentées. Il s'agit de l'héritage et de la relation client-serveur (délégation) qui constituent les seuls moyens de connexion entre les

[1] Voir la définition de MECCANO tirée du Larousse Encyclopédique à la page précédente.

classes. L'héritage est défini dans le contexte de la conception. Nous verrons que l'héritage, vu au niveau conceptuel, ne revêt pas une mais plusieurs significations selon le contexte d'utilisation. MECANO est la première méthode à introduire un typage de la relation d'héritage dans la conception par objets. Ainsi le concepteur aura à sa disposition non plus un héritage mais une variété de types différents, utilisables selon des contraintes bien définies. La relation de délégation entre une classe utilisateur et une classe fournisseur d'un service sera ensuite définie et affinée.
Dans la partie suivante, les entités structurantes sont présentées. Le typage d'héritage sera à la base de la notion de Domaine, entité orientée vers la réutilisation. La relation de clientèle permettra de constituer d'autres entités : les Composites et les Applications.

2 . LES CONCEPTS DE BASE

Afin d'introduire les concepts plus avancés de MECANO, nous présentons ici les notions de base de la méthode[1]. Le paradigme objet est fondé sur le concept de classe décrivant un modèle d'**objets** ou instances. Une instance est composée d'un état décrit par la valeur de ses **attributs**, et d'un comportement (ensemble des **méthodes** applicables à l'état). La **spécification** permet de décrire le comportement d'un objet indépendamment d'une implantation. L'**interface** fait partie de la spécification et assure la propriété de masquage d'information.

2.1. Classe et Objet

La classe est la brique de base de MECANO. En première définition, on dira qu'une classe est une implantation de type abstrait (cf chapitre 1). Un type abstrait peut être vu comme la définition d'un comportement d'une donnée d'un point de vue externe. Dans certains cas, une classe pourra représenter le type abstrait lui-même, une telle classe ne donnant pas lieu à la création d'instance est appelée virtuelle. Son rôle est important dans la conception puisqu'elle fournit un moyen d'abstraction. En fait, la classe virtuelle est utilisée conjointement à l'héritage (cf §4) afin de décrire la connaissance relative à un niveau d'abstraction donné sans se soucier d'une implantation particulière.

Pour implanter un type abstrait, une classe est constituée de **caractéristiques** statiques et dynamiques.
Les caractéristiques statiques, appelées **attributs** sont définies par une paire *nom/type* où *nom* est un identificateur et *type* le nom d'une classe[2]. A partir d'une classe, on peut créer des objets qui seront actifs à l'exécution: les instances de la classe. Chaque objet instancié d'une même classe aura les mêmes attributs avec un ensemble propre de valeurs : c'est l'état interne de l'objet. La valeur d'un attribut peut être soit une valeur de type basique : caractère, entier, chaîne, réel etc... ou bien une référence à un autre objet dont la classe a été définie par la paire attribut/type de l'objet courant.
Les caractéristiques dynamiques ou **méthodes** sont communes à tous les objets d'une même classe. Il y a deux types de caractéristiques dynamiques : les fonctions et les actions. Les **fonctions** retournent un résultat d'un type donné (classe) et n'affectent

---

[1] Le lecteur pourra trouver dans [Meyer 88] et [Korson 90] les définitions plus précises des concepts objets repris par MECANO. On ne sera pas surpris de trouver à la base de MECANO la majorité des concepts introduits par Eiffel [Meyer 87b]. En effet, ce langage commence à être reconnu comme l'un des plus complets [Boussard 90] et des plus rigoureux [Korson 90] , "... mettant surtout l'accent sur la réutilisation et la fiabilité ..." [Masini 89]. Une étude d'Eiffel pourra être trouvée dans [Morat 87] et dans [Boussard 90].

[2] Dans la suite, les termes *type* et *classe* seront employés avec la même signification.

pas l'état interne de l'objet. Les **actions** modifient l'état de l'objet sans renvoyer de résultats.
Attributs, fonctions et actions permettent d'implanter un type abstrait. La structure d'une donnée du type est stockée par les attributs d'une instance particulière, les fonctions accèdent à cette donnée, les actions la modifient par effet de bord (cf chapitre 1).

Le seul mécanisme d'interaction entre les objets est l'envoi de messages. Un message est composé du nom d'une méthode, éventuellement de paramètres effectifs et d'un destinataire. Le destinataire reçoit le message et le traite c'est à dire sélectionne la bonne méthode et l'exécute. Si la méthode est une fonction, l'objet expéditeur attend un résultat. MECANO ne reconnait pour le moment que le modèle séquentiel dans lequel le contrôle est transféré au destinataire et l'expéditeur est mis en attente jusqu'à la fin de l'exécution de la méthode[1].

2.2. Spécification et interface

MECANO fournit d'autres notions permettant de mettre en place une conception basée sur les types abstraits. Il s'agit de la spécification de classe, de la spécification de méthode et de l'interface de classe.

La **spécification de classe** contient des informations liées à la gestion du projet en cours de développement (auteur, date, version, état de conception, rôle de la classe ...) mais aussi une spécification plus formelle du type abstrait implanté par la classe au travers de l'**invariant** de classe et de l'**interface**.

Les données informelles sont constituées de couples attribut/valeur à la manière des index d'EIFFEL. Ces couples, mis en place par le concepteur permettent de caractériser une classe au sein d'un projet. Ces informations seront consultées par les utilisateurs de la classe et par des outils de recherche associés à la méthode (cf Chapitre 5).

Une classe doit pouvoir explicitement décrire les caractéristiques dynamiques disponibles pour les autres classes. C'est le rôle de l'**interface** dans laquelle le concepteur exprime les fonctions, actions et attributs[2] propres à la classe. Ces caractéristiques correspondent aux opérations du type abstrait. Les caractéristiques internes -utilisées uniquement à des fins d'implantation- sont cachées de l'extérieur. Les caractéristiques présentes dans l'interface sont dites **exportées** par la classe.

L'**invariant** de classe permet d'exprimer une propriété logique devant être vérifiée à tout moment par toutes les instances de la classe. L'invariant permet de décrire certains axiomes du type abstrait: par exemple, la propriété contraignant l'indice d'un tableau à être compris entre les bornes du tableau.

La **spécification de méthode** assure la description d'une méthode indépendamment de son implantation. Cette spécification est constituée du nom de la méthode, des paramètres typés, du type du résultat pour les fonctions, d'une pré-condition: condition devant être satisfaite à l'appel, et d'une post-condition: condition assurée au retour. Les

---

[1] Le modèle objet se prête particulièrement bien au parallélisme [De Mare 90]. Un certain nombre de travaux témoignent de l'intérêt porté à cet axe de recherche qu'il s'agisse d'extensions à un langage séquentiel [Caromel 90] et [Meyer 90a], ou de modèles plus directement dédiés au parallélisme comme les langages acteurs.

[2] Dans ce cas, l'attribut est considéré comme une fonction sans argument dont la valeur n'est accessible qu'en lecture.

pre-/post-conditions permettent d'exprimer au niveau de la classe une partie des axiomes sur les opérations du type abstrait.
Par exemple, la post-condition de l'action *empiler(x)* d'une classe PILE peut être :

x = sommet et non pilevide et taille <- old(taille) + 1[1]

qui assure qu'après avoir empilé un élément, la pile n'est pas vide, sa taille a été incrémentée et l'élément se trouve au sommet de pile.
Les commentaires informels permettent de prendre en compte les axiomes impossibles à exprimer sous forme pré-/post-condition, comme par exemple l'axiome de la PILE:

empiler(x,depiler(P)) = P.

```
Methode EMPILER(x: T)
    precondition
        non pile_pleine
    postcondition
        non pilevide et sommet=x et taille=old(taille) + 1
        et {l'élément x a été empilé}
```

Les pré- et post-conditions font partie de l'approche de la programmation par contrat [Meyer 88]: lors de l'envoi d'un message entre un objet expéditeur et un objet destinataire, si la pré-condition est respectée par l'appelant avant l'exécution alors la post-condition sera garantie par le destinataire en fin d'exécution. Autrement dit, tout se passe comme si un contrat avait été établi entre les deux classes correspondantes, stipulant les devoirs et responsabilités de chacune: n'utiliser une méthode que dans les cas prévus (pré-condition= obligation du client), assurer qu'une méthode utilisée correctement rend bien le service attendu (post-condition= obligation du fournisseur).

2.3. Classes génériques

La généricité est un concept provenant de la théorie des types abstraits. Elle apporte une paramétrisation orthogonale à l'héritage en proposant de définir des modèles de classes [Meyer 90b]. Par exemple, la classe ENSEMBLE_ENTIER peut être issue d'une classe générique ENSEMBLE[T] où le type formel T sera instancié au type effectif ENTIER. N'importe quel autre ensemble d'éléments pourra être dérivé de ENSEMBLE[T].
La généricité est un moyen permettant d'associer souplesse et rigueur. La souplesse est apportée par la paramétrisation qui permet de décrire une classe en faisant abstraction de détails non pertinents, la classe générique pouvant ensuite être réutilisée en fixant les paramètres génériques. La rigueur est apportée par la généricité contrainte définie ci-dessous.

Généricité simple et contrainte

Avec la généricité simple, les paramètres génériques formels remplacent n'importe quelle classe. Ces paramètres formels peuvent ensuite être utilisés pour typer les attributs, les paramètres et les résultats des méthodes. Par contre, on ne peut appliquer de méthodes aux entités de type générique car ce type est par définition inconnu. On ne peut que passer l'entité en paramètre de méthode ou l'assigner à une valeur de même type générique formel.
La généricité contrainte permet d'assouplir cette situation en reprenant à son compte le concept de propriété des types abstraits génériques. Le type générique devient **contraint** par un type effectif: la propriété. Dans ce cas, toutes les méthodes du type effectif peuvent être appliquées à un objet du type générique. Un paramètre générique formel contraint doit vérifier que les paramètres génériques effectifs sont compatibles

[1] old(taille) renvoie la valeur de taille avant l'exécution de empiler.

au type de contrainte. Ceci permet de décrire des structures génériques plus riches, qui font intervenir des opérations sur les éléments du type générique.
Dans notre exemple sur les ensembles, on contraint les éléments de l'ensemble à être COMPARABLE afin de disposer de l'égalité pour définir par exemple la méthode d'insertion d'un élément dans un ensemble[1].
On remarquera que l'utilisation de la généricité est tout à fait contrôlée: ou bien la généricité simple est utilisée et l'élément générique ne peut être que sommairement manipulé, ou bien le paramètre générique est contraint et on peut lui appliquer les méthodes du type de contrainte mais le type générique effectif devra alors être compatible à la contrainte[2].

2.4. Relations inter-classes

A partir des concepts précédents, MECANO définit une architecture basée sur deux types de relation: la délégation et l'héritage. Ces relations permettent d'exprimer les connexions entre les classes et définissent l'architecture du futur logiciel. Nous enrichissons au §3. et §4. ces deux concepts, qui seront ensuite utilisés dans la définition des entités structurantes (§5.).

3. LA RELATION DE DELEGATION

La première relation inter-classe de MECANO est la relation de délégation ou "client". Elle se définit de la façon suivante:

Définition
une relation de "Délégation" est établie entre deux classes A et B quand A définit un attribut de type B.

La classe A est appelée "client" et la classe B "fournisseur". A l'exécution, toute instance de A peut envoyer des messages à une instance de B.

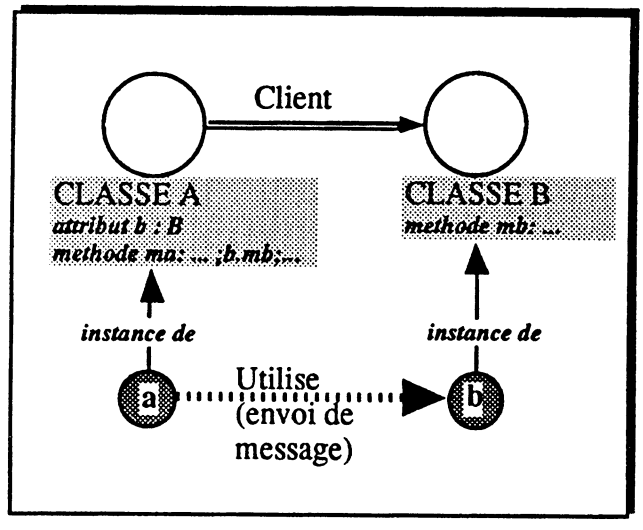


figure 1 : relations CLIENT et UTILISE

[1] Axiome du type ensemble: insérer(x, insérer(y,E)) = si x=y alors insérer(y,E)
[2] Pour la généricité voir aussi le §5.

On parle au niveau instance de relation d'utilisation. C'est cette relation qui, étendue au niveau des classes, est appelée relation de délégation (figure 1). Elle a donc beaucoup d'importance puisque c'est elle qui va supporter le flux de contrôle (envoi de messages) d'une application.
L'intérêt d'une telle relation est de mettre en évidence les coopérations entre les classes.

La relation client doit être affinée pour tenir compte de la nature du lien entre le client et le fournisseur. MECANO propose trois sortes de relation client: Communication, Composition et Utilisation secondaire.

3.1. Communication

Le premier type de relation client est la "communication" ou "délégation collatérale" dont la définition est la suivante:

Définition
A "communique" avec B si A est client de B et si A et B sont sur le même niveau conceptuel. Les instances de B sont indépendantes de A et doivent être créées en dehors de A.

On dit que A est demandeur de service pour B, ou bien que A veut être informée par B. Ce concept est proche de celui de la communication inter-processus. Dans l'approche parallèle, A et B seraient implantées comme des processus indépendants, chacun ayant son propre cycle de vie [Caromel 90]. La création d'une instance de B associée à A n'est pas du ressort de A. Pendant l'exécution, l'objet référencé par A peut changer.
Dans l'exemple de la figure 2, l'Avion est associé à une Tour de contrôle. Ce sont deux entités indépendantes ayant besoin de coopérer pendant un certain temps. Des messages sont envoyés par l'Avion (demande de piste) qui attend un résultat de la Tour de contrôle en retour (autorisation d'atterrissage). Au cours d'un même voyage, l'avion communiquera successivement avec différentes tours de contrôle.

Des relations proches de la communication sont fournies par certaines méthodes: il s'agit de l'"association" [Rumbaugh 91], [Coad 91] , de la "collaboration" [Wirfs-Brock 90] , ou de la relation "client" [Nerson 91] (cf chapitre 2).

```
classe AVION
attribut      communication
     controle:  TOUR_CONTROLE;
methode
     atterrir :
          controle.demande_piste;
          si controle.autorisation_atterrissage = OK
          alors      ...
fin classe AVION
```

figure 2: exemple de communication

3.2. Composition

Le deuxième type de relation client est la "composition" ou "délégation hiérarchique" définie par.

> *Définition*
> *A "est composé" de B si A est cliente de B et si B fait partie de A en tant que composant. En particulier, la création d'une instance de A induit la création d'une instance de B associée à A.*

A et B sont sur des niveaux hiérarchiques différents: B représente une partie de A. En tant que partie, B devra être instanciée par A et lui appartiendra ensuite. De plus, B ne peut être atteinte que via A, elle est masquée de l'extérieur.
A utilise ses composants pour réaliser ses propres services de manière interne.
Du fait de ces contraintes, la Composition est un lien plus fort que la Communication. Cependant, il faut remarquer que les contraintes de visibilité et d'accès ne s'appliquent qu'au niveau des instances et non aux classes elles-mêmes. Dans le cas contraire, les classes masquées ne seraient pas utilisables en dehors de leur client de composition ce qui est contraire aux buts de généralisation et de réutilisation que nous nous sommes fixés.

La relation de composition est une relation hiérarchique stricte (tout/parties) donc acyclique au niveau des instances, ce qui n'est pas vrai au niveau des classes car la fonction **f: {relation d'utilisation} -> {relation délégation}** n'est pas bijective. Pour s'en convaincre, il suffit de regarder l'exemple de la figure 3 dans lequel AVION possède deux composants: aile droite et aile gauche, instances de la même classe AILE.
Dans cet exemple, des liens de composition sont établis entre l'avion et ses composants (ailes et le train d'atterrissage). Il y a bien dans ce cas une association unique entre les objets. Les ailes et le train d'atterrissage sont propres à l'avion, ils existent dès la création de l'avion. L'avion utilise ses ailes pour assurer ses propres services: voler, atterrir et le train d'atterrissage est sorti suite à une demande d'atterrissage adressée à l'avion.

```
classe AVION
attribut    composition
     aile_droite, aile_gauche  :  AILE;
     train_atterrissage  :  TRAIN_ATTER;
     moteur  :  MOTEUR_AVION;
attribut    communication
     controle : TOUR_CONTROLE;
methode
     atterrir :
          controle.demande_piste;
          si controle.requête_atterrissage = OK
          alors     train_atterrissage.sortir;
                    aile_droite.lever_volets;
                    aile_gauche.lever_volets;
                    moteur.reduire_gaz  ...
fin classe AVION
```

figure 3 : exemple de composition

Le concept de composition se retrouve dans les nouvelles méthodes sous différents noms: agrégation (OMT), Tout/parties (OOA) et dans certains langages (LOOPS [Xerox 86], YAFOOL [Ducournau 88]). Ce concept est un affinement du concept d'association et implique une relation plus forte entre deux classes. Il correspond à un concept présent dans le monde réel et permet de modéliser des structures composées (voir la section §5.3. sur la structure composite).

3.3. Utilisation secondaire

Afin de prendre en compte les cas d'utilisation (relation client) ne correspondant ni à la communication, ni à la composition, MECANO propose un troisième type de lien appelé **utilisation secondaire** (abrégé en "utilisation"). Cette relation inter-classes correspond à la sémantique classique de l'utilisation dans les langages de programmation. Elle n'a pas de contrainte d'application. Sa présence dans MECANO est motivée par la mise en oeuvre de la conception détaillée et pour l'utilisation de classes à titre purement utilitaire non significatives dans la conception globale (messages d'erreurs, variables auxiliaires etc...). Par exemple, dans la figure 4, un attribut utilitaire est utilisé pour assurer l'impression d'un message à chaque atterrissage[1].

```
classe AVION
attribut   composition
     aile_droite, aile_gauche : AILE;
     train_atterrissage : TRAIN_ATTER;
     moteur : MOTEUR_AVION;
attribut   communication
     controle : TOUR_CONTROL;
attribut   utilitaire
     atterrissage_OK:  STRING
          = "atterrissage autorisé par contrôle"
methode
     atterrir :
          controle.demande_piste;
          si controle.requête_atterrissage = OK
          alors     atterrissage_OK.print;
                    train_atterrissage.sortir;
                    aile_droite.lever_volets;
                    aile_gauche.lever_volets;
                    moteur.reduire_gaz ...
fin classe PLANE
```

figure 4 : exemple d'utilisation secondaire

[1] Cet affichage peut servir de trace pour le déverminage ou bien pour la mise en place d'une boite noire dans l'avion. On peut néanmoins penser que dans ce deuxième cas, la gestion de la boite noire sera mieux intégrée à l'ensemble du système et que de nouvelles classes et attributs de composition et/ou de communication seront définis dans ce sens: BOITE_NOIRE, EVENT, enregistrer_event etc ...

4. HERITAGE : DEFINITION ET TAXONOMIE

4.1. Définition de l'héritage

L'Héritage est une des notions centrales du modèle objet. C'est une relation inter-classes qui peut être définie de la manière suivante [Meyer 88]:

Définition
B hérite de A:
Les caractéristiques (attributs et méthodes) de A sont disponibles dans B comme si elles lui étaient propres.
De nouvelles caractéristiques peuvent être définies par B.

Vocabulaire

Le vocabulaire associé à l'héritage est très varié: selon les auteurs, on parlera de sous-type, de spécialisation, de sous-classe, de classe héritière etc... Pour notre part, nous employons le vocabulaire suivant:

B hérite de A <=>	
	B est une sous-classe de A A est une super-classe de B B est la fille de A A est la mère de B

Il existe un chemin sur la relation d'héritage de B à A <=>	
	B est descendante de A A est ancêtre B

La relation d'héritage est transitive: une classe hérite de tous ces ancêtres. On parlera de descendance d'une classe pour désigner le sous-arbre d'héritage issu de cette classe.

Compatibilité

Un objet de type T1 est dit compatible à un objet de type T2 si T1 = T2 ou T1 hérite de T2. La notion de compatibilité permet de statuer de façon statique sur la validité d'une expression ou d'une instruction. La règle générale est que tout objet de type T2 peut être remplacé par un objet de type T1 si T2 est compatible à T1[1].

Redéfinition d'une caractéristique

Une caractéristique héritée peut être redéfinie pour l'adapter à son nouveau contexte (la classe fille). Par exemple, la fonction surface peut être redéfinie entre

[1] Les règles complètes de compatibilité, en particulier celles liées à l'utilisation de la généricité, sont décrites dans [Meyer 88].

POLYGONE_QUELCONQUE et CARRE puisqu'il existe un algorithme plus performant pour calculer la surface du carré.
La redéfinition peut aussi s'appliquer aux attributs mais dans ce cas l'attribut de la classe fille doit être compatible au même attribut de la classe mère. Pour les méthodes, les paramètres doivent être compatibles. Cette règle est connue sous le nom de "covariance".

Classe virtuelle

Une classe virtuelle permet de décrire un objet indépendamment d'une implantation. Elle contient donc des spécifications de méthodes (méthode virtuelle). Par extension, une classe est virtuelle si elle contient au moins une méthode virtuelle; ce sont les classes descendantes qui seront chargées de réaliser les méthodes non définies.
Une classe virtuelle ne peut pas donner lieu à la création d'instance, elle n'existe que comme abstraction dans la conception. On peut par contre utiliser une caractéristique ayant pour type de définition une classe virtuelle; cette caractéristique devra, à l'exécution, être associée à un objet de type concret[1] compatible au type de définition.
Les classes virtuelles constituent un puissant moyen d'abstraction pour la conception. MECANO utilise ce concept pour définir de nouvelles entités de structuration (cf §5.).

Renommage d'une caractéristique

Une caractéristique peut être renommée, ce qui permet d'adapter le nom au contexte local mais aussi de garder accès à l'ancienne version d'une caractéristique redéfinie.
Exemple : la fonction *ajout_en_tete* d'une structure de LISTE pourra être renommée *empiler* dans une classe descendante PILE_CHAINEE.
Dans le deuxième cas, le renommage est associé à une redéfinition de la méthode. On garde ainsi l'ancien corps de la méthode sous un nouveau nom, ce qui permet de simuler l'héritage de méthode.

Héritage multiple

MECANO admet l'héritage multiple : une classe peut avoir plusieurs classes mères. Nous verrons au §4.3. que ceci est indispensable pour développer une conception par objets [Meyer 88b]. Par contre, l'héritage multiple peut conduire à des conflits si une caractéristique est héritée plusieurs fois, soit qu'elle provienne de deux classes ancêtres, soit qu'elle provienne de la même classe par héritage répété. Dans ce cas, les mécanismes de redéfinition et de renommage permettent de résoudre les conflits d'héritage.

Polymorphisme

MECANO utilise le concept de polymorphisme conjointement à l'héritage. Un objet d'une classe donnée peut être vu comme un représentant d'une classe ancêtre, on parle alors d'objet polymorphe puisqu'il peut prendre plusieurs formes. A l'exécution, une référence d'objet possède un type dynamique qui permettra de déterminer la bonne réalisation de la méthode à appliquer lors de l'envoi d'un message à la référence. Le fait de limiter le polymorphisme entre classes compatibles permet de conserver le contrôle de type et assure que les objets polymorphes respectent les contrats des méthodes (cf ci-dessous).

Héritage et assertion

---

[1] Une classe non virtuelle est dite concrète.

Une méthode héritée est utilisable dans la classe descendante. A ce titre, les pré- et post-conditions doivent rester valides. Dans le cas d'une redéfinition, la règle de MECANO est que la nouvelle pré-condition vient s'ajouter en "OU" affaiblissant ainsi la pré-condition, et la post-condition vient s'ajouter en "ET" renforçant ainsi la post-condition. Nous rappelons qu'avec le polymorphisme, un même message de la classe cliente pourra être envoyé à des objets appartenant à des classes différentes (mais compatibles), selon le type dynamique du destinataire. Si l'on compare l'héritage à une sous-traitance, cette règle permet d'assurer au client de la classe mère que le contrat est respecté par le sous-traitant (classe fille).
En ce qui concerne l'invariant de classe, il sera traité comme une post-condition. L'héritage d'invariants se fera donc par conjonction (ET).

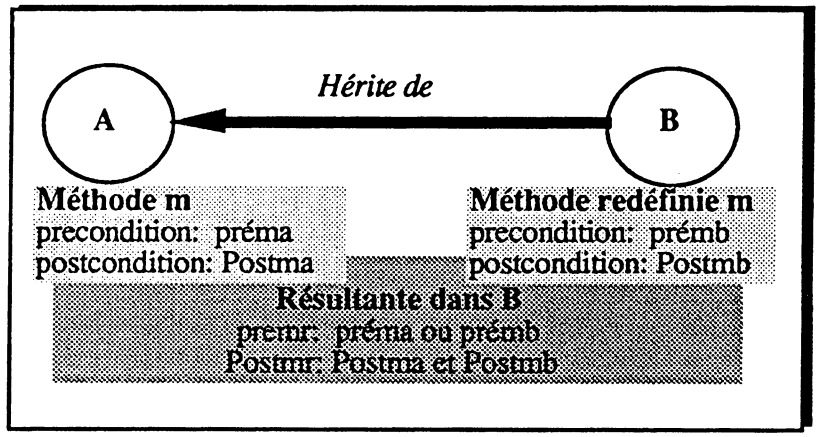


figure 5 : héritage des assertions

Dans le cas de la figure 5, considérons une classe C cliente de A et possédant un attribut **a** de type A. Le message **a.m** est valide et d'après le contrat passé entre C et A, si la pré-condition préma est assurée par C avant l'exécution de **m**, alors A assure la post-condition Postma après l'exécution de m, ce que l'on notera par:

{prema} c.m {Postma}

Considérons un autre attribut b de type B dans C. La séquence d'instruction: *a := b ; a.m* donne à a le type dynamique B, et envoie le message a.m à l'objet référencé par a. Dans ce cas, la méthode exécutée est celle définie dans B.
Du point de vue de la classe C, la pré-condition valide reste prema et la post-condition attendue est Postma. Cependant, la pré-condition premb de m dans B doit être satisfaite et B ne sait satisfaire que la post-condition Postmb. Nous voyons que ceci est possible si les pré- et post-condition complètes de m dans B vérifient:

{prema} => {premr}
et {Postmr} => {Postma}

Cette expression est rendue vraie avec

premr = prema OU premb
et Postmr = postma ET postmb

Si la méthode n'est pas redéfinie, les assertions sont implicitement établies à:

premb= faux donc premr = prema

Postmb= vrai donc Postmr = Postma[1]

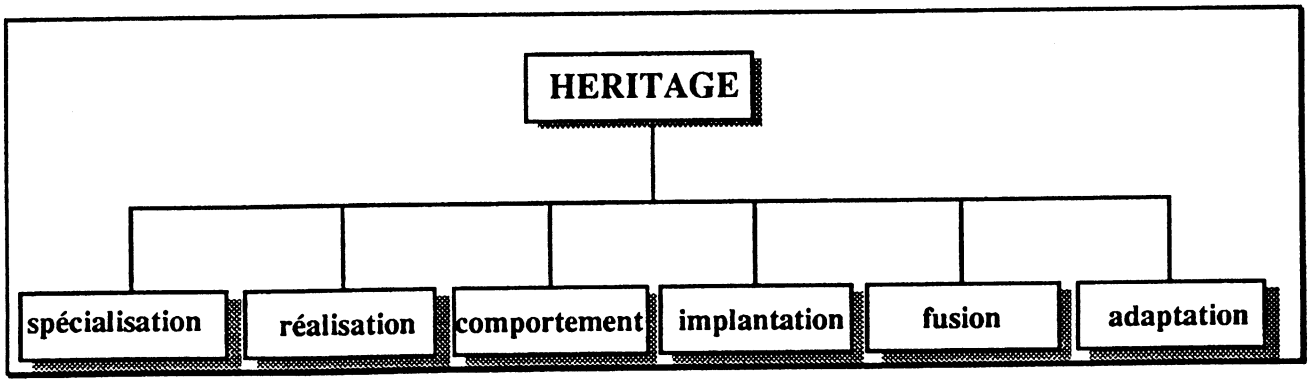
Héritage et interface

Si une classe hérite des caractéristiques de sa super-classe, il n'en est rien pour l'interface. Une classe fille n'offre aucun service à l'extérieur tant qu'elle n'énumère pas explicitement les caractéristiques exportées. Ce choix apporte une grande souplesse à la conception: on peut aussi bien enrichir le service offert en ajoutant les caractéristiques héritées dans l'interface, que le restreindre en n'exportant plus certaines caractéristiques. La méthode utilise ce mécanisme pour définir différent types d'héritage.

4.2. Taxonomie de l'héritage

4.2.1. Pourquoi classifier l'héritage?

Au niveau programmation, l'héritage est une technique permettant de définir incrémentalement une classe à partir d'une autre classe: B hérite de A signifie que B possède toutes les caractéristiques de A et peut les utiliser comme si elles lui étaient propres. Toute classe peut hériter de n'importe quelle autre, quelle que soit la signification que l'on donne à cette relation.
Au niveau conception cependant, l'héritage est plus conceptuel. La sémantique attachée à tel ou tel lien d'héritage doit être explicitée par le concepteur [Snyder86], [Halbert 87]: l'héritage intervient-il sur les spécifications ou sur les implantations? De nombreux auteurs se sont penchés sur cette épineuse question: certains préconisent de n'utiliser que la spécialisation (héritage de spécification) et de proscrire toute autre utilisation [Sakkinen 89], [Rumbaugh 91[2]]; d'autres préfèrent laisser le programmeur libre de son choix [Halbert 87], [Meyer 88].
Pour notre part, nous exploitons une solution intermédiaire [Girod 90c]: la taxonomie de l'héritage. Elle consiste à répertorier et codifier les différents types d'héritage afin d'offrir au concepteur non plus une, mais plusieurs relations d'héritage avec des sémantiques différentes. Quand le concepteur met en place un lien d'héritage, il le fait pour une raison particulière. Avec les types d'héritage il aura le moyen d'exprimer cette raison.
Le typage de l'héritage permettra d'affiner la relation existant entre une classe et sa superclasse. MECANO propose 6 types d'héritage différents (figure 6).



1 Vrai et faux sont respectivement élément neutre de l'opérateur ET et de l'opérateur OU.

2 "there is tension between use of inheritance for abstract data type and for sharing implementation" [Rumbaugh 91 p 64].

figure 6: taxonomie de l'héritage

Pour chacun des types, nous donnons une définition, un schéma décrivant un exemple et les contraintes à respecter pour mettre en place une relation d'héritage de ce type. Les contraintes permettent de spécifier les différents types et obligent le concepteur à respecter cette spécification. Nous verrons que ces contraintes sont automatiquement garanties dans l'environnement MECANO (cf Chapitre 5).

Nota : dans la suite, nous considérons une relation d'héritage entre deux classes A et B dans le sens B hérite de A.

4.2.2. Spécialisation

Définition
une classe A est spécialisée par une classe fille B si B offre de nouvelles caractéristiques en plus de celles héritées, en vue de raffiner le concept exprimé par A

Exemple
CARGO spécialise BATEAU. En effet, un BATEAU est composé d'une coque et d'un moteur et possède la méthode flotter. Un CARGO est un BATEAU qui possède des caractéristiques additionnelles pour pouvoir transporter, charger, décharger des marchandises.
Dans la figure ci-dessous, les termes précédés d'un point correspondent aux caractéristiques exportées par la classe.

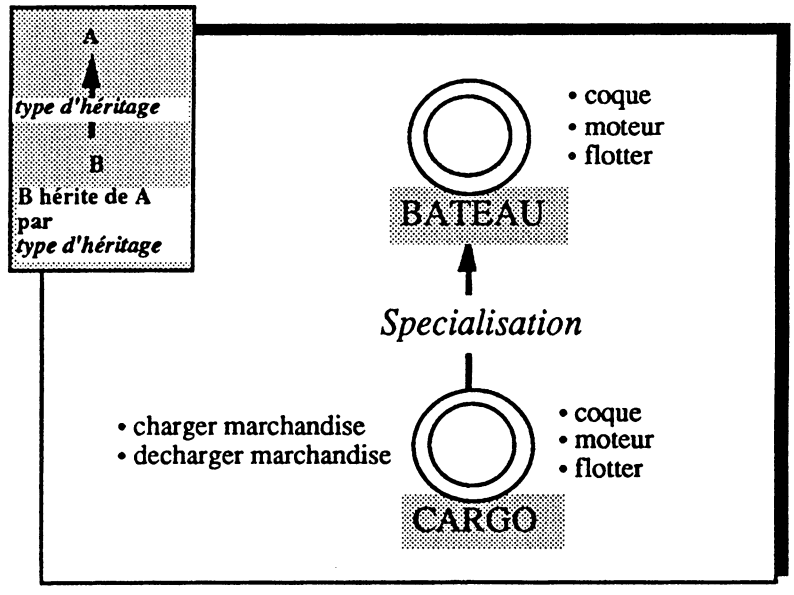


figure 7 : Exemple de spécialisation[1]

Quand un concepteur veut raffiner un concept abstrait, il peut utiliser plusieurs spécialisations pour couvrir les alternatives possibles et dans ce cas la spécialisation est une opération descendante. Cependant, il peut aussi effectuer l'opération inverse qui

[1] La notation graphique sera définie au chapitre 4.

consiste à généraliser plusieurs classes en une classe représentant un concept plus abstrait (par exemple AUTOMOBILE et BATEAU sont généralisées en MOYEN_DE_TRANSPORT), c'est alors une opération de conception ascendante (cf chapitre 4, §3.1.).

Contraintes
- *Toute caractéristique exportée par A doit être exportée par B.*
- *B exporte en plus ses propres caractéristiques.*

Ces contraintes sont naturelles: puisque B est une spécialisation de A, toute caractéristique de l'interface de A, c'est à dire intervenant dans la définition du type abstrait représenté par A, doit être fournie par B, dans le sens où B est un A. Mais de nouvelles caractéristiques peuvent être exportées par B afin d'exprimer la spécificité du concept couvert par B.
L'ajout de caractéristiques peut aussi concerner les relations de communication (attribut supplémentaire). Un exemple est développé dans le chapitre 4 (domaine ORDRE).

4.2.3. Réalisation

Définition
A est réalisée par B, si B décrit une implantation particulière de A.

C'est le cas pour une classe virtuelle qui nécessite d'être "concrétisée" par un descendant.

Exemple 1

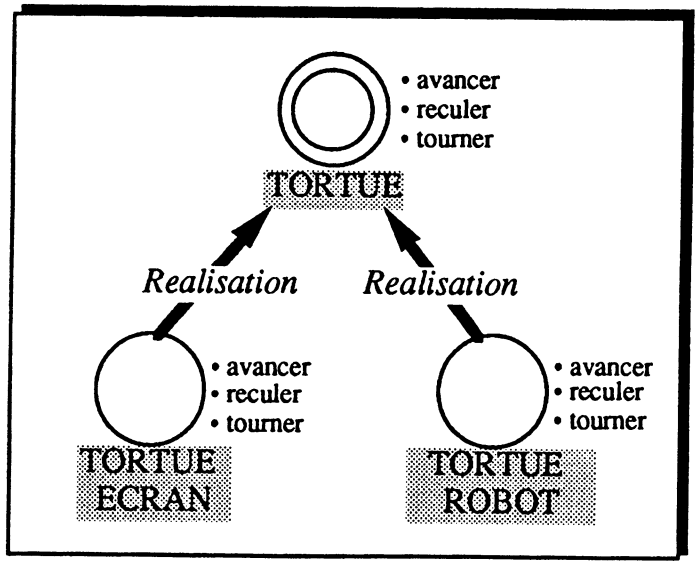


figure 8 : Exemple de réalisation

TORTUE est une classe virtuelle possédant les méthodes *avancer, reculer, tourner* permettant de diriger une tortue[1] .
TORTUE_ECRAN et TORTUE_ROBOT héritent de TORTUE, mais la réalisation des méthodes diffère.
La méthode *avancer* sera réalisée des deux façons suivantes:

```
TORTUE_ECRAN
  avancer(d):
     x2 <- pos.x + d*cos(angle)
     y2 <- pos.y + d*sin(angle)
     effacer
     si plume.estbasse alors
     tracer_ligne(pos.x,pos.y,x2,y2)
     tranlater(x2,y2);  dessiner
```

```
TORTUE_ROBOT
  avancer(d):
     nb_revolution <- d * coeff_moteur
     demarrer;  marche(nb_revolution);  arreter
```

Exemple 2

PILE_FIXE réalise le cas d'une PILE ayant un nombre fixe d'éléments. PILE est une classe virtuelle dans laquelle on ne fait que spécifier le comportement d'une pile indépendamment de son implantation, elle correspond au type abstrait. PILE_FIXE réalise PILE en utilisant des caractéristiques internes propres:

```
stockage :     TABLEAU
sommet :       INDICE
```

Les méthodes de PILE seront réalisées dans PILE_FIXE à l'aide des méthodes sur les tableaux appliquées à *stockage* et des méthodes sur les indices appliquées à *sommet*. Par exemple, on peut définir la fonction dépiler qui dépile et rend l'élément au sommet de la pile par:

```
dépiler : T
     si non vide sommet.decrementer;
     depiler <- stockage.get(sommet)[2]
```

Contrairement à la spécialisation, aucune nouvelle caractéristique ne doit apparaître dans l'interface de B puisque B est une version réalisée de A et ne doit pas raffiner le concept abstrait exprimé par A.

---

[1] Une tortue est un objet graphique ou réel se déplaçant en coordonnées polaires relatives et permettant de tracer des figures (voir la conception MECANO d'un système de gestion d'une tortue au chapitre 4).

[2] *get(i)* est une fonction sur les tableaux qui retourne le ième élément d'un tableau.

Contraintes
- *A est une classe virtuelle*
- *B est une classe concrète*
- *les interfaces de B et de A sont identiques.*

Dans ce cas, on peut dire que A est vraiment le type abstrait, c'est à dire la spécification d'un objet. Ceci permet aisément de décomposer la conception en deux phases (cf chapitre 4) :

- 1ére phase : mise en place des abstractions (classes virtuelles) et des liens d'héritages entre elles.

- 2éme phase : réalisation des abstractions, éventuellement en plusieurs versions selon les besoins. Par exemple, une PILE est réalisée au moyen d'une structure à taille fixe (PILE_FIXE), ou d'une structure dynamique (PILE_CHAINEE).

Il est possible de gagner une étape en combinant les deux héritages précédents en un seul : SPECIALISATION + REALISATION. Cependant, nous ne conseillons pas ce choix pour des raisons d'évolutivité et de réutilisabilité. En effet, le concept spécialisé se trouverait noyé dans la réalisation, et la création de nouvelles variantes de réalisation conduirait à complexifier et obscurcir la conception. D'autre part, on se priverait de la possibilité de raffiner à nouveau le concept abstrait.

4.2.4. Implantation

L'implantation est un héritage non conceptuel qui permet de réutiliser les méthodes d'une classe ("incidental inheritance" [Sakkinen 88]).

Définition
une classe B est implantée par une classe A si les caractéristiques de A ne sont utilisées que de manière interne dans B

C'est le cas, par exemple, lorsqu'on veut hériter d'une classe de librairie afin de pouvoir disposer de ses caractéristiques.
Si la classe mère est générique, l'implantation peut être associée à l'instanciation du ou des paramètres génériques[1].

Exemple

Une TORTUE_ECRAN est implantée par une figure, généralement un petit TRIANGLE isocèle, possédant des méthodes telles que *effacer*, *dessiner* et *translater*. Une TORTUE_ROBOT sera implantée par la classe ROBOT_MOBILE munie des caractéristiques *coeff_moteur*, *demarrer*, *marche* et *arrêter*.

[1] Voir l'exemple de la classe PROGRAMME au § 5.3.

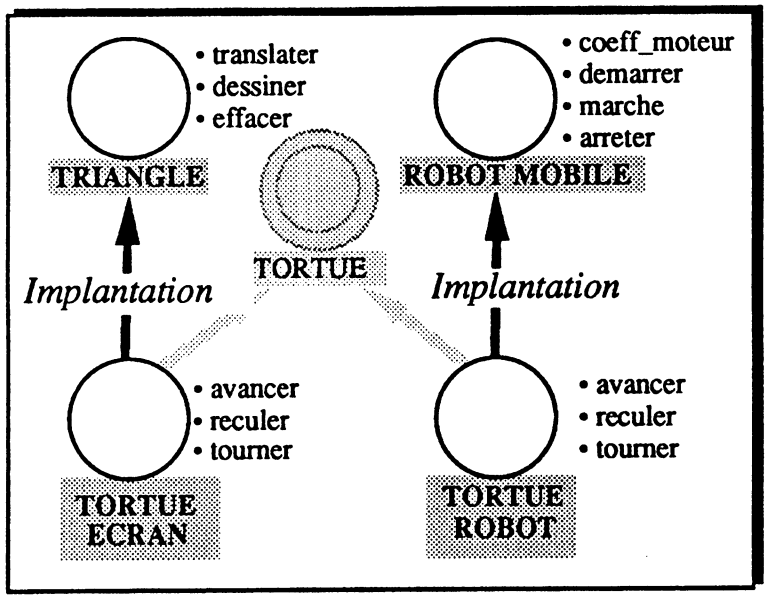


figure 9 : Exemple d'implantation

Contraintes
- *les caractéristiques de A ne sont pas exportées dans B*
- *les caractéristiques de A ne sont utilisées que pour implanter les méthodes de B*
- *A est une classe concrète*

4.2.5. Comportement

L'héritage de "comportement" (ou de "propriété") est utilisé quand le concepteur veut ajouter un comportement donné à une classe (ce que les types abstraits définissent par "propriété") .

Définition
une classe B se comporte comme une classe A si A décrit une propriété particulière que doit posséder B

Exemple
ENSEMBLE_COMP est la classe des ensembles qui peuvent être comparés entre eux. Cette classe se "comporte comme" COMPARABLE qui fournit les méthodes virtuelles *egal* et *inferieur*, et les méthodes dérivées *inferieur_ou_egal*, *superieur*, *superieur_ou_egal*, *different* réalisées en terme de *egal* et *inferieur*. En effet, l'algèbre des éléments comparables peut être basée sur les opérateurs fondamentaux d'égalité et d'infériorité. Héritant du comportement de COMPARABLE, ENSEMBLE_COMP devra (re)définir l'égalité et l'infériorité, par exemple sur la cardinalité, les autres opérateurs devenant alors immédiatement disponibles.

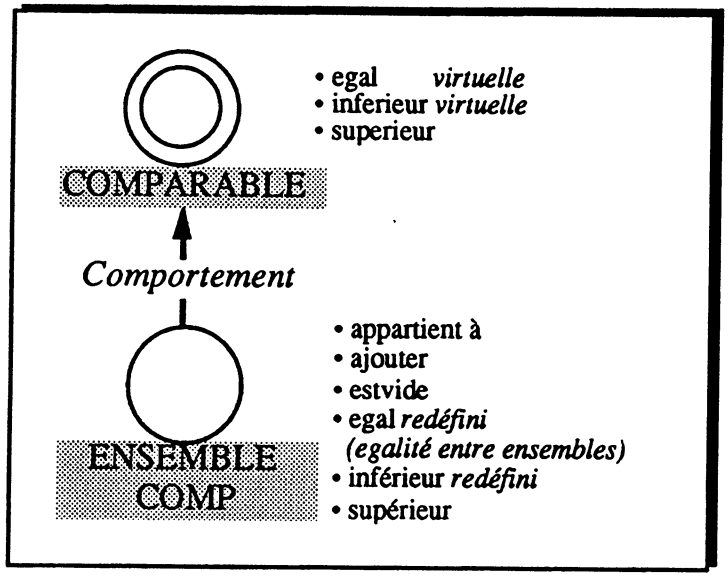


figure 10 : Exemple de comportement

Contraintes
- *A doit être virtuelle*
- *les méthodes virtuelles (opérateurs fondamentaux) de A doivent être définies dans le contexte de B*
- *l'interface de A est ajoutée à l'interface de B*
- *B peut être concrète ou virtuelle*

L'héritage de comportement est utilisé quand une propriété spéciale est nécessaire. En général, les classes de comportement sont suffixées par "ABLE": AFFICHABLE, ENREGISTRABLE, COMPARABLE, HASHABLE etc. L'héritage de comportement est souvent utilisé avec la généricité : le paramètre générique formel d'une classe est contraint par une classe de propriété CP. Le paramètre effectif devra être une classe descendante de CP, c'est à dire qu'il devra posséder la propriété définie par CP.

A la différence de la spécialisation qui est une relation "EST UN", le comportement correspond à la relation "POSSEDE LA PROPRIETE". Le concept de la classe fille n'est pas comparable à celui de la classe mère. On ne pourra pas dire qu'il est un cas particulier de l'autre. L'héritage de comportement peut être vu comme l'adjonction de possibilités à une classe.

4.2.6. Héritage combiné

La taxonomie de l'héritage est particulièrement intéressante dans des situations d'héritage combiné. En effet, les différents types d'héritage peuvent être combinés pour offrir un bon niveau d'expressivité au concepteur.
Par exemple, l'implantation et la réalisation peuvent être associées pour concevoir des classes concrètes : la classe PILE_FIXE réalise la classe virtuelle PILE au moyen d'un héritage d'implantation sur la classe TABLEAU (figure 11). De même l'héritage de comportement permet d'adjoindre une propriété à une classe et en général cet héritage vient en plus d'une spécialisation (la propriété constitue le critère de spécialisation du concept parent). Les héritages de comportement et d'implantation sont comparables: le premier sert à définir l'interface alors que le second sert à réaliser le corps de la classe héritière.

Avec la classification de l'héritage, il devient évident que l'héritage multiple est absolument nécessaire dans une conception par objets.

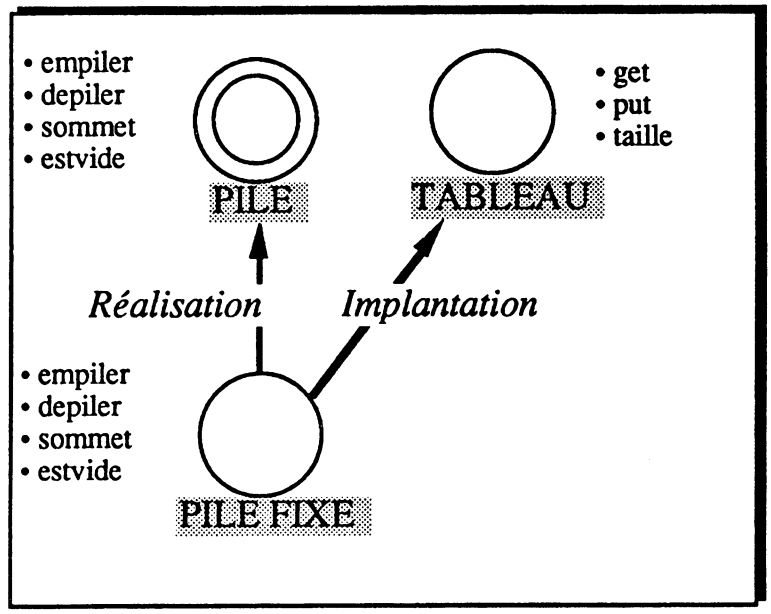


figure 11 : exemple de combinaison réalisation + implantation

Lhéritage multiple ne correspond plus à la possibilité d'avoir plusieurs parents avec la même relation, mais devient un moyen de créer des relations différentes entre une classe et ses ancêtres. C'est pour cela que nous préférons le terme d'héritage combiné plutôt que celui d'héritage multiple. L'héritage multiple sera le moyen technique permettant d'implanter l'héritage combiné de MECANO.

4.2.7. Fusion[1]

Les combinaisons d'héritage précédemment citées font apparaître des situations déséquilibrées: les liens d'héritage ne sont pas de même nature, un axe est privilégié (spécialisation ou réalisation). Dans un contexte d'héritage multiple, il se peut cependant que l'on ait besoin de liens plus équilibrés entre une classe et ses deux parentes. Dans ce cas, on dira que la classe descendante fusionne ses deux parents.

Définition
B fusionne deux classes A_1 et A_2 si les deux liens peuvent être vus comme des liens de spécialisation indépendants

Fusionner c'est créer deux liens d'héritage de même force pour définir un concept hybride. Héritage de même force signifie que B "est un" A_1 et B "est aussi un" A_2. Cela sera vrai si les caractéristiques exportées par A_1 et A_2 sont exportées par B.

Exemple
HYDRAVION est un AVION et est aussi un BATEAU: il a deux ailes et un tirant d'eau

[1] Ce type d'héritage n'a de sens que dans une situation d'héritage multiple : une classe B hérite de deux classes A1 et A2.

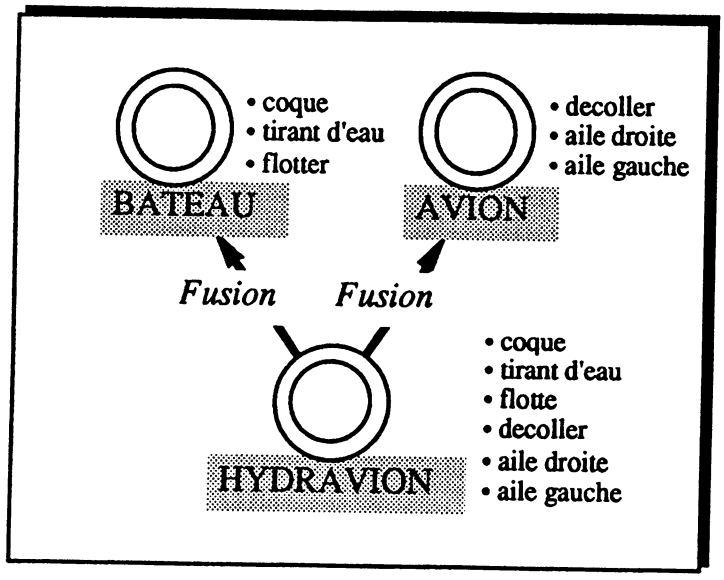


figure 12 : Exemple de fusion

Contraintes
- *les contraintes de chacune des spécialisations s'ajoutent*
- *B n'a pas de caractéristiques propres exportées*

Ce type d'héritage est assez rare. Certains auteurs, ne reconnaissant à l'héritage qu'un rôle de spécialisation, ne voient pas l'utilité de proposer l'héritage multiple puisque ce type de situation est relativement exceptionnelle. Si on adopte la taxonomie de l'héritage, cet opinion ne se justifie pas.

4.2.8. Adaptation

Un dernier type d'héritage proposé par MECANO est l'**adaptation**. L'adaptation est proche de la spécialisation mais peut intervenir sur des classes concrètes. Une adaptation se justifie quand on veut proposer une classe très proche de la classe mère mais dont certaines méthodes ou contraintes sont raffinées.

Définition
B adapte A si B raffine A sans ajouter de nouvelles caractéristiques

Dans l'exemple de la figure 13, CARRE adapte RECTANGLE. La longueur est renommée cote et le code du périmètre: 2*longueur + 2*largeur est redéfini par 4*cote.

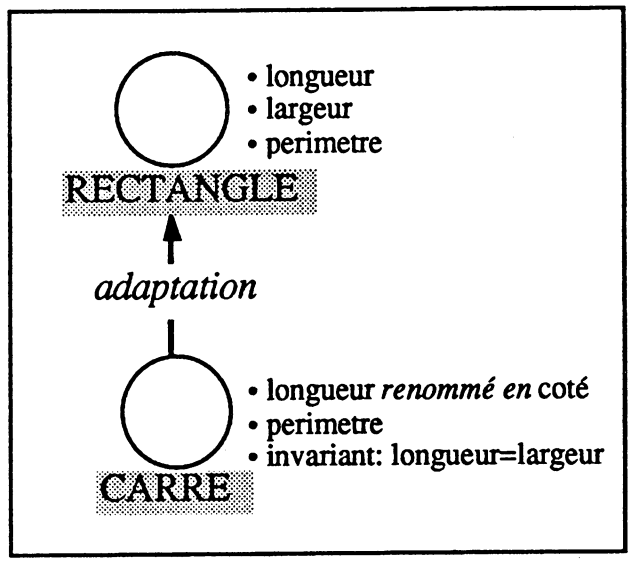


figure 13 : exemple d'adaptation

Contraintes
- *A et B peuvent être virtuelles ou concrètes*
- *aucune caractéristique n'est ajoutée*
- *B redéfinit et/ou renomme certaines d'entre elles*
- *B ajoute éventuellement des contraintes (invariant)*

4.2.9. Apports de la taxonomie de l'héritage

En premier lieu, cette classification apporte un cadre formalisé aux choix et décisions du concepteur. De cette façon, il est possible de capturer plus d'informations pendant le processus de conception et d'améliorer ainsi lisibilité et compréhensibilité.
D'un point de vue méthodologique, le raffinement de l'héritage ne peut que favoriser la compréhension de ce mécanisme complexe, en dégageant les principes d'utilisation et les contraintes induites.

La taxonomie de l'héritage, en apportant de nouveaux types de relations, enrichit le cadre conceptuel offert au concepteur.

Les types d'héritage avec leurs contraintes d'application conduisent à une conception disciplinée et peuvent être intégrés à un outil de conception assistée (CASE). Un fouineur[1] peut mettre à profit cette variété de relations inter-classe[2]. Enfin, le typage d'héritage est utilisé pour la définition du concept de Domaine de MECANO (voir ci-dessous).

---

1 Plus connu sous le terme de "browser" en anglais.

2 Voir chapitre 5.

5. STRUCTURATION DE LA CONCEPTION

5.1. Les besoins de structuration

La plupart des langages objet n'offrent pas de moyen de structuration autre que la classe. Un programme se définit comme une collection non structurée de classes reliées par des liens d'héritage et d'utilisation. Au niveau conception, cette approche apparaît insuffisante pour plusieurs raisons:

• La relation d'héritage n'est pas structurée: le graphe de la relation d'héritage d'un programme devient vite complexe surtout avec l'héritage multiple. De ce fait, la compréhension d'une classe -qui implique la compréhension de l'ensemble de ses ancêtres- devient fastidieuse et nécessite une gymnastique de l'esprit peu compatible avec les qualités de modularité et de lisibilité recherchées[1]. La méthode MECANO tire profit de la multiplicité de la relation d'héritage pour pallier à ce problème, d'une part en obligeant le concepteur à "mettre cartes sur table" en optant pour un type d'héritage bien défini, d'autre part en offrant la structure de Domaine, permettant d'isoler les parties connexes du graphe partiel d'héritage basé sur la spécialisation et la réalisation.

• Le graphe de délégation vient s'ajouter au graphe d'héritage et rend la conception encore plus ingérable. La distinction entre communication et composition améliore la lisibilité. Les structures de domaine et d'application permettront de répartir le graphe de communication dans des composants relativement indépendants.

• La notion d'objets emboîtés (composition) est très utilisée dans les méthodes de conception (cf chapitre 2). Si ce concept n'apparaît pas comme absolument nécessaire au niveau programmation, la conception ne peut s'en passer pour deux raisons:
  • cette notion permet de mettre en place des hiérarchies d'objets correspondant aux hiérarchies naturelles d'objets du monde réel.
  • le processus de conception doit s'appuyer sur des moyens de décomposition et de structuration: la composition d'objets en offre un.

5.2. Les Domaines

Spécialisation et réalisation sont les représentants "nobles" de l'héritage: la spécialisation permet de structurer les abstractions et de concevoir des classes comme des types abstraits, la conception détaillée vient ensuite en utilisant les liens de réalisation et d'implantation afin de réaliser ces types. Le domaine se propose de rassembler les classes liées à un même concept en se basant sur ces deux relations privilégiées.
Les domaines sont des macros entités contenant des classes et constituant des véritables banques de composants réutilisables. C'est l'entité structurante principale de MECANO. Le domaine permet de dégager les grands concepts d'un système et encapsule des sous-graphes connexes du graphe d'héritage général d'un système.

---

[1] Un exemple de restructuration d'une partie de la bibliothèque d'EIFFEL en MECANO est développé au chapitre 4.

5.2.1. Définition

Un domaine est un ensemble de classes connectées par des liens d'héritage. La structure d'un domaine obéit aux règles de construction suivantes:

- un domaine possède une classe racine unique appelée concept du domaine; cette classe est virtuelle et représente le concept général couvert par le domaine,
- les autres classes sont structurées en arbre issu de cette racine,
- les noeuds sont des classes virtuelles reliées par des relations de spécialisation,
- les feuilles peuvent être concrètes ou virtuelles,
- si une classe est concrète, elle doit être connectée à une classe virtuelle par une relation de réalisation, ou bien à une classe concrète par un lien d'adaptation.

Cette définition conduit à une hiérarchie à deux niveaux:
- le niveau supérieur constitué de classes virtuelles décrit le concept et ses raffinements. C'est le niveau conceptuel de spécification,
- le niveau inférieur est un niveau d'implantation, il est constitué de classes concrètes décrivant les différentes réalisations des concepts du niveau virtuel (cf figure 14).

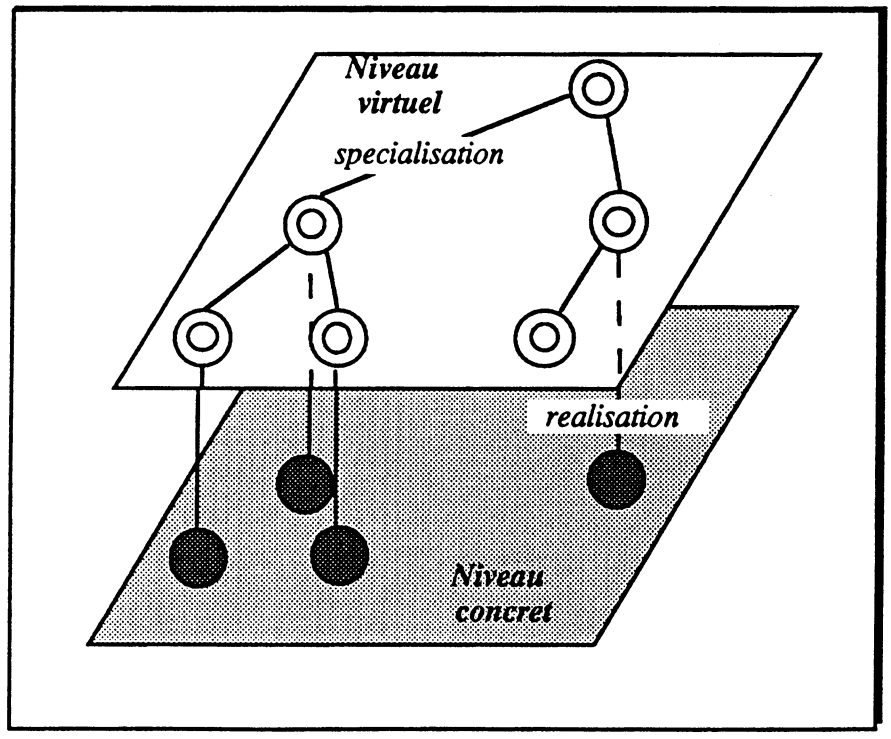


figure 14 : la structure à deux niveaux d'un domaine

Les liens d'héritage internes aux domaines sont figés : spécialisation et réalisation. Cependant, une classe interne à un domaine peut hériter de classes externes provenant d'autres domaines ou d'autres structures (cf chapitre 4). Dans ce cas, on peut utiliser d'autres types d'héritage: implantation, comportement, fusion etc...
Un domaine ne peut contenir d'autres domaines.

5.2.2. Des domaines pour quoi faire ?

Un domaine est un moyen rationnel de structuration de classes et constitue un ensemble de composants potentiellement réutilisables. Construire des domaines c'est généraliser, c'est à dire concevoir réutilisable et évolutif. Les domaines favorisent une approche "bottom-up" de la conception permettant d'inclure l'expérience logicielle d'un concepteur, d'une équipe ou d'une société pour une certaine classe d'applications.

Les domaines peuvent être appréhendés de deux manières selon que l'on conçoit **pour** réutiliser ou **en** réutilisant. La première approche est une activité de construction de domaines, tandis que l'autre concerne l'utilisation de domaines déjà créés. Ces deux activités sont généralement dissociées, soit dans le temps, soit parmi différentes personnes.

5.2.3. Construire les domaines: concevoir réutilisable

La construction des domaines est un processus à long terme qui relève de la "culture produit" dans laquelle on s'intéresse davantage aux composants qu'aux projets ("product culture" selon [Meyer 89b] [Meyer 90b]) .
Dès le début d'un projet, les concepts importants sont capturés dans des domaines. Ils peuvent ensuite être améliorés et étendus, au cours du même projet ou dans d'autres projets, afin d'obtenir des ensembles de classes cohérents et robustes couvrant au maximum toutes les facettes, versions et raffinements des concepts.
Cox imagine, de façon idéale [Cox 89], que la construction de composants réutilisables sera une activité indépendante du développement d'un projet particulier. Un "constructeur de domaine" peut passer en revue les projets anciens ou en cours de développement afin d'en extraire les connaissances sur les conceptions effectuées. Cette connaissance peut être ensuite mise en place dans des domaines appropriés par généralisation de classes spécifiques ou peut permettre d'améliorer des domaines déjà existants. De véritables banques de réutilisation, constituées de classes organisées en domaines validés et stabilisés, peuvent ainsi être développées.

Le deuxième objectif poursuivi dans la construction de domaines est la mise en place de structures supportant le polymorphisme; nous détaillons ce point dans la suite.

5.2.4. Utiliser les domaines: réutiliser la conception

L'utilisation des domaines est une activité de la "culture projet" ("project culture"). Lors du développement d'un projet spécifique, le concepteur doit en premier lieu rechercher dans les librairies de domaines, les classes appropriées à la résolution de son problème. De façon générale, cette recherche est favorisée par le fait que le paradigme objet favorise le rapprochement entre l'espace du problème et l'espace des solutions [Booch 86].
Dans le cas où la classe appropriée n'existe pas dans le domaine concerné, le concepteur peut être amené à ajouter un nouveau raffinement au concept couvert par le domaine. Il faut bien noter cependant que seules les différences et spécificités du nouveau sous-concept devront être définies et réalisées. Les autres caractéristiques demeurent accessibles du moment qu'elles n'ont pas été redéfinies.

En ce qui concerne l'utilisation des classes du domaine (relation de délégation), on distinguera deux types d'utilisation selon que le client est le créateur de l'objet ou simplement désire communiquer avec l'objet (figure 15).
Dans le premier cas, le client référencera directement le niveau concret puisqu'il doit choisir une implantation particulière du concept.

Par contre, toutes les autres classes clientes n'ont besoin que du concept abstrait. Elles utiliseront donc le concept à travers les classes virtuelles du domaine, en choisissant le degré de spécialisation voulu. Le concept est alors utilisé du point de vue abstrait, sans tenir compte de l'implantation qui sera effectivement sélectionnée; ceci est rendu possible grâce au polymorphisme et contribue à la flexibilité de la solution.

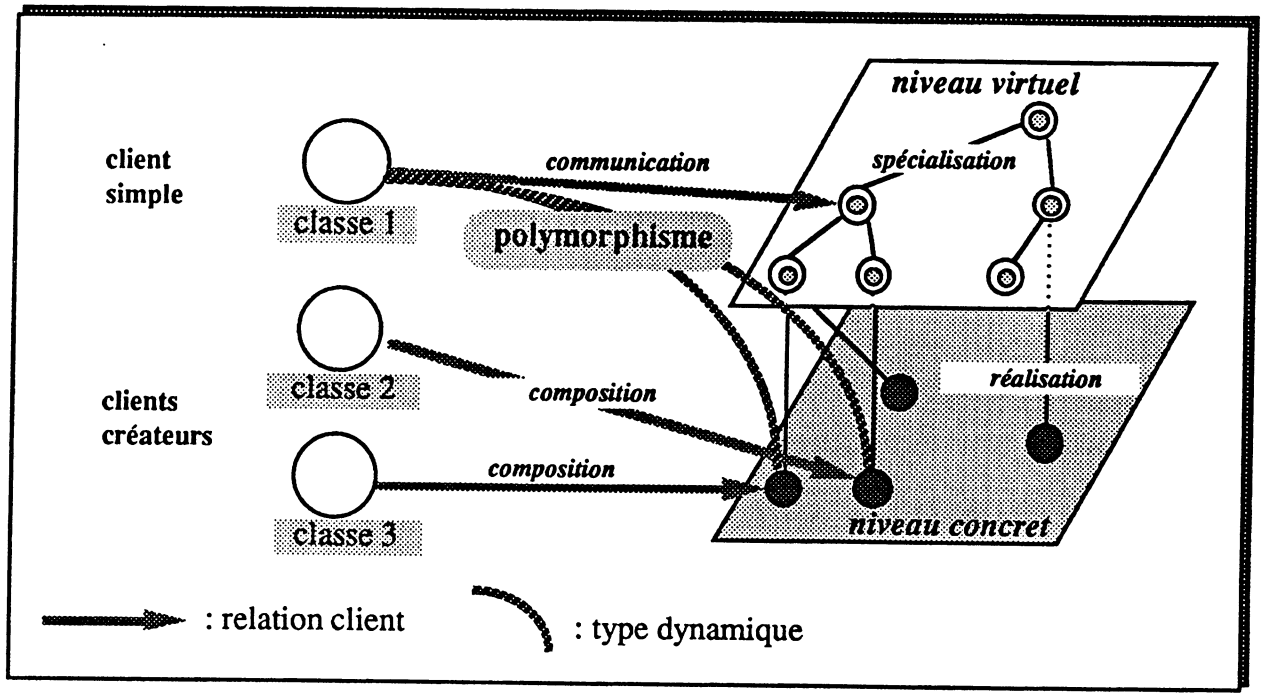


figure 15 : relation client sur les domaines

5.2.5. Le polymorphisme dans les domaines

Le polymorphisme est une notion importante dans le modèle objet. Il apporte beaucoup de souplesse et de flexibilité aux conceptions. Cette flexibilité apparaît de deux façons:

- **Polymorphisme pendant le cycle de vie :**

Lorsqu'un concept possède des versions différentes, des applications différentes peuvent être construites avec les différentes instanciations qui sont faites du concept. Par exemple, il suffit de changer le type de l'attribut *tortue: TORTUE_ECRAN* en *tortue: TORTUE_ROBOT* dans la classe racine de l'application TortueEcran pour construire un nouveau système capable de gérer une tortue réelle radio-guidée. Les autres classes ne sont pas modifiées puisqu'elles utilisent le concept de tortue au travers de la classe virtuelle racine de domaine Tortue (cf chapitre 4). Ce type de polymorphisme autorise la mise en place de versions de système et d'évolutions de façon très souple. La configuration d'un système particulier se fera dans la ou les classes chargées de créer des instances d'objets.

- **Polymorphisme pendant l'exécution:**

La flexibilité peut aussi être désirée pendant l'exécution. On parle souvent, dans ce cas, de liaison dynamique, c'est-à-dire que le code d'un appel d'opération (message) sur un objet n'est pas fixé à la compilation mais sera déterminé à l'exécution selon le type

dynamique de l'objet au moment de l'appel[1]. Par exemple, un système de gestion d'objets graphiques peut appliquer des opérations telles que *afficher*, *translater* ou *tourner* à une figure quelconque (donc à un objet de la classe FIGURE). Si cet objet possède à l'appel le type dynamique CARRE ou TRIANGLE, les fonctions éventuellement redéfinies de CARRE ou TRIANGLE seront alors appliquées; si la figure est réellement quelconque, les fonctions de FIGURE seront appliquées. On pourra ajouter de nouvelles figures sans générer d'effets de bord dans le système de gestion graphique.

Dans MECANO, le polymorphisme est associé exclusivement à la notion de domaine: une entité ne peut prendre la forme d'un de ces ancêtres que si elle est une spécialisation ou une réalisation de celui-ci (relation **est_un**). Dans les autres cas, on interdit l'utilisation du polymorphisme, car il peut conduire à la violation de contraintes sur un type abstrait. En effet, considérons une PILE_FIXE implantée par un TABLEAU: dans cet héritage d'implantation, on ne peut pas dire que la pile est un sous-concept de tableau car une pile **n'est pas** un tableau[2]. Si on considère la pile comme un tableau (polymorphisme), on pourra appliquer des fonctions propres aux tableaux sur la pile, et ainsi violer les invariants ou les axiomes fondamentaux de la pile comme par exemple accéder ou modifier un autre élément que le sommet. Ce problème est souvent exposé pour justifier l'interdiction de l'héritage d'implantation. Nous préférons le résoudre en proposant une taxonomie de l'héritage avec des contraintes d'application.

5.3. La structure composite

5.3.1. Pourquoi un nouveau concept de structuration?

Afin de permettre une certaine hiérarchisation des objets, il est important de fournir des moyens de décomposition d'un objet en sous objets composants. Les travaux sur les hiérarchies de parties en SMALLTALK [Blake 87], les objets composites de LOOPS [Bobrow 83] et de YAFOOL [Ducournau 88], les objets emboîtés de ADA [Jalote 89] et les concepts Tout/Parties des méthodes de conception (cf chapitre 2) montrent l'intérêt d'une telle approche dans le modèle objet. Nous pensons que la composition d'objets est d'autant plus nécessaire en phase de conception qu'elle apporte des facilités d'abstraction et de structuration.
Les classes peuvent être raffinées suivant des niveaux de détails de plus en plus précis, offrant de cette façon un axe de décomposition pour le processus de la conception. De plus, les objets de l'espace du problème (espace réel) sont en général des objets composés qu'il faut pouvoir représenter le plus fidèlement possible par des entités de conception dans l'espace solution (modèle objet).

MECANO offre une telle entité structurante sous le nom de "Composite".

5.3.2. Composite : une structure TOUT/PARTIE

Dans MECANO, un composite est un objet non-atomique qui peut cependant être vu comme une classe unique.
Un composite est un arbre ou un treillis de classes reliées par des liens de **composition**. Par exemple, une AUTOMOBILE est un composite avec les parties suivantes : Moteur, Roues, Portières et Sièges. Chacun des liens de composition est étiqueté par l'identificateur de l'attribut composant, ainsi une classe pourra faire partie de plusieurs

---

[1] Voir le domaine ORDRE dans l'exemple du chapitre 4 §4.

[2] Nous rappelons qu'un tel héritage, autorisé par MECANO, se justifie car le tableau constitue la structure de données principale permettant d'implanter la pile.

composites ou intervenir plusieurs fois dans le même composite (figure 16). Les composants suivent les contraintes des liens de compositions (cf §3.1.). Aucun accès à un composant n'est permis en dehors de la classe racine. Les composites peuvent être emboîtés.

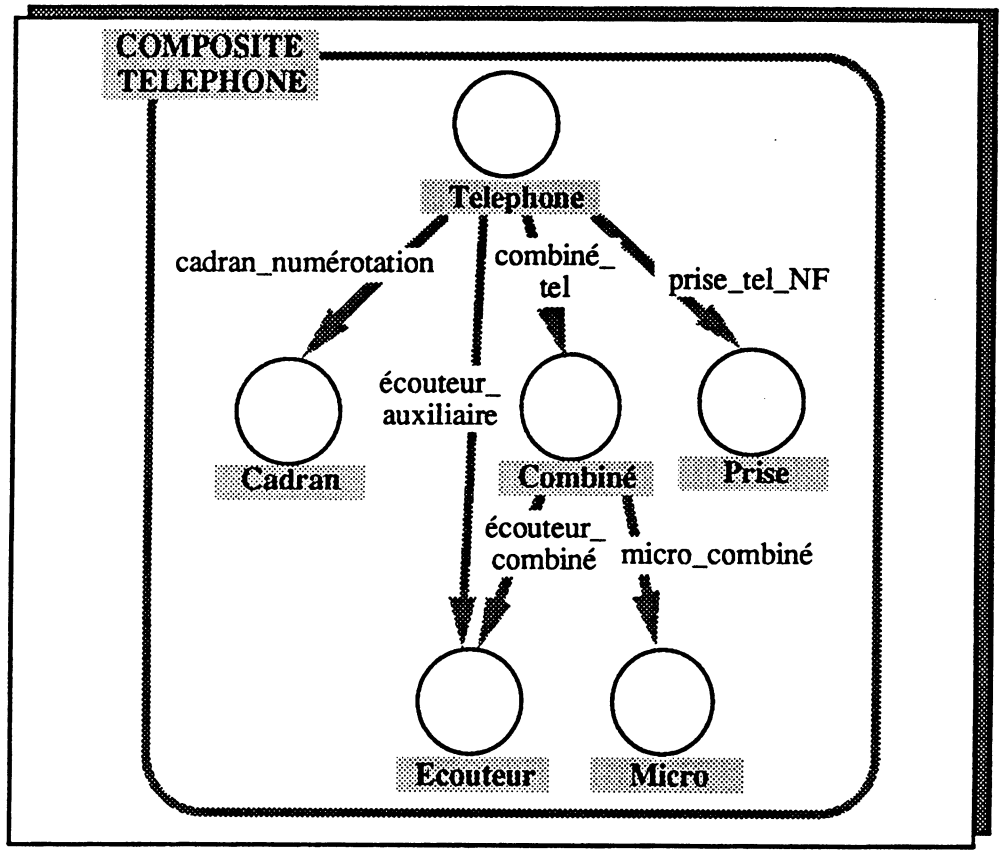


figure 16: exemple de composite

Les Composites peuvent être créés de haut en bas lorsque les objets sont décomposés en parties; ou bien de bas en haut lorsque des objets existants sont rassemblés dans un tout. Le composite, bien qu'étant une entité complexe, se comporte comme une classe et peut être utilisé à la place d'une classe. Les caractéristiques du composite sont celles de sa classe racine.

5.3.3. Composites et domaines

La composition intervient comme un axe d'abstraction orthogonal à l'héritage et aux domaines.

• Le concept de domaine offre un axe horizontal puisqu'il fournit un moyen de raffinement mono-concept : les classes d'un domaine implantent les versions et raffinements du concept défini par la racine du domaine. Un Domaine peut être réduit à sa racine par le polymorphisme: les méthodes adaptées seront exécutées selon le type dynamique (spécialisation) de l'instance. Les classes virtuelles ne seront pas instanciées.

• Le composite apporte un axe vertical d'abstraction car il permet de décomposer un concept en sous-concepts qui implantent ensemble la racine du composite. Un Composite

est vu comme une classe unique depuis l'extérieur. L'instanciation de la classe racine provoque l'instanciation des composants.

5.3.4. Composites et sous-systèmes

Le concept de sous-système tel qu'il existe dans les méthodes ObjectOry ou "Class Responsabilities" n'est pas présent dans MECANO. Nous pensons qu'un nouveau concept est inutile car les composites peuvent être utilisés pour mettre en place les sous-systèmes.
Il suffit de définir une classe racine dont l'interface correspond aux services offerts par le sous-système à l'extérieur ("Un sous-système est un groupe de classes ... travaillant ensemble pour fournir un élément bien délimité de fonctionnalités" [Wirfs-Brock 90]). Dans le corps de la classe, on répartira les différentes méthodes sur les objets composants, à la manière des objets Parent/Enfant de HOOD. Le composite encapsulera les classes internes au sous-système, les masquant ainsi pour les autres sous-systèmes. La classe racine sera chargée de la création de ses parties (classes contenues dans le sous-système). L'application finale est alors un graphe de communication sur les différents sous-systèmes (composites) vus comme des objets unitaires.
La décomposition en sous-systèmes peut se faire sur plusieurs niveaux puisque les composites peuvent être emboités.

5.4. Les applications: une structure pour le système exécutable

Pour pouvoir construire des systèmes, un concepteur doit combiner des classes avec des liens de communication et/ou d'utilisation secondaire pour obtenir une application correspondant à une version finale d'un projet particulier. Une application est un graphe connexe de classes et de composites dont les arcs sont des relations clients. Des liens additionnels d'héritage (implantation, fusion, comportement) peuvent être utilisés si nécessaire. Les applications sont construites en utilisant des classes provenant des domaines ou bien en créant de nouvelles classes. Une classe peut appartenir à plusieurs applications différentes.

Une application doit posséder une racine (classe n'ayant aucun client) qui sera instanciée au lancement du système. En général, la ou les applications sont mises en place à la fin de la phase de conception détaillée (cf chapitre 4). Ceci permet de retarder le plus possible les choix sur la structure de contrôle et l'interface externe du système, parties les plus exposées aux modifications et évolutions dans un logiciel [Meyer 88].

Les applications permettent de décrire l'aspect exécutable du projet en cours de développement. MECANO offre la possibilité d'avoir plusieurs applications par projets permettant de décrire des versions du projet ou des variantes d'applications proches utilisant les mêmes domaines. Dans le cas de gros logiciels, on peut aussi définir des applications pour encapsuler des sous-systèmes.

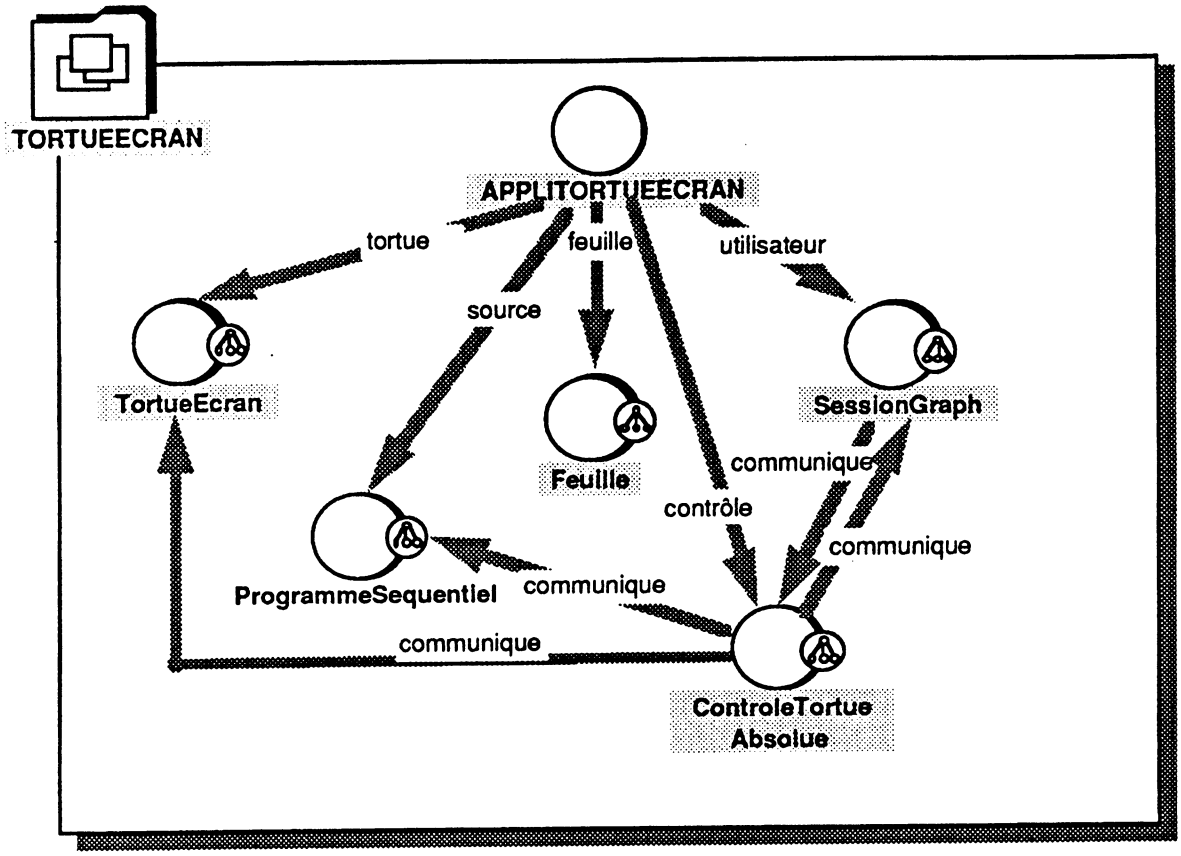


figure 17: exemple d'application

6. CONCLUSION

La méthode MECANO aborde la conception par objet en utilisant et en adaptant les concepts de la programmation orienté-objet. Un raffinement de la sémantique des relations inter-classes permet d'améliorer le modèle et conduit à une conception plus claire par un enrichissement des outils mis à la disposition du concepteur. Ces relations fournissent une base pour la définition de nouvelles structures chargées de l'organisation des classes.

La distinction entre la relation de communication et la relation de composition permet de dissocier les collaborations des agrégations.
La taxonomie de l'héritage assure une utilisation rigoureuse de ce concept et favorise la compréhension des graphes d'héritage. De ce fait, l'héritage multiple sort de sa confidentialité et devient nécessaire à l'élaboration des conceptions (héritage combiné).
Les Domaines offrent une structure pour exprimer les hiérarchies de spécialisation/réalisation et encapsulent les classes reliées à un même concept.
Les Composites donnent un moyen d'abstraction supplémentaire basé sur la relation structurante de composition.
Les différents systèmes opérationnels sont conçus au moyen des Applications. Les objets d'une application sont considérés comme des agents communiquants.

Qu'il s'agisse des Modules de OMT, des objets Parent de HOOD, des Sujets de OOA, des Sous-systèmes de Class Responsabilities, des Blocs de ObjectOry ou des Clusters de OOAD, la sémantique associée aux entités structurantes est généralement pauvre. Il s'agit, le plus

souvent, de découper l'ensemble des classes en parties, mais le critère de cette nouvelle modularité est souvent mal formalisé. Notre méthode offre trois types d'entités structurantes possédant chacune des règles de construction bien établies et assurant la structuration du logiciel sur trois axes complémentaires.

Ce chapitre nous a permis d'exposer les concepts de la méthode, c'est à dire le modèle objet utilisé. Nous allons dans la partie suivante nous intéresser aux deux autres composants de la méthode: le formalisme et le processus de conception.
Le formalisme donne une notation pour l'expression des concepts. Nous verrons que MECANO a choisi une notation principalement graphique.
Le processus de conception est "l'algorithme" de la méthode. Il indique les différentes étapes à accomplir et donne des indications pour la mise en oeuvre des concepts.

CHAPITRE 4:

CONCEVOIR AVEC MECANO

Ne commencez pas par demander CE QUE fait le système
Demandez A QUOI il le fait [Meyer 88]

PROCESSUS DE CONCEPTION ET EXEMPLE

1. INTRODUCTION

Dans ce chapitre, nous présentons la mise en oeuvre de la méthode MECANO. Il s'agit de définir la notation utilisée pour exprimer les concepts définis au chapitre précédent. Nous proposons ensuite un processus de conception, décrivant les étapes à accomplir pour construire un système avec MECANO.
Ce processus est illustré par un exemple de conception. Cet exemple, de taille moyenne, nous permettra de concrétiser la mise en oeuvre des concepts et de la notation associés à la méthode. A travers des exemples de modifications et d'évolutions du système obtenu, nous montrons comment la conception peut s'adapter de façon aisée à ces changements.
La dernière partie traite de l'implantation d'une conception MECANO dans un langage de programmation et du problème de la restructuration de programmes déjà développés ("reverse engineering").

2. FORMALISME ET ENTITES DE CONCEPTION

Une conception MECANO est constituée d'entités mises en place progressivement par le concepteur. Nous précisons ici les différents types d'entités et les niveaux d'abstractions qu'elles engendrent selon la relation d'appartenance. Des éléments du formalisme associé à la méthode ont déjà été introduits dans le chapitre précédent, nous les définissons ici de façon exhaustive.

2.1. Le projet : ensemble cohérent d'entités de conception

Le projet est défini comme entité racine de toute conception. Un projet MECANO correspond à un ensemble de données de conception relatives à un même contexte d'application logicielle. Le projet est constitué d'applications et de domaines. Il est la racine de l'arbre d'appartenance représentant une conception MECANO (figure 2). Un certain nombre d'informations complémentaires peuvent être associées à un projet (voir chapitre 5).

2.2. Entités complexes et entités basiques

Les entités de conception sont réparties en deux groupes: les entités complexes et les entités basiques. Une entité complexe est décomposable en sous-entités. Une entité basique ne contient pas d'entités. La décomposition d'entités se fait selon la relation d'appartenance (figure 2). Les entités complexes sont le Projet, les Domaines, les Applications et les Composites, elles sont représentées graphiquement.
La seule entité basique de MECANO est la Classe. Une classe est définie à partir de rubriques textuelles: spécification, méthodes et attributs de communication, de composition et d'utilisation secondaire (cf §2.6.).

Les entités complexes et basiques sont symbolisées dans la représentation graphique de leur entité englobante par des pictogrammes (figure 1).

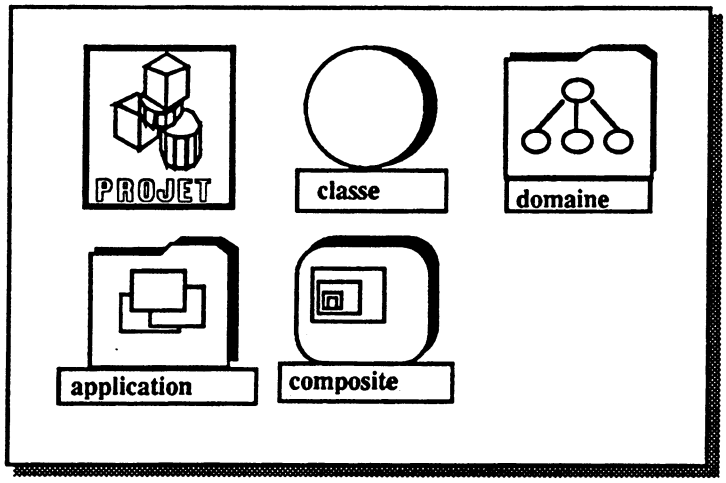


figure 1 : représentation graphique des entités MECANO

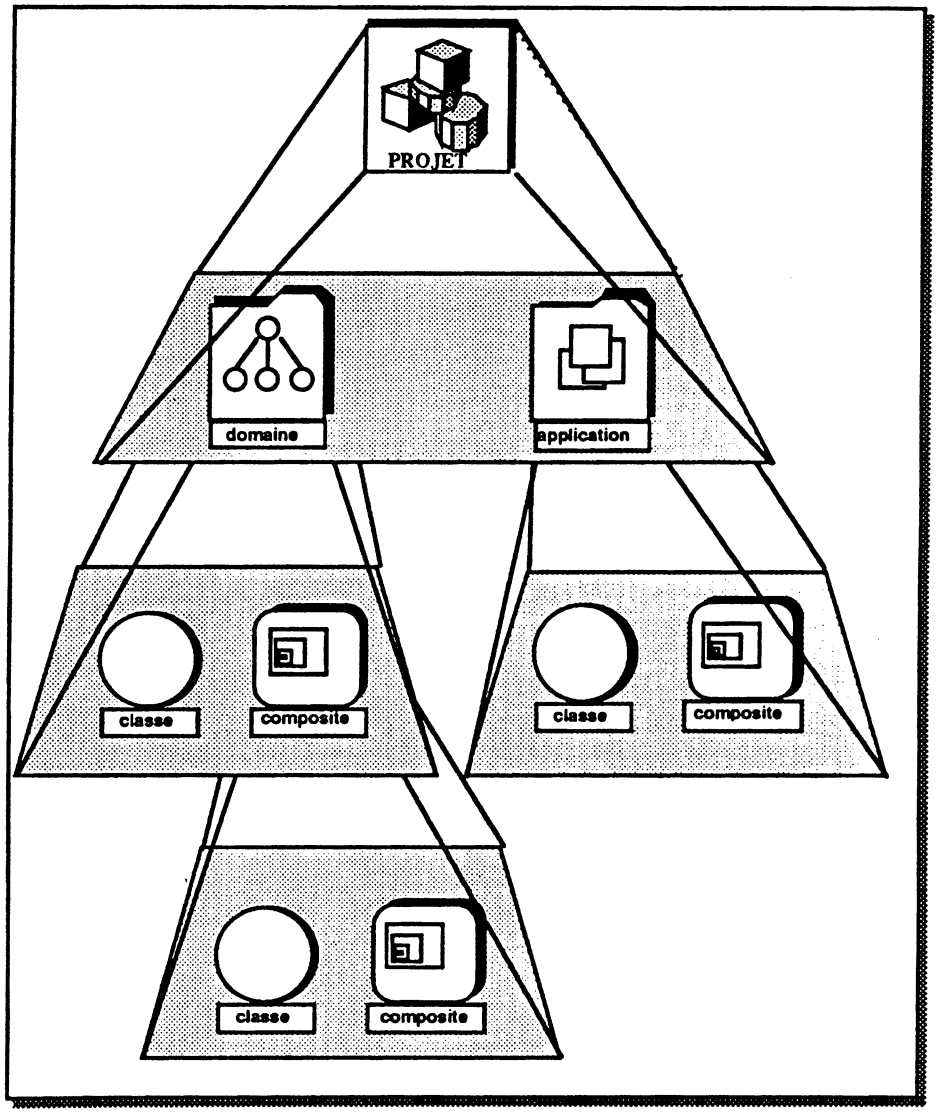


figure 2 : relation d'appartenance sur les entités MECANO

Les pictogrammes de classe ont des formalismes différents selon que la classe est virtuelle ou concrète, générique ou non générique. La figure 3 montre ces différentes représentations. Les pictogrammes de classe apparaîtront comme composants des entités complexes.

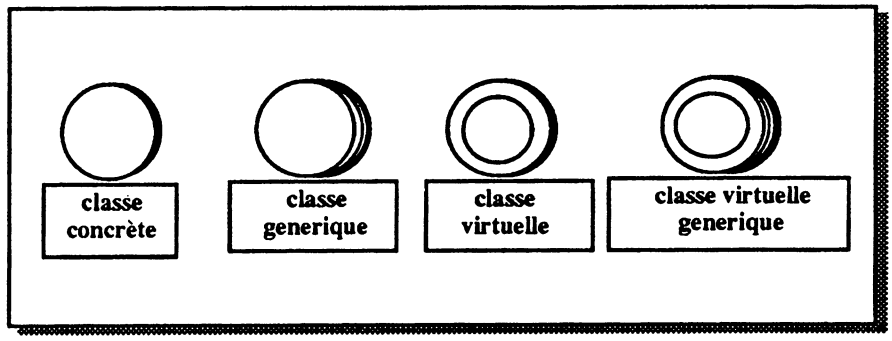


figure 3 : types de classes

2.3. Relations entre entités

Les entités à l'intérieur des applications, des domaines et des composites sont connectées par des relations orientées. Les deux grandes catégories de relation se distinguent par leur formalisme: héritage (flèche noire) et délégation (flèche large grisée). Le type de la relation utilisée est ensuite défini par une étiquette sur la flèche correspondante.
Dans le cas d'une relation de délégation, les termes utilisés sont *communique* pour la relation de communication et *utilise* pour la relation de délégation secondaire. La relation de composition est identifiée par une étiquette correspondant au nom de l'attribut concerné (composant). Ceci permet d'avoir plusieurs liens sur la même classe correspondant à des couples {classe cliente, étiquette} différents. Une autre raison à ce choix est le statut particulier de la relation de composition et les contraintes fortes qu'elle exige: l'attribut ne joue pas le simple rôle de référence à un objet quelconque de la classe destination, mais assure que l'objet référencé, une fois créé, fait partie intégrante de l'objet créateur et ne peut être remplacé par un autre objet.
Corollairement, MECANO interdit les mots clés *communique* et *utilise* comme attribut de composition.
Les relations d'héritage sont étiquetées par les types de la taxonomie (cf chapitre 3).

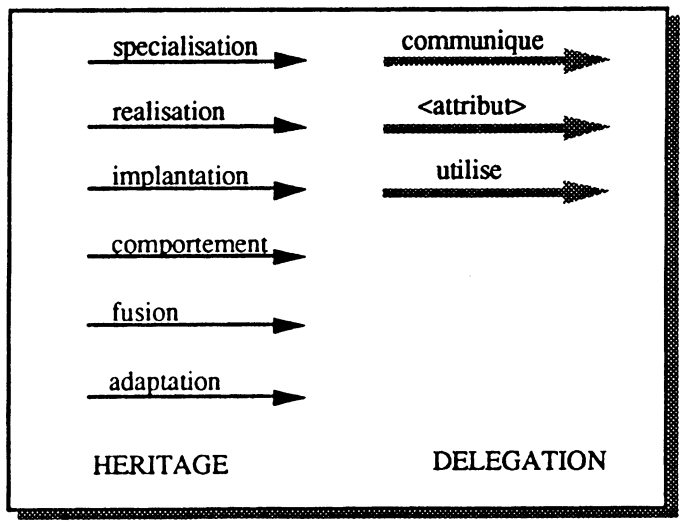


figure 4 : représentation des relations

2.4. Instanciation de paramètre générique

2.4.1. Utilisation d'une classe générique

Dans le cas de l'utilisation d'une classe générique avec instanciation du ou des paramètres génériques, on autorise l'établissement d'un lien de délégation entre la classe utilisatrice de la classe générique et la classe dont est issu le paramètre effectif. Ce mécanisme permet de clarifier la conception en masquant un niveau de paramétrisation, surtout pour les structures de données génériques très souvent utilisées. On exprime ainsi directement une relation 1-n en précisant la structure de données (liste, tableau, ensemble ...)[1].

L'intérêt de cette approche est qu'elle permet d'exprimer graphiquement les informations pertinentes au niveau d'abstraction voulu.
Dans l'exemple suivant,

```
classe GARAGE
attribut de composition
     parking : LISTE(AUTOMOBILE)
     ...
fin classe GARAGE
```

l'information pertinente à modéliser est la relation entre Garage et Automobile, "un garage est constitué d'un parking contenant des automobiles" et non la relation entre Garage et Liste "un garage utilise une liste d'éléments" (dont le type sera défini dans la conception détaillée de la classe Garage). On assimile donc l'utilisation d'une classe générique à l'utilisation directe de la ou les classes correspondant aux paramètres effectifs. Le schéma 1 de la figure 5 montre la solution classique. La solution du schéma 2 établit un lien d'utilisation sur le paramètre effectif de LISTE.

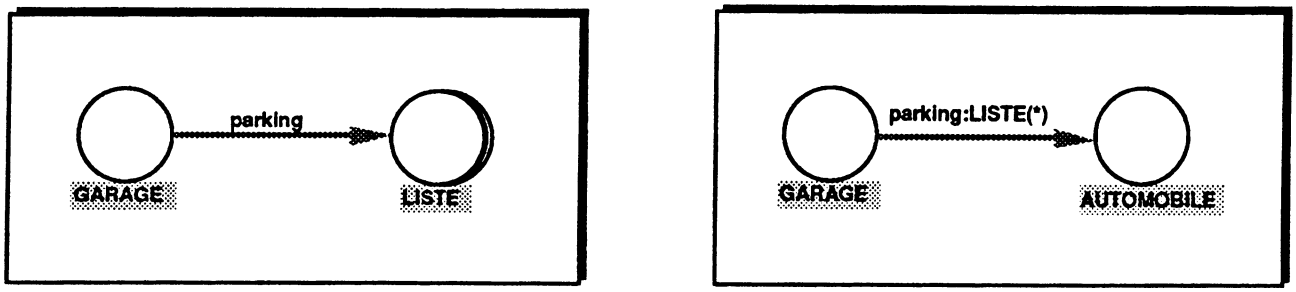


Schéma 1 Schéma 2

figure 5: instanciation de paramètre générique

MECANO permet aussi dans la solution classique d'ajouter le type générique effectif directement sur l'étiquette (figure 6). Le choix entre les deux solutions (schéma 2 ou 3) sera guidé par le degré d'abstraction de l'information à représenter. Dans l'exemple suivant, on privilégie la classe GRAPHE comme classe composante d'une CARTE ROUTIERE, on indique le paramètre générique VILLE sur le lien de composition.

---

[1] La généricité, avec cette notation, nous semble plus riche et plus homogène que les constructeurs de listes et d'ensembles tels qu'ils sont proposés par exemple dans OMT [Rumbaugh 91].

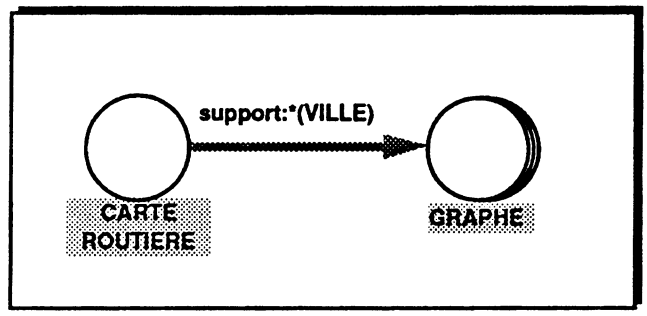


figure 6
instanciation de paramètre générique: schéma 3

Dans les deux cas, le symbole "*" correspond à la classe destination de la relation.

2.4.2. Héritage d'une classe générique

Pour être homogène, l'héritage de classe générique avec instanciation est traité de la même façon. Dans ce cas, seul le schéma 3 est autorisé, le schéma 2 n'ayant pas de sens dans ce contexte (une classe n'hérite pas du paramètre générique de sa classe mère).

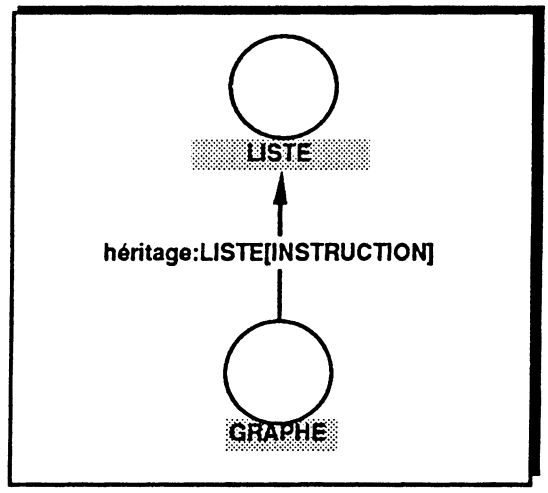


figure 7: héritage de classe générique

2.4.3. Syntaxe de l'étiquette

La formulation de l'étiquette, dans le cas de l'utilisation d'une classe générique avec instanciation des paramètres, répond à la syntaxe suivante:

<label> ::= <type rel>:<classe generique>(<e1>,<e2>,...,<en>)

avec

- <label> : nom de l'étiquette
- <type rel> : *communique, utilise*, nom de l'attribut de composition ou type d'héritage
- <classe generique> : nom de la classe générique utilisée, si le caractère * est utilisé, la classe générique est la destination de la relation,
- n : nombre de paramètres génériques de la classe

- <ei> = "*" : convention pour exprimer à quel paramètre correspond la classe destination ($i^{ème}$ paramètre générique)
- <ei> = nom de classe : le $i^{ème}$ paramètre effectif est <ei>
- <ei> = "-" : le $i^{ème}$ paramètre n'est pas exprimé sur ce lien

2.4.4. Instanciation de plusieurs paramètres génériques

Dans le cas d'une classe générique à plusieurs paramètres, on peut rencontrer les situations des schémas 1 et 2 de la figure 8. Dans ces deux exemples, on considère la classe générique des double-listes définies par DB_LISTE(T1,T2) qui permet de gérer des listes de paires d'éléments de types quelconques T1 et T2. On peut alors concevoir la classe des polynômes à une variable (schéma 1) utilisant une liste de paires d'entiers pour représenter ses monômes. Dans ce cas, un seul lien de composition est utilisé en instanciant simultanément les deux paramètres effectifs (ENTIER).

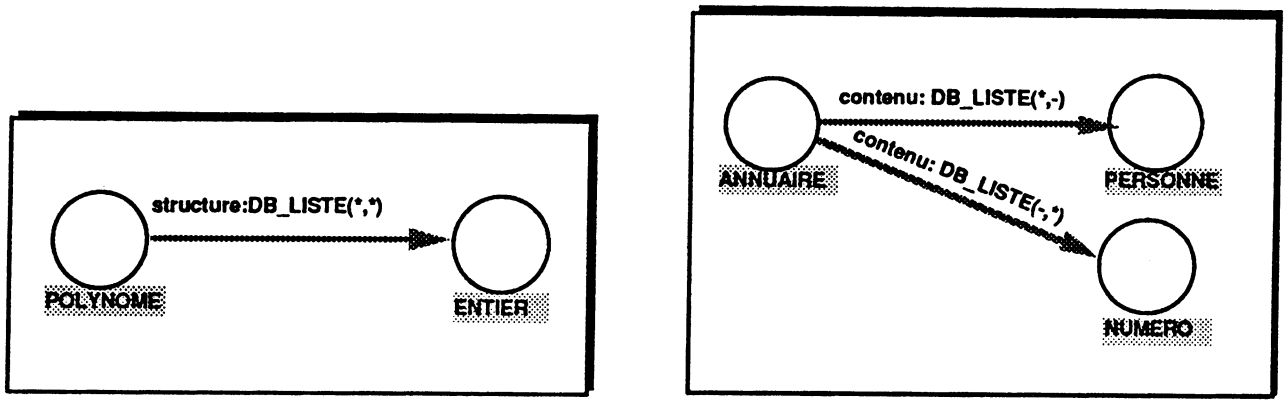


schéma 1 schéma 2

figure 8 : instanciation de plusieurs paramètres génériques

Dans le schéma 2, on définit la classe ANNUAIRE qui gère des couples (personne, numéro de téléphone) contenus dans une liste double. Dans ce cas, deux liens de clientèle sont utilisés pour un seul attribut. Chacun des liens correspond à l'instanciation d'un paramètre différent.

2.4.5. Contraintes

Les contraintes associées à la relation client avec instanciation de paramètres génériques sont les suivantes:

- le caractère "*" remplace toujours la classe destination de la relation
- la classe spécifiée dans l'étiquette doit être générique
- le nombre de paramètres présents dans l'étiquette doit être le même que le nombre de paramètres formels de la classe
- si caractère * remplace la classe destination, il ne peut remplacer la classe générique
- si caractère * remplace la classe destination, il peut être répété (voir figure 8, schéma 1)
- pour une classe donnée, il peut y avoir plusieurs utilisations avec le même attribut en étiquette[1] (voir figure 8, schéma 2)
  Dans ce cas les listes de paramètres doivent être compatibles, c'est à dire:

[1] C'est le seul cas où deux liens de composition peuvent porter le même attribut.

- la classe générique fournisseur est identique
- le caractère "*" ne peut apparaître à la même place dans les deux étiquettes
- les notations "-", "*" et <nom de classe> pour les paramètres doivent être compatibles:
  - si le ième paramètre est "-" dans une relation,
    alors le ième paramètre est soit "-" soit "*" dans les autres relations.
  - si le ième paramètre est un nom de classe,
    le même nom est présent dans toutes les autres relations.

2.5. Entités internes et entités externes

2.5.1. Entité interne

Une entité est interne à une entité complexe englobante si elle est un constituant de celle-ci, c'est à dire si le concepteur l'a créée depuis cette entité. Une entité interne respecte les règles de constitution de son entité complexe englobante. Ces règles sont les suivantes (cf chapitre 3):

Règles de constitution des entités complexes pour les entités internes

1) Une conception MECANO est un arbre d'entités complexes et basique,s construit sur la relation d'appartenance dont la racine est le projet.

2) Un PROJET est composé de domaines et d'applications sans relations.

3) Une APPLICATION est constituée de classes et de composites reliés par des relations de communication, composition et utilisation secondaire.

4) Un DOMAINE est constitué de classes et de composites reliés par des liens d'héritage de réalisation, de spécialisation et d'adaptation.

5) Un COMPOSITE est constitué de classes et de composites reliés par des relations de composition.

2.5.2. Entité externe

Une entité complexe peut utiliser des entités externes (provenant d'autres entités complexes) pour sa propre conception[1]. Ce sont des entités internes à d'autres entités complexes mais qui fournissent un service (fournisseur ou classe mère) aux constituants de l'entité complexe courante.
Les entités externes représentent des liens entre les entités complexes: si une entité complexe EC1 utilise une entité externe Eext provenant d'une autre entité complexe EC2, on dira par abus de langage que EC1 est "cliente" de EC2. Cette notion de client est différente de la relation client/fournisseur entre deux classes puisque la relation entre EC1 et EC2 correspond à un véritable achat de service qui intervient aussi bien à travers

[1] Dans l'exemple du §4., la classe TortueEcran est une sous-classe (implantation) de la classe Triangle, extérieure au domaine TORTUE.

une relation inter-classes de délégation que d'héritage[1]. C'est de cette façon que sera mise en oeuvre la réutilisation de composant: une classe interne à EC1 hérite ou est cliente d'une classe d'une autre entité complexe EC2 provenant d'un autre projet ou d'une bibliothèque.

Du point de vue du formalisme, les entités externes se différencient des entités internes par un "plot" d'appartenance indiquant la provenance de l'entité (figure 9) Un plot est une mini icône symbolisant le contexte (type de l'entité complexe) dans lequel est définie l'entité externe.

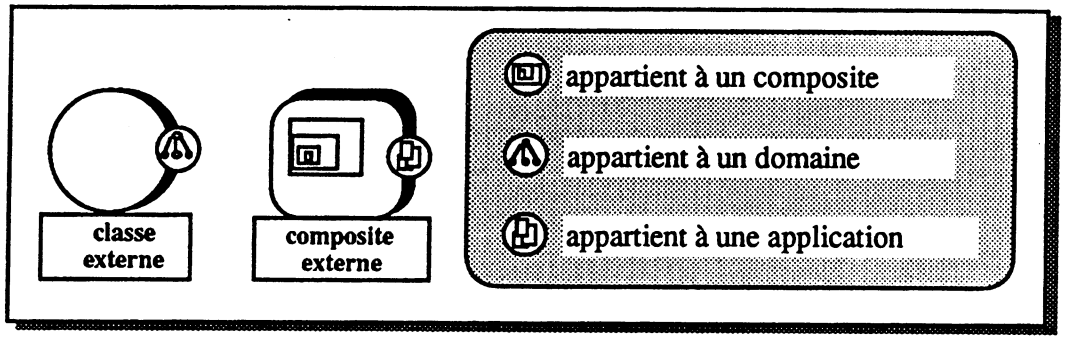


figure 9: entités externes et plots d'appartenance

2.5.3. Règles sur les relations entre entités internes et externes

Une entité externe à une entité complexe doit vérifier les contraintes suivantes:

Contraintes d'utilisation des entités externes

Eext: entité externe
EC1: entité complexe utilisatrice de Eext

1) Eext est interne à une autre entité EC2.
2) Eext est reliée directement à une entité interne Eint de EC1 par un lien client ou une relation d'héritage.
3) la relation entre Eext et Eint, qu'elle soit de délégation ou d'héritage, est dans le sens Eint -> Eext.

• La contrainte 1 assure qu'une entité externe doit être créée d'abord dans son contexte d'origine (entité englobante)[2].
• La contrainte 2 assure que les entités externes sont bien aux "frontières" des entités complexes.
• La contrainte 3 assure que les entités externes sont bien des fournisseurs de services (par délégation ou héritage) aux entités internes. De cette façon, la mise en place de relations entre une entité interne et une entité externe n'a pas d'effet sur l'externe. Détaillons les cas possibles:

---

[1] Le terme d'achat peut même recouvrir une certaine réalité économique dans le cas de l'utilisation de base de composants commerciale.

[2] Nous verrons au chapitre que cette contrainte pourra être levée par le poste de conception, permettant ainsi, temporairement, d'avoir des références à des classes non encore définies. L'outil sera capable, sur demande, de fournir la liste de telles classes afin d'assurer la complétude de la conception.

1) Si Eint hérite de Eext:
Eint possède les caractéristiques de Eext, mais Eext qui ne connaît pas ses descendants n'est pas concernée par cette nouvelle relation
2) Si Eint est client de Eext:
Eint possède un moyen d'accès à des instances de Eext mais Eext n'a pas connaissance statiquement de ses clients.

Dans les deux cas la règle concernant le sens de la relation (Eint -> Eext) permet d'assurer l'intégrité de l'entité externe qui ne doit effectivement pas être impactée par cette relation.

De plus, toutes les entités, qu'elles soient internes ou externes, vérifient toujours les contraintes sur les différents types de relation (cf chapitre 3).

2.5.4. Relations entre entités externes

Que se passe-t-il si le concepteur a besoin d'une relation entre deux entités externes? C'est le cas dans la situation suivante:

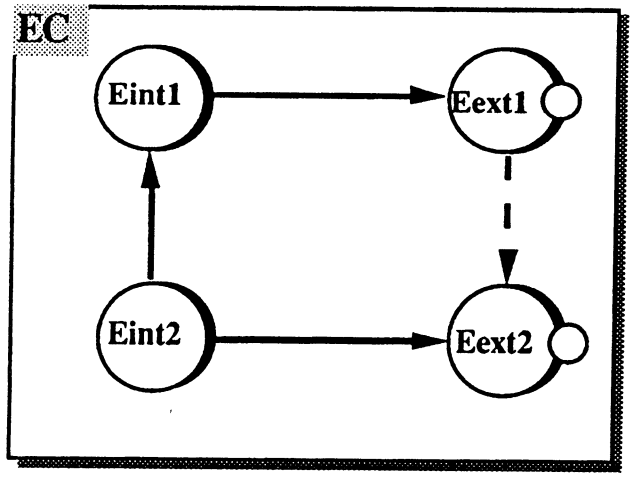


figure 10: relation entre entités externes

Cette relation ne sera pas considérée par MECANO comme locale à l'entité complexe EC puisqu'elle ne met en jeu aucune se ses classes internes. il faut donc établir cette relation dans le contexte de l'entité englobante EC' de la classe source (Eext1). Dans le contexte EC', Eext1 est interne et Eext2 est reliée à Eext1 dans le sens Eext1 -> Eext2[1].

2.6. Définition d'une classe

La classe est la seule entité de base de MECANO. Une classe n'est pas décomposable graphiquement, elle est constituée de **rubriques** textuelles. Chaque rubrique est composée d'une liste d'informations élémentaires. Nous détaillons ici la structure des différentes informations manipulées.

Six rubriques sont définies: **spécification**, **généricité**, **interface**, **renommage**, **méthode** et **attribut**. Une rubrique est généralement constituée d'une liste d'éléments de types différents ou de même type. Un élément est une information structurée

[1] Le poste de conception permet de visualiser des relations entre externes. Ces relations ne sont pas modifiables.

élémentaire. Pour chaque rubrique, nous définissons la structure des éléments intervenant dans celle-ci.

Remarque: la structure est définie par une notation de type grammaire avec les conventions habituelles:

/	=	*alternative*
[]	=	*facultatif*
*	=	*répété 0 ou plusieurs fois*
mot en gras	=	*mot-clé ou caractère-clé*
mot entre <>	=	*information terminale (type simple ou identificateur)*

2.6.1. Rubrique SPECIFICATION

La rubrique spécification contient des informations générales sur la classe: type, commentaire sur le rôle, auteur, date de création, axiomes (invariants) vérifiés par la classe. La spécification est une liste d'éléments prédéfinis, certains sont non éditables:

virtuelle : <booléen>
  indique si la classe est virtuelle ou concrète
générique : <booléen>
  indique si la classe est générique ou non
rôle : <texte>
  commentaire informel explicitant le rôle de la classe
mot-clé : <chaîne> [, <chaîne>]*
  liste des mots clés concernant la classe
auteur : <chaîne>
  nom de l'auteur
état: <chaîne>
  état de la classe dans la conception (incomplète, testée, officiel...)
création: <date>
  date de création de la classe
invariant: <expression booléenne>
  axiomes vérifiés par les objets de la classe

2.6.2. Rubrique GENERICITE

La rubrique généricité est facultative. Elle contient deux informations distinctes concernant la définition ou l'utilisation de classe générique:
1) la liste des paramètres génériques formels et les contraintes éventuelles, si la classe est générique,
2) la liste des paramètres génériques effectifs associés au nom de la classe générique dans le cas d'un héritage ou d'une utilisation de classe générique avec instanciation.
Un élément de cette liste sera de l'une des formes suivantes:

<parametre_generique_formel> [**contrainte** <classe_de_contrainte>]
herite <super classe> (<liste_parametre_generique_effectif>)
utilise <super classe> (<liste_parametre_generique_effectif>)

Remarque: cette rubrique doit être compatible avec le formalisme graphique (étiquette sur des liens d'héritage ou de délégation avec des classes génériques - cf §2.4.).

2.6.3. Rubrique INTERFACE

La rubrique interface contient la liste des caractéristiques exportées par la classe. Les attributs peuvent aussi être exportés, ils se comportent dans ce cas comme des fonctions sans argument (lecture seulement). Un élément exporté est décrit par son nom, la liste des paramètres formels typés et éventuellement le type du résultat[1]:

```
<nom_de_méthode> ( <parametre> : <type> [, <parametre>: <type>]* ) [: <type>] ou
<nom_d'attribut> : type
```

2.6.4. Rubrique RENOMMAGE

La rubrique renommage contient la liste des méthodes et des attributs renommés avec la classe parente correspondante et le nouveau nom associé. Un élément de cette liste est du type:

```
<nom_de_caractéristique> de <nom_classe_parente>
    renommé <nouveau_nom_de_caractéristique>
```

2.6.5. Rubrique METHODE

La rubrique méthode contient la liste des définitions de méthodes (exportées et internes). Un élément de cette liste est défini par:

```
(action <nom_de_méthode> ( <parametre> : <type> [, <parametre>: <type>]* )
 | fonction <nom_de_méthode> ( <parametre> : <type> [, <parametre>: <type>]* )
   [: <type>] )
[retardé | défini | redéfini]
spécification <texte>
[précondition <expression booléenne>]
[postcondition <expression booléenne>]
[local <variable> : <type> [, <variable>: <type>]* ]
[définition <texte>]
```

Le mot clé *retardé* indique s'il s'agit d'une méthode virtuelle non définie. *Redéfini* indique que le corps qui suit constitue une redéfinition d'une méthode héritée. *Défini* indique la définition d'une méthode héritée retardée. La *spécification* est un commentaire informel décrivant la méthode. Le champ *local* permet de définir des variables locales. Viennent ensuite les pré- et post-condition définies par une expression booléenne dont certaines parties peuvent être informelles. La *définition* est le corps de la méthode qui peut être rédigé en langue naturelle, en langage algorithmique ou bien directement dans un langage orienté-objet.

2.6.6. Rubrique ATTRIBUT

La rubrique attribut contient la liste des attributs typés de la classe. Un élément de cette liste a la structure suivante:

```
(communication | composition | utilitaire) <nom_d'attribut> : <type>
[défini | redéfini]
spécification <texte>
```

[1] Nous préférons cette notation à celle d'EIFFEL dans laquelle seuls les noms de méthodes apparaissent.

Le mot clé précédant le nom de l'attribut précise le genre de la relation de délégation. Le type (classe fournisseur) de l'attribut doit être fourni ainsi que le mot clé *redéfini* s'il s'agit d'une redéfinition (le type doit être un descendant du type initial). Le mot clé *défini* est réservé au fonctions retardées réalisées par un attribut. La spécification précise le rôle de l'attribut dans la classe.

Nota: des exemples de classes, projets, domaines, applications et composites seront exposés dans l'exemple du §4.

3. PROCESSUS DE CONCEPTION

Une méthode doit permettre de décrire un **état** de la conception à un instant donné, c'est à dire qu'elle doit prendre en compte l'aspect statique de la conception [Girod 87a]. Cet aspect a été décrit dans le chapitre 3 à travers la définition des concepts et l'introduction d'un formalisme (graphique et textuel) permettant d'exprimer ces concepts (§2).
Pour être applicable, une méthode doit aussi prendre en compte l'aspect dynamique de la conception. La dynamique est généralement modélisée par un processus de conception décrivant pas à pas les tâches que le concepteur est censé effectuer [Girod 88]. On trouve ainsi, dans l'état de l'art, des processus pour la conception fonctionnelle ou pour la conception orienté-objet (Cf Chapitre 2) qui fournissent une trame générale d'aide à la conception. Il est souhaitable que de tels processus ne soient pas trop rigides pour offrir une certaine souplesse à l'activité de conception de nature essentiellement créatrice. Même si un certain cheminement à suivre peut être favorisé, il faut laisser le concepteur procéder par essais et erreurs, démarche qui est à l'essence même de toute activité de résolution de problème.
Dans cette optique, cette partie propose un **guide** de conception adapté à la méthode MECANO.

3.1. "Top-down" ou "Bottom-up"?

Une des questions posée couramment à propos du processus de conception est de savoir s'il doit être descendant ("Top-down") ou ascendant ("Bottom-up").
On a souvent opposé la conception fonctionnelle (descendante) à l'approche objet (ascendante). En fait, la conception par objet procède des deux à la fois. Il nous semble qu'une bonne conception est celle qui trouve le juste milieu entre généralisation et spécificité, réutilisation et création, originalité et standardisation. La confusion vient du fait que les termes "ascendant" et "descendant" ont des sens différents selon les auteurs.
Dans certains cas, la notion d'approche descendante est associée à la relation d'abstraction. Une conception sera dite descendante si elle part du plus abstrait et qu'elle descend vers le concret: on parle alors 'approches par raffinements successifs. Le qualificatif "ascendant" sera dans ce cas associé à toute opération inverse qui consiste à généraliser ou à abstraire des éléments, en vue d'obtenir un élément plus général.
D'autres auteurs qualifient de descendante une approche qui suit strictement les spécifications initiales pour créer progressivement tous les éléments nécessaires à la mise en place de la solution finale. Dans ce cas, toute opération tendant à réutiliser des éléments conçus auparavant (dans des projets antérieurs) est qualifiée d'ascendante.

Le problème réside dans le fait que ces deux approches sont différentes, voire contradictoires. Examinons dans le contexte de la méthode MECANO leurs significations.

Si on considère la première approche, on peut dire que l'opération de spécialisation par héritage est descendante, alors que l'opération inverse de généralisation est ascendante. Dans ce cas, la relation de spécialisation est vue comme une relation d'abstraction.

Dans la seconde approche, les deux opérations peuvent être qualifiées de descendantes puisque dans les deux cas de nouveaux éléments sont créés pour atteindre la solution finale, la deuxième opération n'étant qu'une mise en facteur améliorant la simplicité et la lisibilité de la conception.

Prenons maintenant la mise en place d'un composite, la création de ses parties sera considérée comme descendante par la première approche (la relation d'abstraction étant cette fois "fait partie de") mais elle sera qualifiée d'ascendante par la seconde si les parties sont importées depuis d'autres entités (domaines, composites ou applications).

Un dernier exemple enfin montre cette différence: une classe peut hériter d'une classe de la bibliothèque afin d'utiliser un concept en l'adaptant. Il s'agit d'un raffinement donc d'une opération descendante selon la première définition. Pourtant, il y a eu effectivement une réutilisation, opération ascendante selon la deuxième approche.

Nous voyons bien que si les deux approches ont des points de recouvrement (le raffinement d'un concept est associé à la création de nouveaux éléments), il existe aussi des points de divergence importants. Nous dirons donc que l'approche objet, et à fortiori MECANO, est une combinaison à la fois ascendante et descendante selon qu'on cherche à spécialiser ou au contraire à généraliser, et qu'elle est aussi fortement ascendante puisqu'elle fournit les outils pour réutiliser le plus possible.

Nous allons détailler, dans la partie suivante, le processus de conception MECANO.
Dans une première approche informelle, nous dirons que le processus de conception met l'accent sur les domaines. Ils doivent être mis en place dès le commencement de la conception, même s'ils ne comportent dans un premier temps qu'une seule classe.
Le domaine permet de définir, dans un univers relativement clos, un concept important du système en cours de conception. Il sera possible de rattacher à ce domaine, par des liens externes, tout ce dont il a besoin pour être défini, que ce soit via des composites, des applications ou d'autres domaines.
Les domaines interviennent aussi dès le départ dans l'objectif de réutilisation. Si la méthode a été utilisée sur d'autres projets traitant du même thème, il y a toutes les chances pour qu'un domaine contienne des classes qui pourront être directement réutilisées, ou bien qui offriront des spécialisations intermédiaires adaptées à de nouvelles réalisations.

Les composites sont mis en place en même temps que les classes. Il peut se produire qu'une classe soit, dans la suite, identifiée comme un composite. Dans ce cas, une opération descendante consistera à définir les nouvelles classes appartenant au composite. L'opération inverse a peu de chance de se produire puisque les parties ne peuvent pas exister indépendamment du composite.

La conception des classes elles-même obéira à un processus en trois étapes: définition des liens externes (héritage, client), définition de la spécification (interface, assertions), et enfin définition des caractéristiques internes et algorithmes des méthodes rédigés en langage informel ou directement dans un langage de programmation.

La mise en place des applications finales interviendra à la fin du processus. Il s'agira alors d'assembler des classes et composites appartenant aux différents domaines avec des classes ou des composites appartenant spécifiquement à l'application. En particulier, une classe racine sera construite pour initialiser l'exécution et lancer le système.

3.2. Mise en oeuvre de la méthode MECANO

3.2.1. Avertissement

MECANO est une méthode de conception qui propose un formalisme permettant d'exprimer une décomposition par objets et une implantation d'un problème informatique dans un modèle exécutable. La mise en oeuvre que nous proposons ici n'est pas une méthode d'analyse indiquant la manière de trouver les classes, les attributs ou les méthodes associées aux objets. Nous renvoyons pour cela le lecteur aux méthodes d'analyse déjà citées (voir chapitre 2) et à l'ouvrage de B. Meyer.
Cette partie présente un guide pour la mise en oeuvre d'une **conception** avec MECANO. Il définit la façon dont sont utilisés les concepts définis dans le chapitre précédent. Des réponses sont données aux questions suivantes: comment commencer une conception MECANO? comment construire un domaine? comment utiliser un domaine? un composite? de quelle façon construit-on une classe? Comment et à quel moment élaborer une ou plusieurs applications?
Dans le §4., nous donnerons un exemple de système conçu avec la méthode pour illustrer et affiner le processus de conception proposé.

3.2.2. Processus de conception MECANO: description générale

Le processus de conception MECANO se décompose en 5 grandes étapes:

PROCESSUS DE CONCEPTION MECANO: vue synthétique

A) IDENTIFICATION DES DOMAINES
B) CREATION DES DOMAINES
Pour chaque domaine identifié en 1)
B.1) DEFINITION DE LA CLASSE RACINE
B.2) MISE EN PLACE DES SPECIALISATIONS
B.3) MISE EN PLACE DES REALISATIONS
C) RETOUR A L'ETAPE A) avec les nouvelles classes identifiées EN B)
D) MISE EN PLACE DES APPLICATIONS
E) CONCEPTION DETAILLEE DES CLASSES CONCRETES

Ces cinq étapes constituent l'ossature de l'activité de conception avec MECANO. La chronologie des étapes est conseillée mais non obligatoire. Les étapes A) B) et C) devront cependant avoir lieu avant l'étape D). L'étape E) peut être traitée de façon plus diffuse dans les étapes précédentes si le besoin s'en fait sentir (clarification de la conception, vérification de la faisabilité d'une solution).
Le processus de conception est détaillé au §3.3.4.

3.2.3. SOUS-PROCESSUS DE CONCEPTION D'UNE CLASSE

Le concepteur doit mettre en place des structures (domaines, applications...) et des classes. La phase de définition d'une classe qu'elle soit virtuelle ou concrète suit un processus en 3 étapes.

SOUS-PROCESSUS DE CONCEPTION D'UNE CLASSE
i) MISE EN PLACE DES RELATIONS DE DELEGATION
ii) SPECIFICATION DE LA CLASSE
iii) CONCEPTION DETAILLEE DE LA CLASSE

Les étapes i) et ii) correspondent à la phase de définition de la classe.

Nota: ce sous-processus sera intégré au processus général dans les étapes B.1), B.2) et B.3).

i) MISE EN PLACE DES RELATIONS DE DELEGATION

Cette étape définit les liens de composition et de communication de la classe vers les autres classes. Pour cela, on recherche d'abord les structures Tout-Parties. On pourra décider de définir la classe comme un composite afin d'offrir un niveau d'abstraction supplémentaire.
Les collaborations nécessaires à la classe en cours de définition sont ensuite examinées. La règle est que toute relation de communication, portant sur une classe de domaine et ne devant pas supporter de création d'objet à l'exécution, doit utiliser la classe virtuelle la plus élevée possible dans la hiérarchie de spécialisation. De cette manière, le client s'abstrait de tous les détails inutiles à son niveau (détails d'implantation ou détails de spécification).

ii) SPECIFICATION DE CLASSE

On spécifie la classe en utilisant le formalisme textuel défini dans le §2.
Toutes les rubriques sont définies. Pour la rubrique **méthode**, seules les méthodes exportées doivent apparaître. Les méthodes sont uniquement spécifiées (nom, paramètres, type résultat, spécification, pré-condition et post-condition), la réalisation du corps intervenant plus tard dans le cycle. Cependant, dans une classe virtuelle, on admet que soient réalisées les méthodes qui s'expriment simplement en terme des autres méthodes de la classe (méta-algorithme). Les attributs correspondant aux relations de communication seront nommés. Les attributs de composition doivent être identiques à ceux utilisés dans la représentation graphique (étiquette de la relation).

iii) CONCEPTION DETAILLEE DE LA CLASSE

Cette étape pourra, selon les classes, être différée à l'étape E) du processus de conception global. Elle est décrite au §3.2.4.E).

3.2.4. Processus de conception: description détaillée

A) IDENTIFICATION DES DOMAINES

De façon générale, il est conseillé de créer un domaine par concept identifié lors de l'analyse. Dans certains cas, le domaine sera réduit à deux classes: la classe virtuelle racine définissant l'abstraction du domaine et une classe concrète définissant la réalisation particulière de cette abstraction dans le contexte du système à développer. Cette démarche est importante car elle permet d'une part de bien se poser la question de l'abstraction du concept "quels sont les services proposés et les responsabilités de la classe?", et d'autre part, de garder une marge de manoeuvre pour des concrétisations ou des spécialisations ultérieures (évolution ou réutilisation).

B) CREATION DES DOMAINES

Il s'agit de mettre en place la classe racine (concept du domaine) et la structure de spécialisation/réalisation. Les trois étapes B.1), B.2) et B.3) doivent être consécutives mais peuvent avoir lieu de façon croisée avec les même étapes pour un autre domaine. On pourra, par exemple, effectuer l'étape B.1), B.2) d'un domaine D1 puis l'étape B.1) d'un domaine D2, puis l'étape B.3) du domaine D1 etc ...

B.1) DEFINITION DE LA CLASSE RACINE

Pour créer la classe virtuelle racine, on s'attache à définir le concept le plus général possible, en s'appuyant sur une modélisation de type abstrait, sans faire d'hypothèses sur une implantation particulière.
Pour chaque objet identifié dans le cahier des charges ou dans l'analyse, il suffit de le considérer comme une boite noire et de chercher les services qu'il offre aux autres objets. Ces services sont de deux natures: action et fonction. Les actions sont les boutons qui permettent d'actionner la boite, elles modifient l'état de l'objet. Les fonctions correspondent aux voyants et indicateurs de la boite, elles donnent des compte-rendus sur l'état de l'objet.
La définition de la classe racine suit le sous-processus de définition de classe présenté plus haut.

B.2) MISE EN PLACE DES SPECIALISATIONS

B.2.1) Création des liens de spécialisation

On recherche les spécialisations intéressantes du concept défini par la classe racine. Il y a généralement plusieurs spécialisations possibles pour un concept donné. Dans certains cas, on choisira la plus pertinente dans l'application en cours de développement, et dans d'autres cas la plus naturelle c'est à dire celle correspondant à une spécialisation du monde réel. Si d'autres spécialisations sont intéressantes, elle pourront intervenir sous la forme d'un héritage de comportement.
La classe spécialisée doit enrichir le service offert par sa parente ou bien avoir une collaboration de plus (cf taxonomie de l'héritage, chapitre 3). Les spécialisations peuvent être définies pour donner des niveaux d'abstraction pertinents pour la compréhension du logiciel. Les spécialisations intermédiaires doivent aussi être recherchées en tenant compte de leur capacité à être réutilisées.

Une spécialisation peut aussi dépendre d'une spécialisation dans un autre domaine. Par exemple, la spécialisation d'une classe A d'un domaine DA pourra dépendre de la spécialisation d'une classe B d'un domaine DB si A est une partie du composite B, ou plus généralement, si B est cliente de A. Dans ce cas, la spécialisation de A sera parallèle à la spécialisation de B (espaces de spécialisations parallèles).

Enfin, des spécialisations peuvent être introduites quand une structure de choix sur des objets est pressentie. En effet, les structures de choix peuvent être réalisées au moyen du polymorphisme, rendant le logiciel évolutif [Meyer 88].

L'ajout de méthode peut se faire soit en créant les nouvelles méthodes adéquates, soit en définissant un lien d'héritage de comportement sur une classe fournissant les méthodes.

B.2.2) Définition des classes spécialisées

Pour chaque nouvelle spécialisation découverte, la classe correspondante est définie en appliquant le processus de définition de classe.
Les méthodes redéfinies conservent la spécification d'origine. Certaines méthodes peuvent s'exprimer en terme de méthodes virtuelles. Le concepteur peut choisir de les réaliser dès maintenant si cette réalisation est suffisamment générale ou si elle

constitue une réalisation par défaut. Par exemple, la méthode *inférieur* est exprimée en terme des méthodes virtuelles *supérieur* et *égal*, la méthode *gauche* de tortue est définie en fonction de la méthode *droite*.
Les attributs peuvent aussi être redéfinis, c'est à dire que le type de l'attribut doit être compatible (hérite) à celui du même attribut dans la classe mère (covariance).

Enfin, dans certains cas, la classe virtuelle racine peut être candidate à la généralisation si on voit immédiatement un concept pertinent plus abstrait que le premier.

B.3) MISE EN PLACE DES REALISATIONS

B.3.1) Création des liens de réalisation

Les réalisations sont définies pour concrétiser les classes. Le choix des concrétisations repose essentiellement sur deux principes:
a) concrétisations pour une version du système: on réalise autant de réalisations que de versions du système à produire.
b) concrétisations candidates au polymorphisme: les réalisations seront toutes utilisées dans le système final. Elles seront manipulées par des clients de classes virtuelles ancêtres de la réalisation. Des héritages d'implantation peuvent être définis à cette étape.

B.3.2) Définition des classes concrètes

Chaque classe concrète est définie en utilisant des liens d'implantation, des nouveaux attributs et des méthodes internes. Le concepteur peut effectuer cette étape immédiatement ou bien la différer à la fin de la conception (conception détaillée).

C) RETOUR à l'étape A) avec les nouvelles classes identifiées en B)

Dans cette phase, il s'agit de repasser dans les étapes A) et B) du processus, en identifiant et créant des domaines relatifs aux nouvelles classes référencées dans les étapes précédentes. Ces nouvelles classes ont été introduites pour fournir un service ou comme super-classes (comportement, implantation) des classes des domaines définis à l'étape B). Dans ces domaines, elles apparaissent comme des entités externes.

Pour les classes réutilisées, on doit indiquer la provenance (domaine ou application d'un autre projet, d'une librairie). Pour une classe nouvelle, on pourra l'intégrer à l'application (les applications) en cours de conception, ou plus généralement, un nouveau domaine sera créé afin de l'encapsuler dans un couple classe virtuelle/réalisation. Cette dernière solution est préférable car elle force à la généralisation et permet d'envisager des évolutions et des réutilisations futures.
Pour chaque nouveau domaine identifié, on recommence l'étape B).

D) MISE EN PLACE DES APPLICATIONS

Cette étape concerne la création d'une ou des applications finales, décrivant le système sous sa forme opérationnelle. On peut définir plusieurs applications qui correspondent à des versions différentes du système (par exemple, une version avec interface graphique et une autre avec interface ligne à ligne) ou bien à des évolutions d'une version par ajout de fonctionnalités.
Cette phase se décompose en trois sous-étapes. Pour chaque application, les activités suivantes doivent être effectuées:

D 1) création de la classe racine et des classes auxiliaires,
D 2) définition des liens de communication,
D 3) conception détaillée de la classe racine et des classes auxiliaires.

D.1) Création de la classe racine et des classes auxiliaires
Il s'agit de définir la classe racine de l'application qui sera instanciée en premier lors de l'exécution du système. Des classes auxiliaires peuvent être introduites si cela est nécessaire. Cependant, la majorité des classes de l'application proviendra des domaines construits. Dans ce cas, les classes utilisées seront les classes concrètes adaptées à telle ou telle application. En effet, la classe racine sera chargée de créer des instances de ces classes, elle doit donc connaître le type exact des objets manipulés. Elle manipulera les objets au travers des classes concrètes et non des classes abstraites.

D.2) Définition des liens de communication
Les liens de communications sont établis entre les classes de l'application. Ces liens forment l'ossature opérationnelle de l'application.

D.3) Conception détaillée de la classe racine et des classes auxiliaires
Dans cette phase on rédige la conception détaillée des classes introduites dans l'application. On décrit en particulier le **script de création** des objets (séquencement des créations d'objets, paramètres d'initialisation).
Il faudra veiller à ce que toutes les références à des classes virtuelles correspondent bien à un objet concret à l'exécution.

E) CONCEPTION DETAILLEE DES CLASSES CONCRETES

Les classes concrètes n'ayant pas fait l'objet d'une conception détaillée sont traitées à cette étape. La conception détaillée des classes concrètes se décompose en 4 étapes. L'objectif est de définir la réalisation des classes afin que la programmation dans le langage cible soit non ambiguë et aisée. Ceci est fait au travers de nouveaux liens d'héritage et de délégation "utilitaires" et en définissant l'implantation des méthodes exportées et internes.

E.1) AJOUT D'HERITAGE DE COMPORTEMENT ou D'IMPLANTATION EVENTUEL
Des liens d'héritage plus techniques peuvent être créés pour assurer des fonctionnalités. C'est le cas, par exemple, de l'héritage sur des classes de "support" telles que STORABLE, STD_FILE, VIEWABLE en EIFFEL. Ces héritages sont généralement à classer dans les implantations et les comportements. On pourra ne pas les faire apparaître sur la représentation graphique pour ne pas alourdir la conception.

E.2) DEFINITION DU CODE DES METHODES EXPORTEES
La définition du code des méthodes exportées se fera au moyen d'un pseudo-langage ou bien directement dans le langage de programmation cible.

E.3) SPECIFICATION ET DEFINITION DES METHODES INTERNES
Pour réaliser l'étape précédente, le concepteur peut être amené, par la stratégie habituelle du raffinage fonctionnel, à définir des méthodes internes. Ces méthodes seront, dans un premier temps, spécifiées au moment de leur utilisation dans la méthode appelante, puis, dans un deuxième temps, seront réalisées comme les méthodes exportées. Il est important de bien spécifier les méthodes internes car, en cas de scission ultérieure de la classe en deux classes collaborantes, elles pourront devenir des méthodes exportées.

E.4) DEFINITION DES ATTRIBUTS UTILITAIRES
Les attributs utilitaires peuvent ne pas être représentés dans le formalisme graphique car ils ont un rôle technique proche de l'implantation. Ils seront définis directement dans la classe. Ils appartiennent ,le plus souvent, à des types de base (entiers, caractères, booléens).

4. EXEMPLE DE CONCEPTION MECANO: LA TORTUE

Cette partie développe un exemple de conception élaborée au moyen de la méthode MECANO et du processus défini dans la section précédente. L'exemple choisi est volontairement simple, pour que les spécifications soient vite comprises de tous, sans être simpliste. Nous verrons qu'il se décompose en 8 domaines nouveaux et 4 applications, ce qui correspond à une cinquantaine de classes.
Le système à développer est celui, bien connu, de la tortue introduite par S. Papert dans LOGO [Papert 81]. Nos objectifs en développant cet exemple sont multiples. Il s'agit, en premier lieu, de concrétiser les concepts de la méthode et le processus de conception. D'autre part, il est possible de montrer comment la méthode permet d'obtenir un certain nombre des qualités définies dans le premier chapitre dont la réutilisation, la réutilisabilité et l'évolutivité. Nous montrons, en utilisant deux évolutions différentes des spécifications externes, en quoi la conception obtenue possède une bonne flexibilité permettant de ne modifier que les quelques classes concernées sans remettre en cause l'architecture générale.

4.1. Le problème

La tortue est une entité capable de se mouvoir d'une position à une autre, dans un espace à deux dimensions. L'utilisateur donne des ordres de déplacement qu'elle exécute. Les ordres exécutables sont *avancer* ou *reculer* d'un certain nombre de "pas", tourner à *gauche* ou à *droite* d'un certain angle.
La tortue possède une plume qu'elle peut baisser ou lever sur demande:
- Si la plume est basse, la tortue trace une droite en se déplaçant.
- Si la plume est haute, le déplacement se fait sans traçage.

En plus des ordres de déplacement, les ordres *lever*, *baisser* (la plume) sont donc disponibles.
Une session de travail avec la tortue consiste à acquérir et à traiter les ordres successifs que donne l'utilisateur.

Dès à présent, on demande d'envisager deux implantations possibles de la tortue:

1) un **robot mobile** télécommandé qui se déplace sur une feuille de papier étendue sur le sol[1]. Ce robot est commandé depuis une console dans laquelle on introduit des cartes correspondant aux ordres à exécuter. Une plume, dissimulée au centre de la tortue, est actionnable (descente ou montée).
2) une **tortue simulée** sur l'écran de la machine par un petit triangle isocèle. Le sommet du triangle indique en permanence la direction de la tortue (appelée aussi le *cap*). Deux ordres supplémentaires sont prévus pour cette version:

reset : efface l'écran et met la tortue au centre de l'écran
maison : recentre la tortue, cap au nord

L'utilisateur a le choix entre deux modes d'utilisation:
1) mode **interactif**: exécution des ordres au fur et à mesure de la saisie
2) mode **programmable**: l'utilisateur dispose d'un ordre *début*, indiquant que les ordres suivants correspondent à un programme qui doit être stocké en vue d'une

[1] Historiquement, cette version est la première à avoir existé. Les études de Papert inspirées de Piaget montre que les enfants en s'identifiant à la tortue traçant des traits sur le sol, acquièrent très vite des notions de géométrie relativement complexes [Papert 81].

exécution ultérieure. La fin du programme, c'est à dire la fin du stockage des ordres, est indiquée par l'ordre *fin*. L'exécution du programme en entier peut être lancée par l'ordre *exécuter*.

Les ordres *avancer*, *reculer*, *gauche*, *droite* nécessitent un paramètre

L'interface externe de saisie n'est pas fixée : le système doit être facilement adaptable à une interface de type TTY (interface sessionnelle) ou de type menu (semi-graphique ou graphique comme X-windows). Dans le cas de la tortue radio-commandée, l'interface peut aussi se présenter sous la forme de cartes décodées par un lecteur.

4.2. Mise en oeuvre du processus de conception MECANO

Nous appliquons le processus de conception présenté en §3. sur l'exemple de la tortue.

4.2.1. ETAPE A) IDENTIFICATION DES DOMAINES

Cette première étape permet de dégager les domaines TORTUE, PLUME, PROGRAMME, ORDRE, SESSION (figure 11). Ces premiers objets peuvent être trouvés avec une méthode d'extraction systématique depuis les spécifications comme celle proposée par G. Booch[1] (cf chapitre 2).

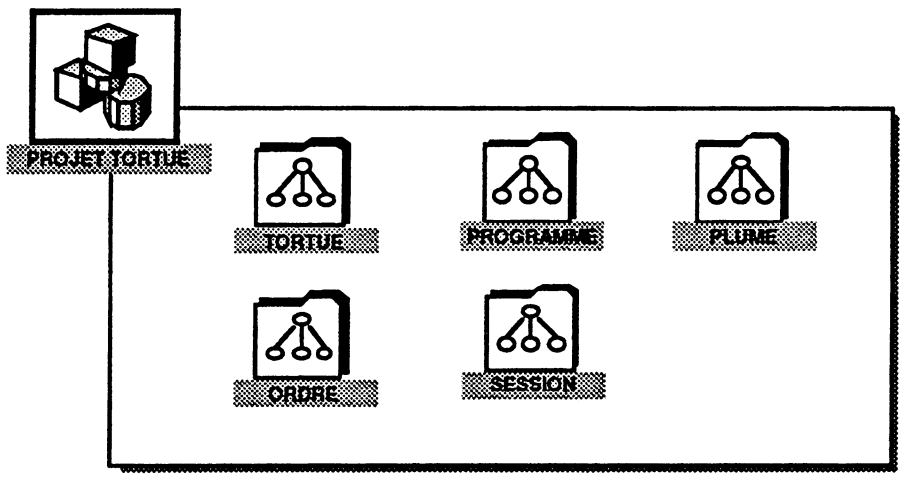


figure 11 : le projet tortue

La TORTUE représente le concept principal de l'application, c'est à dire une entité capable de se déplacer et de réagir aux ordres avancer, reculer, gauche etc...
La PLUME est le composant de la tortue lui permettant de tracer des traits lors des déplacements.
Un PROGRAMME est une séquence d'ordres destinés à une tortue. L'exécution d'un programme correspond à l'exécution successive des ordres qui le composent.
Un ORDRE est un objet stockable dans un programme et applicable en général à une tortue.
Une SESSION représente l'interface utilisateur et donc la possibilité d'acquérir des ordres en vue de leur traitement de façon séquentielle.

---

[1] On pourra cependant remarquer que le bon sens conduit sans doute à la même liste.

4.2.2. ETAPE B) CREATION DES DOMAINES

Nous présentons dans cette section, les différents domaines introduits pour la conception du système tortue.

DOMAINE TORTUE

B.1. DEFINITION DE LA CLASSE RACINE

La classe TORTUE sera la racine du domaine des tortues.

B.1.1. MISE EN PLACE DES LIENS DE DELEGATION

Une tortue est composée d'une PLUME et d'un NOM: c'est un composite.

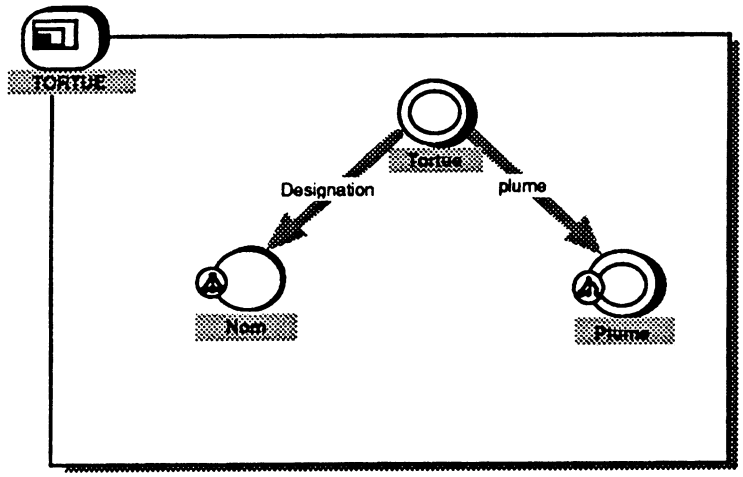


figure 12 : composite tortue

B.1.2. SPECIFICATION DE LA CLASSE

Nota: les méthodes retardées sont en format **relief.**

CLASSE TORTUE
SPECIFICATION **virtuelle** **rôle** racine du domaine tortue, définit un objet pouvant se déplacer droit devant lui. La tortue peut aussi tourner d'un angle donné à gauche ou à droite. Elle possède une plume: quand la plume est baissée, elle trace un trait en se déplaçant. **mot-clé** tortue, déplacement, dessiner **invariant** la tortue possède une direction et avance ou recule toujours dans cette direction et plume.haute => (avancer(d) ; reculer(d) = identité) et (gauche(a) ; droite(a) = identité)
INTERFACE avancer(D : DISTANCE), reculer(D : DISTANCE), droite(A : ANGLE), gauche(A : ANGLE), leveplume, baisseplume
METHODE **action avancer(D : DISTANCE) retardée** **spécification** fait avancer la tortue de D positions **précond** D >= 0 (pour les utilisateurs uniquement)

```
        postcond tortue avancée de D
    action reculer(D : DISTANCE)
        spécification fait reculer la tortue de D positions
        précond D >= 0
        postcond tortue reculée de D
        définition Current.avancer(- D)
    action droite(A : ANGLE) retardée
        spécification fait tourner la tortue de A degrés à droite
        précond A >= 0 et A < 360
        postcond tortue tournée de A dans le sens antitrigonométrique
    action gauche(A : ANGLE)
        spécification fait tourner la tortue de A degrés à gauche
        précond A >= 0 et A < 360
        postcond tortue tournée de A dans le sens trigonométrique
        définition droite(- A.modulo(360))
    action leveplume
        spécification lève la plume de la tortue
        postcond plume.haute = vrai
        définition Plume.lever
    action baisseplume
        spécification baisse la plume de la tortue
        postcond plume.haute = faux
        définition Plume.baisser
ATTRIBUT
    composition
        plume : PLUME
        spécification plume de la tortue, peut se baisser ou se lever.
        identification : NOM
        spécification nom de la tortue, permettant de l'identifier de manière unique.
FIN CLASSE TORTUE
```

Remarques:
La méthode *reculer* est définie en terme de la méthode *avancer*, de même pour *gauche* et *droite*.
La classe PLUME devra posséder les méthodes *lever*, *baisser* et la fonction booléenne *haute*.

B.2. SPECIALISATIONS

B.2.1. Création des liens de spécialisation

TORTUE se spécialise en TORTUE_ABSOLUE définissant le concept d'une tortue connaissant un point de référence (le centre de l'écran par exemple) afin de pouvoir exécuter les ordres *reset* et *maison*.

B.2.2. Définition des classes spécialisées

Cette tortue possède les deux nouveaux ordres *reset* et *maison*. Remarque: la tortue robot ne possède pas ces ordres.

CLASSE TORTUEABSOLUE

```
SPECIFICATION
    virtuelle
    rôle Spécialisation de tortue. Ce type de tortue se déplace dans un plan connu (par exemple l'écran) et peut se
    repérer par rapport au centre du plan. Elle possède les ordres supplémentaires RESET et MAISON.
    A l'initialisation, la tortue est au centre, cap au Nord (axe des y).
    mot-clé tortue absolue
    invariant
```

le cap est l'angle de la direction de la tortue avec l'axe des y

INTERFACE

avancer(D : INTEGER), reculer(D : INTEGER), droite(A : ANGLE),
gauche(A : ANGLE), leveplume, baisseplume, maison, reset,
Cap: ANGLE, Pos : INTEGER

METHODE

action avancer(D : integer) retardée
spécification fait avancer la tortue de D positions
précond D >= 0 (pour les utilisateurs uniquement)
postcond Pos.x = old Pos.x + Cap.cosinus et Pos.y = old Pos.y + Cap.sinus

action droite(A : Angle) retardée
spécification fait tourner la tortue de A degrés à droite
précond D >= 0 et D < 360
postcond Cap = old Cap.ajout(A)

action maison retardée
spécification la tortue rentre à la maison c'est a dire en position 0,0, cap au nord
postcond Pos.x=0 et Pos.y=0 et Cap=0

action reset retardée
spécification la tortue rentre à la maison c'est a dire position 0,0, cap à 0 et la feuille de tracés est effacée
postcond Pos.x=0 et Pos.y=0 et Cap=0 et feuille effacée et plume haute

ATTRIBUT

composition
Plume : PLUMESIMULEE **redéfini**
spécification plume de la tortue, on s'intéresse uniquement à l'état de la plume
Position : POINT;
spécification position de la tortue en x et y
Cap : ANGLE;
spécification cap de la tortue, angle avec l'axe des y dans le sens antitrigonométrique

FIN CLASSE TORTUEABSOLUE

Remarque: Les post-conditions de *avancer* et *droite* ont été affinées. La tortue possède une *position* et un *cap* pour se repérer dans le plan.

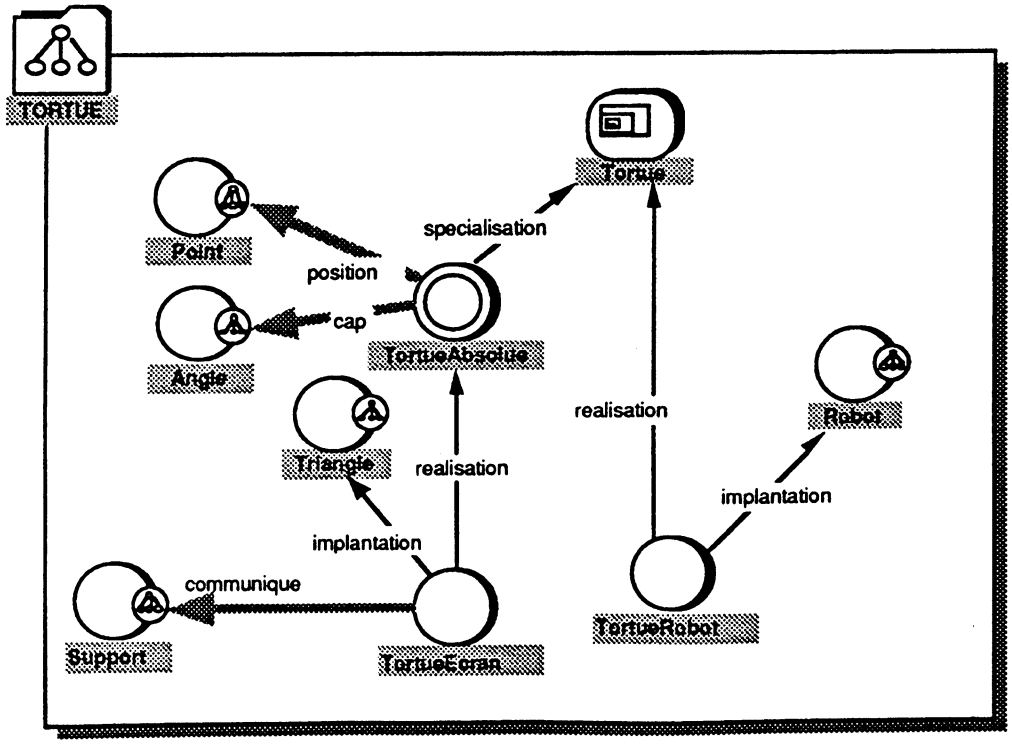


figure 13 : le domaine tortue

B.3. REALISATIONS

B.3.1. Création des liens de réalisation (figure 13)

Deux réalisations sont définies. TORTUE_ECRAN réalise TORTUE_ABSOLUE en héritant de l'implantation TRIANGLE (classe réutilisée provenant du domaine des FIGURES) et en utilisant un SUPPORT. TORTUE_ROBOT réalise TORTUE au moyen de la classe ROBOT (héritage d'implantation).

B.3.2. Définition des classes concrètes

Différée à l'étape de conception détaillée

DOMAINE PLUME

B.1. DEFINITION DE LA CLASSE RACINE

La classe PLUME sera la racine du domaine des plumes.

B.1.1. MISE EN PLACE DES LIENS DE DELEGATION

La plume est un composant de Tortue, elle n'est cliente d'aucune autre classe.

B.1.2. SPECIFICATION DE LA CLASSE

CLASSE PLUME

SPECIFICATION
virtuelle
rôle racine du domaine plume, définit un objet abstrait à deux états, pouvant se lever ou se baisser
mot-clé plume, crayon, tortue

INTERFACE
lever, baisser, haute: BOOLEEN

METHODE
action lever retardée
spécification fait lever la plume
précond
postcond haute

action baisser retardée
spécification fait baisser la plume
précond
postcond non haute

action haute : booléen retardée
spécification indique si la plume est haute
précond
postcond

FIN CLASSE PLUME

B.2. SPECIALISATIONS

Pas de spécialisation particulière de plume.

B.3. REALISATIONS

B.3.1. Création des liens de réalisation

La plume étant un composant de la tortue, les réalisations de plume correspondent aux deux réalisations de Tortue: PlumeSimulée pour TortueEcran, et Crayon pour la

TortueRobot. Le Crayon hérite du mécanisme de CrayonPiloté (commande d'électroaimant).

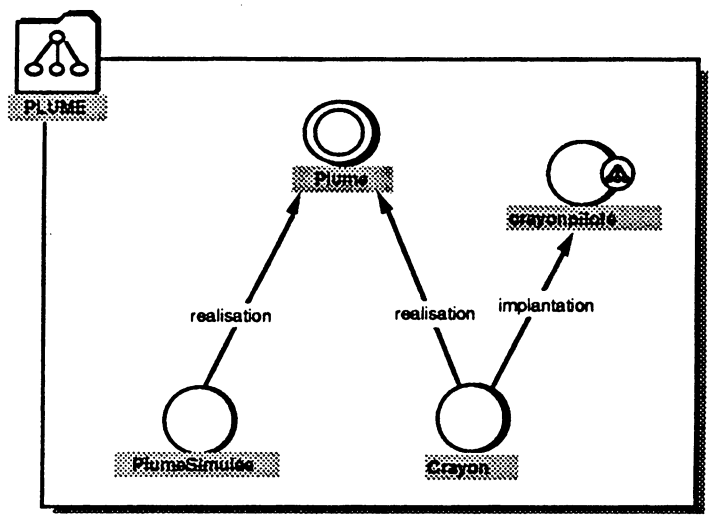


figure 14 : le domaine plume

B.3.2. Définition des classes concrètes
Voir conception détaillée

DOMAINE PROGRAMME

B.1. DEFINITION DE LA CLASSE RACINE
La classe Programme sera la racine du domaine des programmes.

B.1.1. MISE EN PLACE DES LIENS DE DELEGATION
Pas de liens de délégation pour le programme

B.1.2. SPECIFICATION DE LA CLASSE

Remarque: les méthodes *Debutprog*, *Finprog*, *Exécutable*, *Programmable* sont réalisées.
Les méthodes *Run* et *Stocker* sont différées.

CLASSE PROGRAMME
SPECIFICATION **virtuelle** **rôle** programme contenant des ordres de l'utilisateur. On peut en demander une exécution. Gère de plus les modes programmable et exécutable **mot-clé** programme, exécution , stockage d'ordres **invariant** programmable ou exécutable
INTERFACE Run, Debutprog, Finprog, Programmable: Booléen, Exécutable: Booléen, Stocker(O: ORDRE)

METHODE
action Run retardée
spécification exécute le programme
précond programmable = faux (exécutable = vrai)
postcond le programme est exécuté
action Stocker(O:Ordre) retardée
spécification stocke l'ordre O
précond
postcond ordre O stocké
action Debutprog
spécification met le programme en mode programmable
précond
postcond programmable= vrai
définition mode <- vrai
action FinProg
spécification met le programme en mode exécutable
précond
postcond exécutable= vrai
définition mode <- faux
fonction Programmable: Booléen
spécification rend l'état du programme (vrai si mode programmable)
définition resultat <- mode
fonction Exécutable: Booléen
spécification rend l'état du programme (faux si mode programmable)
définition resultat <- not mode

ATTRIBUT
utilitaire
spécification: état du programme (exécutable ou programmable)
mode: Booléen

FIN CLASSE PROGRAMME

Remarque: les méthodes *DebutProg*, *FinProg*, *Programmable* et *Exécutable* peuvent être réalisées dès maintenant. Par contre, aucune stratégie n'est fixée pour l'implantation du programme (liste, arbre abstrait, fichier ASCII), en conséquence les méthodes *Run* et *Stocker* sont retardées.

B.2. SPECIALISATIONS

Pas de spécialisations de programme.

B.3. REALISATIONS

B.3.1. Création des liens de réalisation

Le programme est réalisé par la classe ProgrammeSéquentiel décrivant les programmes constitués d'une séquence d'ordres. Cette classe hérite en implantation de la classe LinkedList (Liste chaînée) provenant du domaine des LISTES[1]. LinkedList est une classe générique ayant comme paramètre formel le type des éléments contenus dans la liste. L'héritage d'implantation de ProgrammeSequentiel s'accompagne de l'instanciation du paramètre générique par la classe ORDRE. Le domaine ORDRE est défini dans la section suivante.

[1] On peut considérer ce domaine comme étant un domaine réutilisé provenant d'une librairie de structures de données comme il en existe dans la majorité des langages de programmation par objets (cf §6.2. , [ISE 90a] et [ISE 90b]).

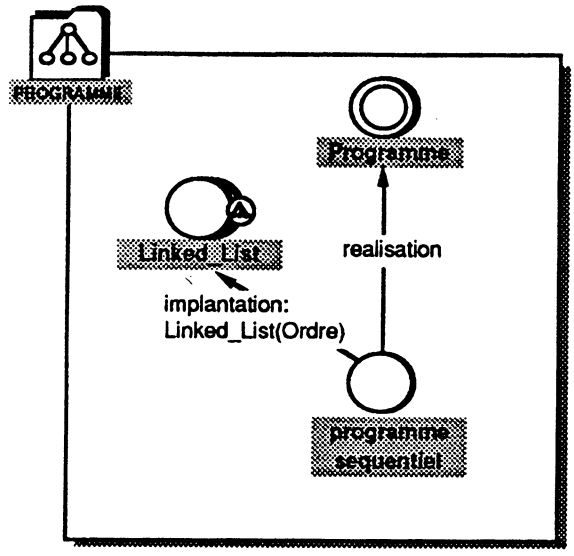


figure 15 : le domaine programme

B.3.2. Définition des classes concrètes

Voir conception détaillée

DOMAINE ORDRE

Ce domaine décrit les ordres possibles de l'utilisateur sur le système de gestion de la tortue. Le choix de créer une classe par Ordre possible est dû au fait que le programme doit stocker des ordres en vue de leur exécution. Il faut donc pouvoir concrétiser le concept d'ordre. Il faut de plus disposer d'une structure polymorphe afin que les intervenants sur les ordres (programme, session, contrôle) puissent les manipuler de façon abstraite.

B.1. DEFINITION DE LA CLASSE RACINE

La classe ORDRE sera la racine du domaine des ordres. Un ordre a la compétence de s'exécuter et d'exiger un paramètre.

B.1.1. MISE EN PLACE DES LIENS DE DELEGATION

Un ordre est composé d'un NOM et d'un PARAMETRE. C'est un composite.

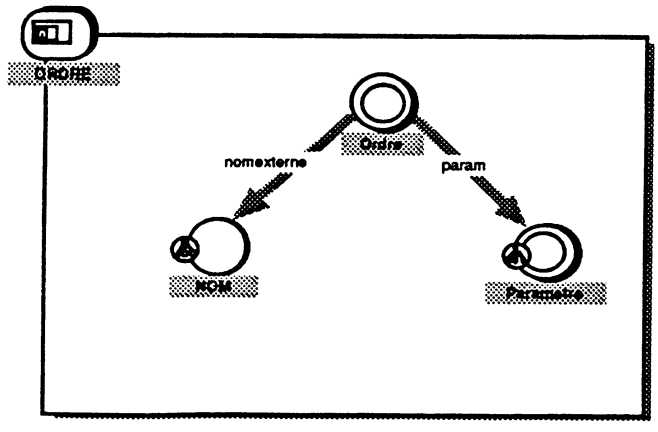


figure 16: composite ordre

B.1.2. SPECIFICATION DE LA CLASSE

CLASSE ORDRE
SPECIFICATION **virtuelle** **rôle** racine du domaine ordre, définit un ordre possible du système donné par l'utilisateur **mot-clé** ordre, commande, utilisateur **invariant**
INTERFACE exécuter, paramètre: PARAM, exige_paramètre: BOOLEEN, set_paramètre(P: INTEGER) nom: STRING, set_nom(N: STRING), tracer(S: OUTPUT)
METHODE **action exécuter retardée** **spécification** exécute l'ordre **précond** exige_paramètre => not paramètre.void **postcond** l'ordre a été exécuté **action** set_paramètre(P: INTEGER) **spécification** met à jour le paramètre de l'ordre **postcond** le paramètre est mis a jour **définition** Paramètre.creer, Paramètre.Set_Valeur(P) **action** set_nom(N: STRING) **spécification** met à jour le nom de l'ordre **définition** Nom <- N **action** tracer(S : OUTPUT) **spécification** affiche l'ordre sur la sortie S (écran, imprimante, etc) et permet de tracer l'exécution des ordres **définition** S.affiche('ordre: ',Nom) **fonction exige_paramètre : BOOLEEN retardée** **spécification** indique si l'ordre a besoin d'un paramètre
ATTRIBUT **composition** Nom : STRING **spécification** nom externe de l'ordre Paramètre : PARAM **spécification** paramètre de l'ordre (éventuellement) **FIN CLASSE ORDRE**

B.2. SPECIALISATIONS

B.2.1. Création des liens de spécialisation (figure 17)

On choisit d'homogénéiser le concept d'ordre en spécialisant sur l'objet auquel s'applique l'ordre (spécialisation sur la délégation). L'ordre peut s'appliquer au programme (début, fin, exécuter), à la tortue (avancer, reculer, gauche, ...) et à la session (halt). Une Session correspond à l'interface utilisateur et sera décrite dans la suite.

Trois spécialisations sont donc créées: OrdreTortue, OrdreProgramme et OrdreSession. OrdreTortue est de nouveau spécialisé en OrdreAbsolu (ordres applicables à une tortue absolue uniquement).

B.2.2. Définition des classes spécialisées

Relation de délégation

OrdreProgramme communique avec Programme. On ajoute la méthode *set_programme* et on redéfinit *tracer*. La fonction *exige_paramètre* est définie à faux.

OrdreTortue communique avec Tortue. La méthode *set_tortue* est ajoutée et *tracer* est redéfinie.

OrdreSession communique avec Session. On ajoute la méthode *set_chef* et *exige_paramètre* est définie à faux.
OrdreAbsolu communique avec TortueAbsolue. L'attribut *tortue* est redéfini par le type TortueAbsolue (covariance).
Les autres méthodes, dont *exécuter*, sont toujours retardées.

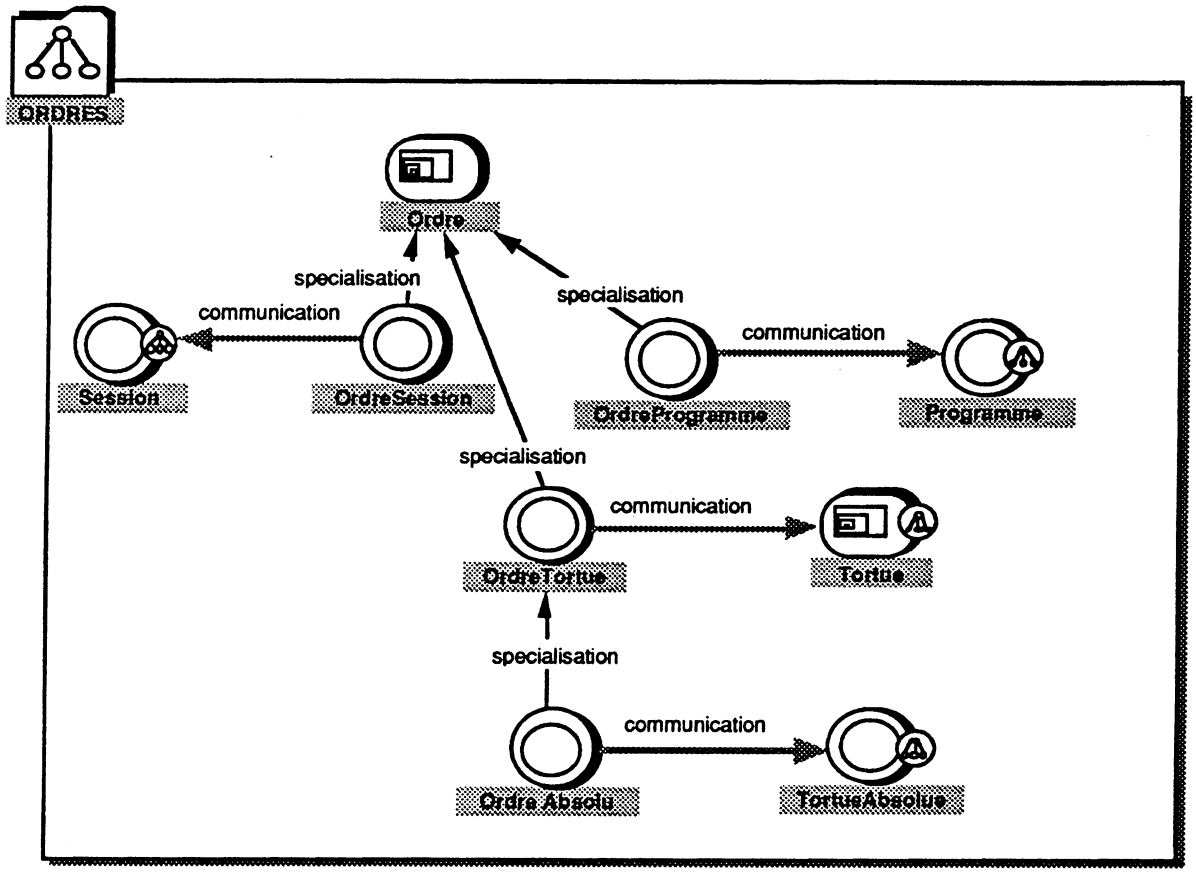


figure 17: domaine ordre (spécialisations)

CLASSE ORDREPROGRAMME
SPECIFICATION **virtuelle** **rôle** spécialisation de ordre, définit un ordre applicable au programme **mot-clé** ordre, programmation **invariant**
INTERFACE exécuter, paramètre: PARAM, exige_paramètre: BOOLEEN, set_programme(P:PROGRAMME), set_paramètre(P:INTEGER), nom : STRING , set_nom(N: STRING), tracer(S: OUTPUT)
METHODE **action** set_programme(P: PROGRAMME) **définition** Source <- P **action** tracer(S : OUTPUT) **redéfini** **spécification** affiche l'ordre sur la sortie S (écran, imprimante, etc), permet de tracer l'exécution des ordres **définition** S.affiche('ordre: ',Nom,' appliqué au programme') **fonction** exige_paramètre **défini** **définition resultat** <- faux

ATTRIBUT
communication
Source : PROGRAMME
spécification programme du système (éventuellement vide)

FIN CLASSE ORDREPROGRAMME

CLASSE ORDRETORTUE

SPECIFICATION
virtuelle
rôle spécialisation de ordre, définit un ordre applicable à une tortue
mot-clé ordre, programme, dessin, tortue
invariant

INTERFACE
exécuter, paramètre: PARAM, set_tortue(T: TORTUE),
set_paramètre(P: INTEGER), exige_paramètre: BOOLEEN
nom : STRING, set_nom(N: STRING), tracer(S: OUTPUT)

METHODE
action set_tortue(T : comme Tortue)
spécification met à jour l'attribut tortue de l'ordre
postcond Tortue = T
définition Tortue <- T
action tracer(S : OUTPUT) **redéfini**
spécification affiche l'ordre sur la sortie S (écran, imprimante, etc), permet de tracer l'exécution des ordres
définition S.affiche('ordre: ',Nom,' appliqué à la tortue:', Tortue.nom)

ATTRIBUT
communication
Tortue : TORTUE
spécification tortue sur laquelle doit s'appliquer l'ordre
FIN CLASSE ORDRETORTUE

CLASSE ORDRESESSION

SPECIFICATION
virtuelle
rôle spécialisation de ordre, défini un ordre applicable à session.
mot-clé ordre, appliqué à session
invariant

INTERFACE
exécuter, paramètre: PARAM, exige_paramètre: BOOLEEN,
set_paramètre(P:INTEGER), set_chef(C: SESSION),
nom : STRING , set_nom(N: STRING), tracer(S: OUTPUT)

METHODE
action set_chef(C: SESSION)
définition Chef <- C
fonction exige_paramètre **défini**
définition resultat <- faux

ATTRIBUT
communication
Chef : SESSION
spécification chef est l'objet session qui contrôle l'interface utilisateur,
on doit lui indiquer que l'utilisateur demande un arrêt
FIN CLASSE ORDRESESSION

B.3. REALISATIONS

B.3.1. Création des liens de réalisation

Les réalisations de OrdreTortue sont Avancer, Reculer, Droite, Gauche, LeverPlume et BaisserPlume.
Les réalisations de OrdreAbsolu sont Reset et Maison.
Les réalisations de OrdreProgramme sont Debut, Fin et Exécuter.
Les réalisations de OrdreSession sont Halt et Inconnu. Inconnu est un ordre particulier introduit pour prendre en compte les ordres erronés. Halt est l'ordre de fin de session.
Chacune des réalisations devra définir les caractéristiques *exécuter*, *nom* et pour les ordres de la tortue, *exige_paramètre*.

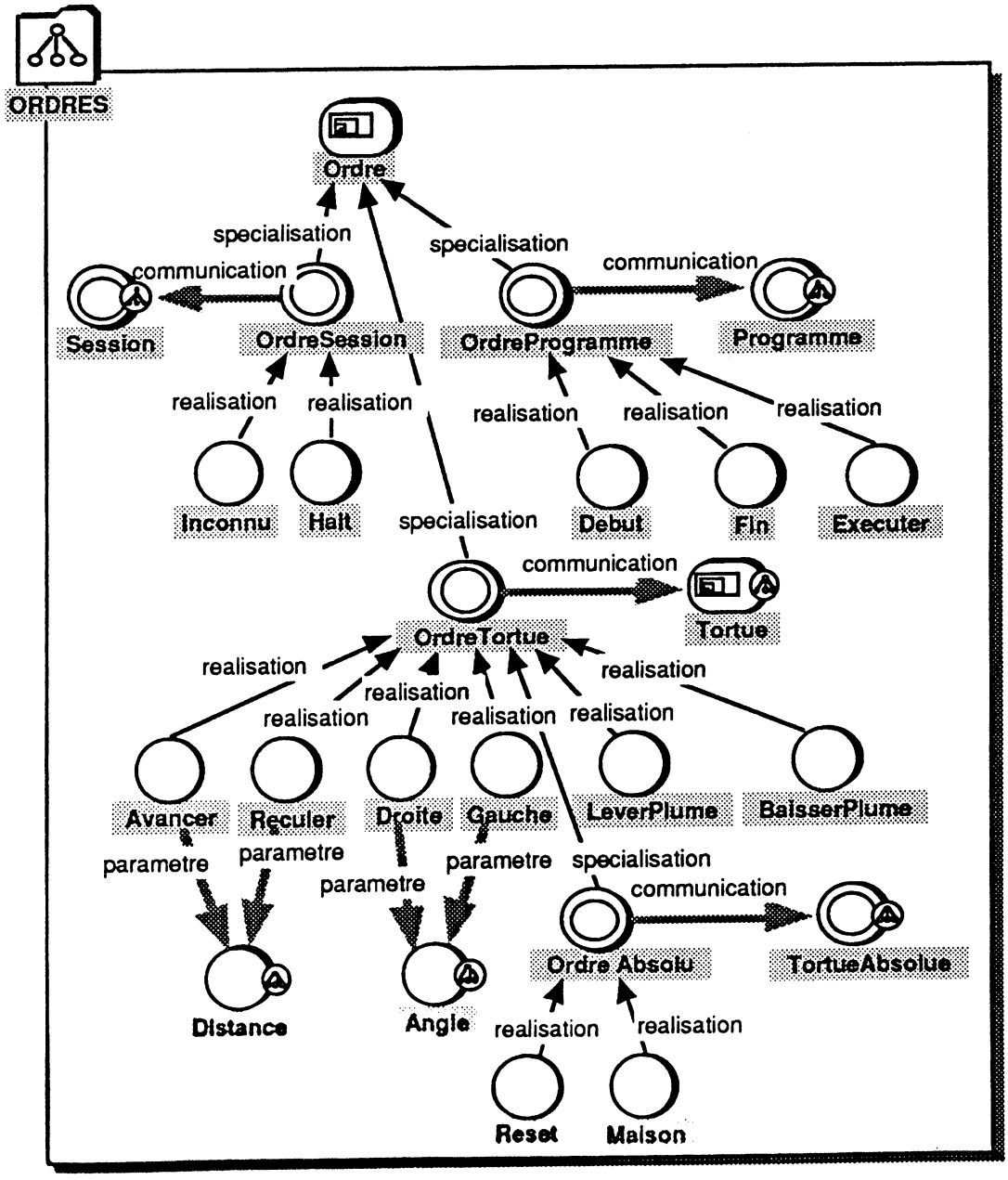


figure 18 : le domaine ordre (réalisations)

Remarque: Les ordres *avancer* et *reculer* redéfinissent le type de leur paramètre par Distance, de même pour *droite* et *gauche* avec Angle.

B.3.2. Définition des classes concrètes

Voir conception détaillée

DOMAINE SESSION

B.1. DEFINITION DE LA CLASSE RACINE

La classe Session représente l'abstraction d'une session de saisie/exécution d'ordres de la part de l'utilisateur. Cette classe représente donc l'interface utilisateur du système. Afin de pouvoir faire varier cette interface selon les besoins, nous spécifions la classe la plus générale possible.

B.1.1. MISE EN PLACE DES LIENS DE DELEGATION

Une Session communique avec un Ordre (l'ordre courant) et un contrôle (classe non introduite, chargée du décodage des ordres).

B.1.2. SPECIFICATION DE LA CLASSE

La Session sait dérouler une boucle "top-level" constituée de 3 actions: saisie de l'ordre, saisie du paramètre éventuel et exécution de l'ordre (via le contrôle).

CLASSE SESSION

SPECIFICATION
virtuelle
rôle abstraction d'un utilisateur de tortue (plus généralement d'une session de saisie/exécution d'ordres)
mot-clé exécution , ordre, session, utilisateur
invariant

INTERFACE
Top_level, Action, Set_FinSession(V:BOOLEEN),
AfficherMessage(M: STRING), Saisir_ordre: STRING, Saisir_param: INTEGER, Set_controle(C: CONTROLE)

METHODE
action Top-level
spécification lance une session utilisateur
précond
postcond FinSession = vrai
définition
Set_FinSession(faux)
Tantque non FinSession faire Action
action Action
spécification un pas de Top-Level
précond
postcond
local Ordre_saisi: STRING, param: INTEGER
définition
Ordre_saisi <- Saisir_ordre
Ordre <- Contrôle.traiter_ordre(Ordre_saisi)
si Ordre.exige_paramètre alors param <- Saisir_param; set_paramètre(param)
Contrôle.exécuter_ordre(Ordre)
action AfficherMessage(M: STRING) retardée
spécification affiche le message M en sortie (dépend de l'interface)
postcond message M affiché pour l'utilisateur

```
   fonction  Saisir_ordre: STRING retardée
      spécification acquisition d'un ordre de l'utilisateur
   fonction  Saisir_param: INTEGER retardée
      spécification acquisition d'un paramètre
   action  Set_controle(C: Contrôle)
      spécification met à jour l'attribut contrôle (décodage de l'ordre)
      définition
         Contrôle <- C
   action  Set_FinSession(V: BOOLEEN)
      spécification met à jour l'attribut FinSession
      définition FinSession <- V
ATTRIBUT
   communication
      ordre: ORDRE
      spécification ordre en cours d'exécution
      contrôle: CONTROLE
      spécification contrôle de la tortue
   utilitaire
      FinSession : BOOLEEN
      spécification si FinSession devient faux on arrête le Top_Level
FIN  CLASSE  SESSION
```

Remarques: les méthodes *Top-level* et *action* sont définies. Les méthodes *afficher_message*, *saisir_ordre* et *saisir_param* dépendent du type de l'interface et devront être réalisées dans les implantations filles de Session.

B.2. SPECIALISATIONS

Pas de spécialisations.

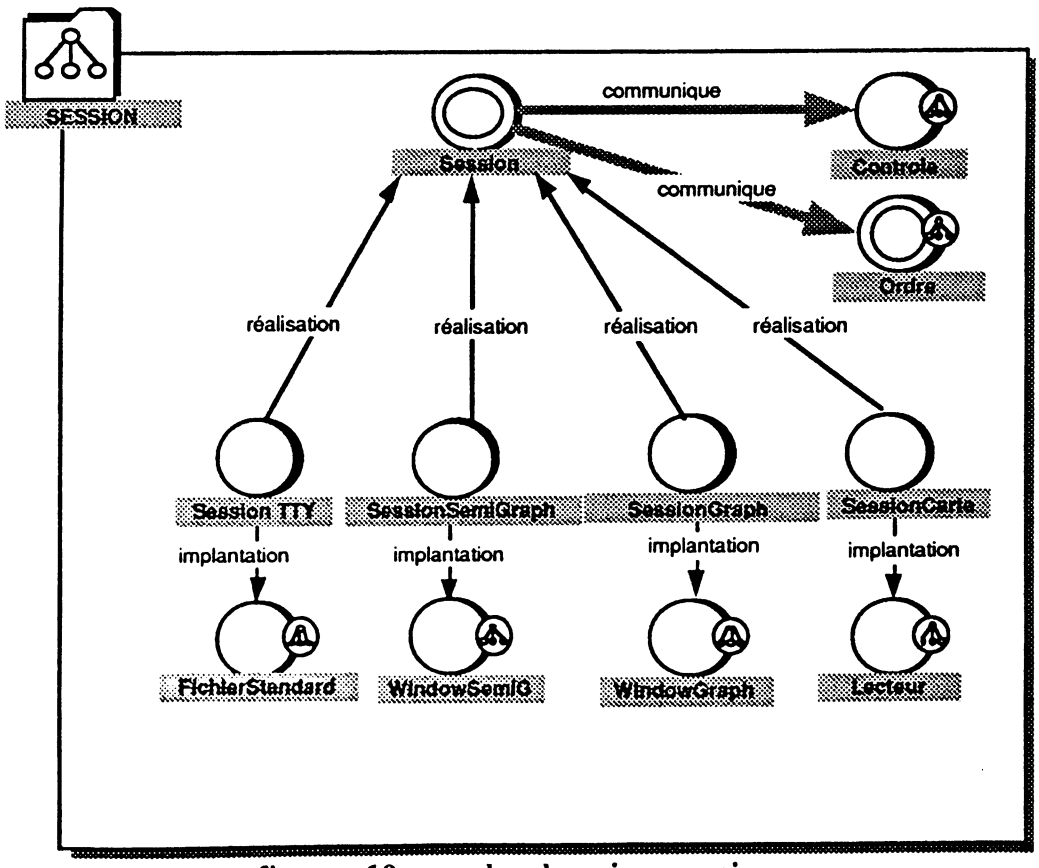


figure 19 : le domaine session

B.3. REALISATIONS

B.3.1. Création des liens de réalisation (figure 19)

Quatre réalisations sont définies. SessionTTY est la version ligne à ligne de l'interface utilisateur. SessionSemiG est une version avec menus fixes semi-graphiques. SessionGraph est la version graphique avec menus déroulant, souris, fenêtre, etc.... Enfin, SessionCarte décrit l'interface par lecteur de cartes.

Remarque: chacune des réalisations de Session hérite, en implantation, d'une classe de la bibliothèque.

B.3.2. Définition des classes concrètes

Voir étape de conception détaillée.

4.2.3. ETAPE C) RETOUR A L'ETAPE A)

A ce stade nous avons défini les domaines identifiés en A). La construction des domaines engendre de nouvelles classes, non identifiées au départ: Contrôle, Paramètre, Support, Triangle, Point, Robot, Nom, CrayonMobile, FichierStandard, WindowSemiG, WindowGraph, Lecteur.
En ce qui concerne FichierStandard, WindowSemiG, WindowGraph, Nom, Triangle et Point, nous considérons pour simplifier que ces classes sont fournies par des domaines appartenant à des bibliothèques préexistantes.
Nous laissons de coté la conception de CrayonMobile, Robot et Lecteur qui nous conduiraient à des considérations trop techniques.
Nous détaillons dans la suite Contrôle, Paramètre et Support qui donnent chacune naissance à un domaine.

DOMAINE CONTROLE

Ce domaine décrit les systèmes de pilotage de la tortue.

C.1. DEFINITION DE LA CLASSE RACINE

La classe contrôle est la racine du domaine.

C.1.1. MISE EN PLACE DES LIENS DE DELEGATION

Le contrôle communique avec une tortue (sur laquelle s'applique les ordres), un programme (dans lequel on stocke les ordres) et une session (de laquelle elle reçoit les ordres à décoder). Pour chacune de ces délégations, on utilise la classe la plus abstraite dans le domaine correspondant (cf figure 20).

C.1.2. SPECIFICATION DE LA CLASSE

La classe contrôle décrit un contrôle général d'une tortue et d'un programme. Les ordres décodables ne sont pas définis, ils le seront dans les différentes réalisations de contrôle.
La méthode *traiter_ordre* est responsable du décodage de l'ordre et de la création de l'instance d'ordre correspondante. La méthode *exécuter_ordre* sera lancée par la session et traitée par le contrôle en tenant compte du programme (mode programmable ou executable).
Les ordres inconnus sont traités comme des ordres à part entière. Leur exécution provoque l'envoi d'un message à l'utilisateur.

CLASSE CONTROLE

SPECIFICATION
virtuelle
rôle gère le pilotage de la tortue et la mise en place du programme le contrôle est l'interface entre l'utilisateur et la tortue. La création des instances d'ordre se fait dans les descendants de cette classe.

mot-clé contrôle, gestion , tortue

INTERFACE
traiter_ordre(NomOrdre:STRING): ORDRE, exécuter_ordre(O: ORDRE),
Set_Tortue(T: TORTUE), Set_Programme(P: PROGRAMME), Set_Session(U: SESSION)

METHODE
action Set_Tortue(T: TORTUE)
spécification met à jour l'attribut Tortue
définition
Tortue <- T
action Set_Programme(P: PROGRAMME)
spécification met à jour l'attribut Source
définition
Source <- P
action Set_Session(U: SESSION)
spécification met à jour l'attribut User
définition
User <- U
fonction traiter_ordre(NomOrdre:string): ORDRE retardée
spécification traite la commande utilisateur NomOrdre (décodage)
précond
postcond l'ordre est décodé, on renvoie l'ordre correspondant
fonction Traiter_autre(NomOrdre:string): ORDRE
spécification traite les autres ordres (fonction interne pour les descendants), dans ce cas tout autre ordre est inconnu.
local
inc: INCONNU
définition
inc.creer
inc.set_chef(User)
inc.set_nom(NomOrdre)
resultat <- inc
action exécuter_ordre(O:ORDRE)
spécification exécute l'ordre correspondant, tient compte du mode programmation
précond
postcond Ordre O exécuté
définition
si Source.programmation et type ordre inclus dans ORDRETORTUE alors Source.stocker(Ordre)
sinon Ordre.exécuter

ATTRIBUT
communication
Tortue: TORTUE
spécification Tortue soumise au contrôle
Source: PROGRAMME
spécification Programme à transmettre aux ordres concernés
User: SESSION
spécification communication avec l'utilisateur

FIN CLASSE CONTROLE

Remarque: la seule méthode retardée est *traiter_ordre*. Le contrôle abstrait ne sait décoder aucun ordre sauf l'ordre inconnu (*traiter_autre*). La méthode *exécuter_ordre* peut être définie indépendamment du type de l'ordre concerné. Selon que le mode est

programmable ou non, l'ordre correspondant est stocké ou exécuté. Par le polymorphisme, la méthode *exécuter* de l'ordre concret correspondant est exécutée.

C.2. SPECIALISATIONS

Pas de spécialisations. Une abstraction plus générale du contrôle peut être définie comme, par exemple, un contrôle ne gérant pas de programme (ordre debut, fin, exécuter non reconnus). Dans notre contexte, cette abstraction serait superflue puisque le programme doit être géré. En cas de réutilisation dans un autre système de pilotage interactif, une telle classe pourra cependant être créée.

C.3. REALISATIONS

C.3.1. Création des liens de réalisation

La première réalisation de Contrôle concerne la version simple de la tortue (non absolue). Les ordres *reset* et *maison* ne sont donc pas pris en compte. Une deuxième réalisation issue de la première (héritage d'adaptation) est définie pour prendre en compte ces deux ordres et piloter une tortue absolue (tortue écran par exemple).

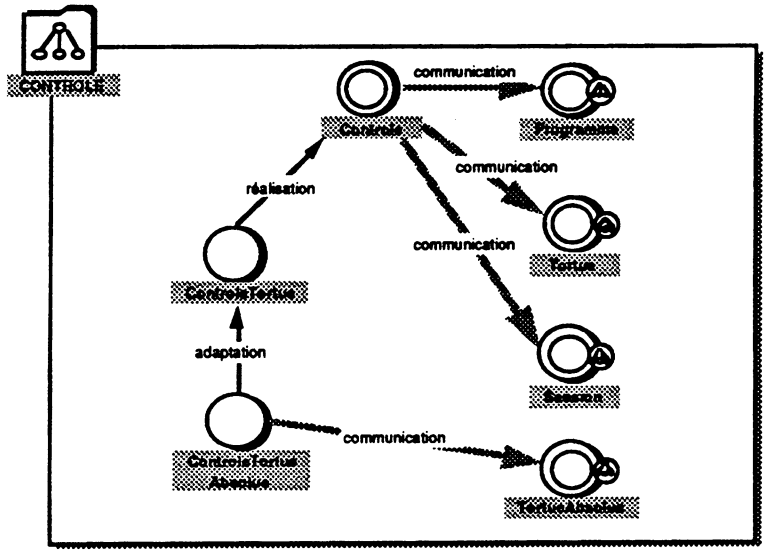


figure 20 : le domaine contrôle

C.3.2. Définition des classes concrètes

Voir conception détaillée.

DOMAINE PARAMETRE

Ce domaine décrit les paramètres des ordres avancer, reculer, gauche et droite.

C.1. DEFINITION DE LA CLASSE RACINE

La classe PARAMETRE est la racine du domaine paramètre. Un paramètre encapsule une valeur pouvant être argument d'un ordre. Il possède les méthodes *ajout*, *valeur* *set_valeur* et *RAZ*.

C.1.1. MISE EN PLACE DES LIENS DE DELEGATION

Pas de liens de délégation.

C.1.2. SPECIFICATION DE LA CLASSE

CLASSE PARAM
SPECIFICATION **virtuelle** **rôle** racine du domaine paramètre, définit un paramètre possible d'un ordre **mot-clé** ordre, paramètre **invariant**
INTERFACE RAZ, Ajout(P: comme PARAM): comme PARAM , Set_valeur(V: INTEGER), valeur: INTEGER
METHODE **action** RAZ **spécification** initialise le paramètre avec une valeur par défaut **précond** **postcond** paramètre remis a zéro **définition** valeur <- 0 **fonction** Set_valeur(V: INTEGER): INTEGER **spécification** initialise valeur **définition** Valeur <- V **fonction Ajout(P: comme PARAM): comme PARAM retardée** **spécification** Ajoute P à param
ATTRIBUT **composition** valeur : INTEGER **spécification** valeur du paramètre **FIN CLASSE PARAM**

C.2. SPECIALISATIONS

Pas de spécialisations

C.3. REALISATIONS

C.3.1. Création des liens de réalisation

Les réalisations de Paramètre sont Distance (argument de Avancer et Reculer) et Angle (argument de Droite et Gauche).

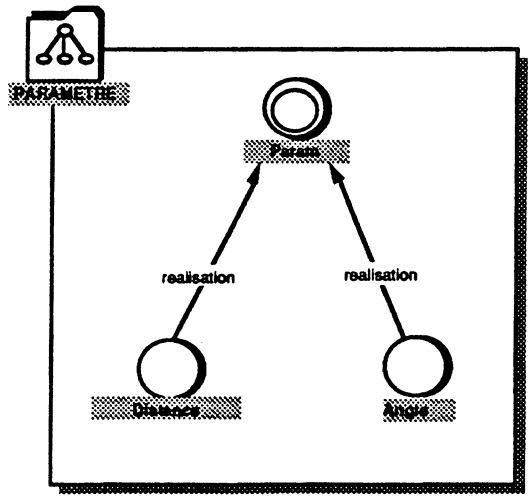


figure 21 : le domaine paramètre

C.3.2. Définition des classes concrètes
Voir conception détaillée

DOMAINE SUPPORT

Ce domaine décrit le support d'affichage nécessaire au traçage des déplacements pour une TortueEcran.

C.1. DEFINITION DE LA CLASSE RACINE

C.1.1. MISE EN PLACE DES LIENS DE DELEGATION
Pas de liens de délégation. Support est une classe fournissant des primitives graphiques réalisées par des classes concrètes descendantes.

C.1.2. SPECIFICATION DE LA CLASSE
Support fournit à une TortueGraphique des méthodes pour tracer des traits, afficher, translater et effacer des figures (dont les triangles), et une méthode pour effacer le support lui-même.

CLASSE SUPPORT
SPECIFICATION **virtuelle** **rôle** support pour l'affichage des objets graphiques **mot-clé** support, sortie graphique, figures
INTERFACE TracerTrait(P1,P2: POINT), AfficherFigure(F: FIGURE, Pos: POINT), EffacerFigure(F:FIGURE) EffacerSupport
METHODE **action** **TracerTrait(P1,P2: POINT)** **retardée** **spécification** trace un trait entre l'origine et la destination du vecteur **action** **AfficherFigure(F: FIGURE, Pos: POINT)** **retardée** **spécification** affiche la figure F en positionnant son barycentre en Pos **action** **EffacerFigure(F:FIGURE)** **retardée** **spécification** efface la figure F **action** **EffacerSupport** **retardée** **spécification** efface tous les objets graphiques du support **FIN CLASSE SUPPORT**

C.2. SPECIALISATIONS
Pas de spécialisations.

C.3. REALISATIONS
C.3.1. Création des liens de réalisation
La seule réalisation définie est la feuille graphique utilisant une "bitmap" pour l'affichage graphique et une liste chaînée de figures pour la gestion des figures. La classe figure provient de la librairie graphique (domaine des figures).

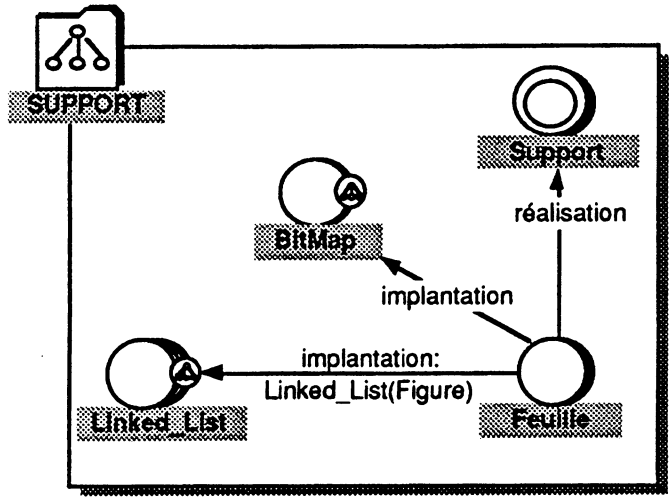


figure 22 : le domaine support

4.2.4. ETAPE D) MISE EN PLACE DES APPLICATIONS

Dans cette étape, nous devons décrire le système opérationnel final. Nous nous intéressons aux deux applications de gestion de tortues définies dans les spécifications: tortue écran et tortue robot.

APPLICATION TORTUE ECRAN

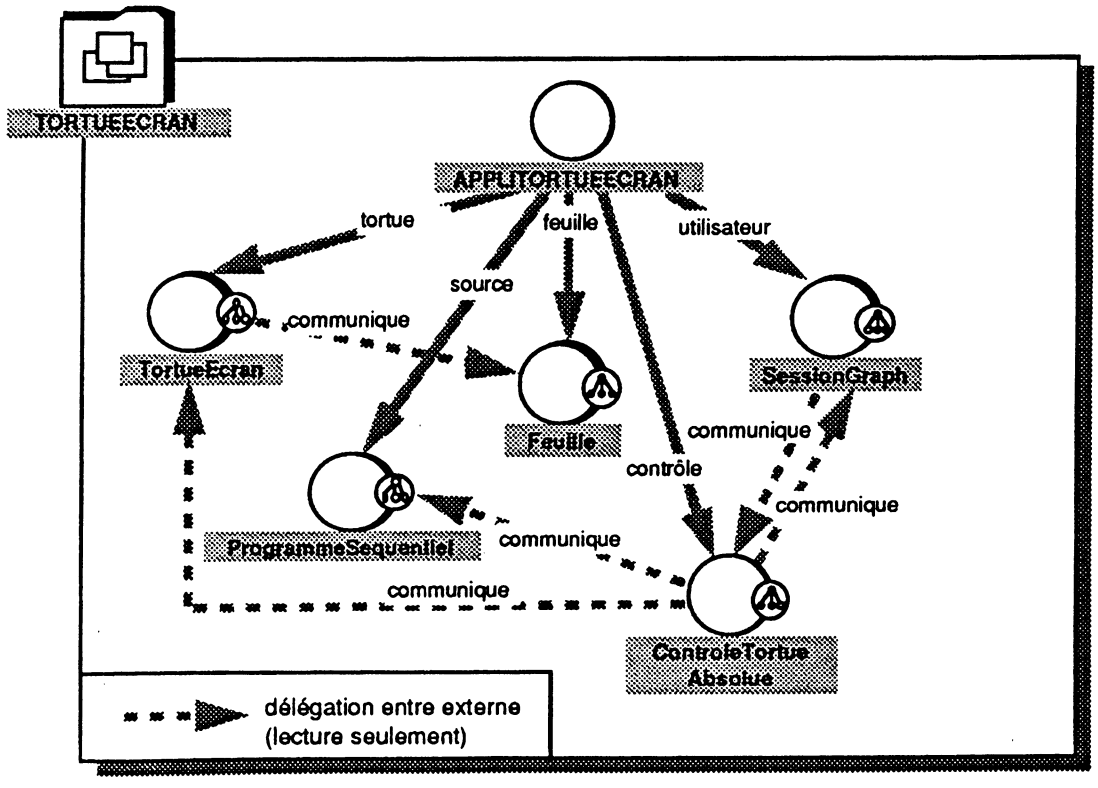


figure 23: application TortueEcran

D.1) Création de la classe racine et des classes auxiliaires
Création d'une classe racine AppliTortueEcran qui créera les instances nécessaires au bon déroulement du programme.

D.2) Définition des liens de délégation (figure 23)
L'application va être composée d'une TortueEcran, d'une Feuille (support associé à la tortue), d'un ProgrammeSequentiel, d'un ContrôleTortueAbsolue et d'une SessionGraph.

Remarque: Les relations de délégation entre classes externes ne sont présentes que pour la lisibilité. Elles ont été mises en place dans les domaines respectifs des classes origines de chaque relation (cf §2.5.4.).

D.3) Conception détaillée de la classe racine et des classes auxiliaires
Dans cette phase, on rédige la conception détaillée des nouvelles classes introduites dans l'application. On décrit, en particulier, le script de création des objets (séquencement, paramètres d'initialisation). Dans cette application, la seule nouvelle classe est la classe racine:

CLASSE APPLITORTUEECRAN
SPECIFICATION **rôle** racine de l'application TortueEcran **mot-clé** application, ordre simple, tortue écran, mono tortue **invariant**
INTERFACE Lancer
METHODE **action** Lancer **spécification** lance l'application (création et initialisation), c'est le script d'instanciation de l'application **définition** Feuille.Créer Tortue.Créer(Feuille, 3) Utilisateur.Créer Source.Créer Contrôle.Créer Contrôle.Set_tortue(Tortue) Contrôle.Set_Programme(Source) Contrôle.Set_session(Utilisateur) Utilisateur.Set_controle(contrôle) Utilisateur.Top_level
ATTRIBUT **composition** Tortue: TORTUEECRAN **spécification** tortue de l'application Feuille: FEUILLE **spécification** support d'écriture pour la tortue (écran) Utilisateur: SESSIONGRAPH **spécification** type de l'interface contrôle: CONTROLETORTUEABSOLUE **spécification** contrôle choisi (pas d'ordres composés) Source: PROGRAMMESEQUENTIEL **spécification** programme (unique) pour stocker et exécuter des séquences d'ordres (pas d'ordres composés)
FIN CLASSE APPLITORTUEECRAN

APPLICATION TORTUE ROBOT

D.1) Création de la classe racine et des classes auxiliaires

Une classe racine AppliTortueRobot est définie. Cette classe créera les instances nécessaires au bon déroulement du programme de pilotage d'une tortue robot.

D.2) Définition des liens de communication

L'application est composée d'une TortueRobot, d'un ProgrammeSequentiel, d'un ContrôleTortue et d'une SessionCarte. On peut aussi envisager de commander la tortue par une sessionTTY sur un terminal, ce qui n'induira qu'une seule modification sur la classe racine de l'application.

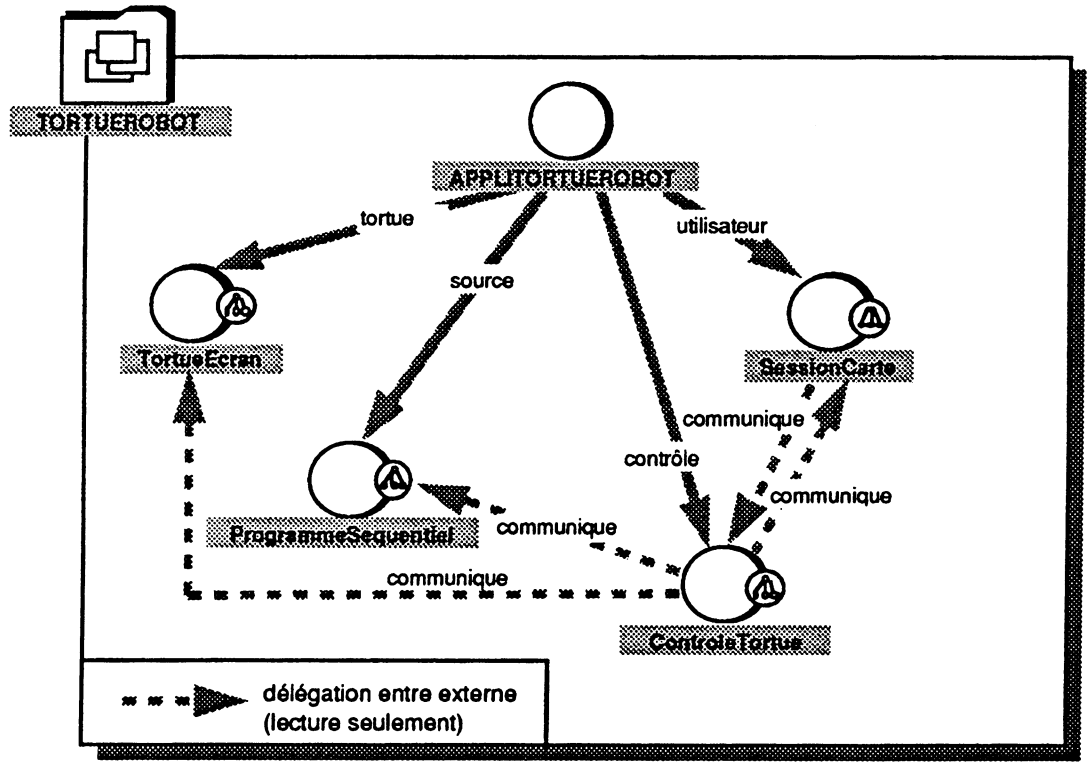


figure 24: application TortueRobot

D.3) Conception détaillée de la classe racine et des classes auxiliaires

La seule classe à définir est la classe racine:

CLASSE APPLITORTUEROBOT
SPECIFICATION **rôle** racine de l'application TortueRobot **mot-clé** application, ordre simple, tortue robot, mono tortue **invariant**
INTERFACE Lancer

METHODE
action Lancer **spécification** lance l'application (création et initialisation). Script d'instanciation. **définition** Tortue.Créer Utilisateur.Créer Source.Créer Contrôle.Créer Contrôle.Set_tortue(Tortue) Contrôle.Set_Programme(Source) Contrôle.Set_session(Utilisateur) Utilisateur.Set_controle(contrôle) Utilisateur.Top_level
ATTRIBUT **composition** Tortue: TORTUEROBOT **spécification** tortue de l'application Utilisateur: SESSIONCARTE **spécification** type de l'interface contrôle: CONTROLETORTUE **spécification** contrôle choisi (pas d'ordres composés) Source: PROGRAMMESEQUENTIEL **spécification** programme (unique) pour stocker et exécuter des séquences d'ordres (pas d'ordres composés) **FIN CLASSE APPLITORTUEROBOT**

4.2.5. ETAPE E) CONCEPTION DETAILLEE DES CLASSES CONCRETES

Dans cette étape, on reconsidère, en vue de leur réalisation, toutes les classes concrètes spécifiées et non réalisées dans les étapes B), C) et D). Pour chacune des classes, le processus de conception des classes concrètes est appliqué (cf §3.2.4.E.)

A titre d'exemple, nous donnons ici la conception détaillée des classes TORTUE_ECRAN, AVANCER et CONTROLETORTUEECRAN. Les autres classes n'offrent pas d'intérêt supplémentaire.

CLASSE TORTUEECRAN
SPECIFICATION **rôle** Réalisation de tortueecran au moyen d'un triangle isocèle, l'angle aigu indique le cap. **mot-clé** tortue, écran, triangle **invariant** sommet.angle.value ≤ 90 et sommet.angle.value ≠ 120 (l'angle du sommet du triangle est aigu et le triangle n'est pas équilatéral) et le cap est indiqué par l'angle aigu du triangle isocèle et le barycentre du triangle indique la position de la tortue
INTERFACE avancer(D : INTEGER), reculer(D : INTEGER), droite(A : ANGLE), gauche(A : ANGLE), leveplume, baisseplume, maison, reset, Cap: ANGLE, Pos : INTEGER
RENOMMAGE Créer de TRIANGLE par CreerTriangle

```
METHODE
   action creer(F: SUPPORT, Taille: INTEGER)
      postcond      Tortue créée, triangle créé, feuille attachée à la tortue et plume créée
      définition    CreerTriangle(F,Taille); Plume.Créer; Feuille <- F
                    Déplacer le triangle au centre (barycentre en 0,0)
                    Faire tourner le triangle sur son centre pour amener
                    l'angle aigu en Oy
   action avancer(D : integer) défini
      précond       D >= 0 (pour les utilisateurs uniquement)
      postcond      Pos.x = old Pos.x + Cap.cosinus et
                    Pos.y = old Pos.y + Cap.sinus
      local         Pdep, Parr: POINT
      définition    effacer le triangle
                    Pdep = position actuelle
                    Calculer Parr:
                       Parr.x = Pdep.x + D*sinus(Cap)
                       Parr.y = Pdep.y + D*cosinus(Cap)
                    translation de vecteur PdepParr du triangle
                    si non plume.haute alors tracer un trait (Parr,Pdep)
                    afficher le triangle
   action droite(A : Angle) défini
      précond       D >= 0 et D < 360
      postcond      Cap = old Cap.ajout(A)
      définition    Cap <- Cap.ajout(A)
                       effacer le triangle
                       effectuer une rotation de -A sur le triangle (sens trigonométrique)
                       afficher le triangle
   action maison défini
      spécification la tortue rentre à la maison c'est à dire en position 0,0, cap au nord
      postcond      Pos.x=0 et Pos.y=0 et Cap.val=0
      définition    P.RAZ; Cap.RAZ
   action reset défini
      spécification la tortue rentre à la maison c'est à dire en position 0,0,
                    cap au nord et la feuille de tracé est effacée
      postcond      Pos.x=0 et Pos.y=0 et Cap.val=0 et feuille effacée
                       et plume haute
      définition maison; feuille.effacer; leveplume
ATTRIBUT
      communication
         Feuille : SUPPORT
         spécification Feuille sur laquelle la tortue se déplace et trace des traits

FIN CLASSE TORTUEECRAN
```

Remarque: les classes concrètes possèdent une méthode "créer", toutes les méthodes retardées doivent être définies.

CLASSE AVANCER

```
SPECIFICATION
   rôle réalisation de ORDRETORTUE (ordre avancer)
   mot-clé ordre tortue, avancer, tortue
   invariant
INTERFACE
   exécuter, paramètre DISTANCE, set tortue(T: TORTUE),
   set parametre(P: DISTANCE), exige paramètre: BOOLEEN
   nom: STRING; set nom(N: STRING), tracer(S: OUTPUT)
```

```
METHODE
   action exécuter défini
      définition Tortue.avancer(Param.valeur)
   fonction exige_paramètre: BOOLEEN défini
      définition resultat <- vrai
   action set_paramètre(P: DISTANCE) redéfini
      définition param <- P
ATTRIBUT
   composition
      Paramètre : DISTANCE redéfini

FIN CLASSE AVANCER
```

L'ordre *avancer* exige un paramètre. Son exécution provoque l'exécution de la méthode *avancer* de la tortue avec la distance contenue dans le paramètre de l'ordre. L'attribut *Paramètre* a été redéfini par le type Distance (covariance).

CLASSE CONTROLETORTUE

```
SPECIFICATION
   virtuelle
   rôle réalisation de la version 1 du contrôle permettant de décoder les ordres debut, fin, exécuter, halt, avancer,
   reculer, gauche , droite, leverplume et baisserplume.
   mot-clé contrôle, gestion , tortue
INTERFACE
   traiter_ordre(NomOrdre:string): ORDRE, exécuter_ordre(O: ORDRE),
   Set_Tortue(T: TORTUE), Set_Programme(P: PROGRAMME), Set_Session(U: SESSION)
METHODE
   fonction traiter_ordre(NomOrdre:string): ORDRE défini
      spécification traite la commande utilisateur traiter_ordre (décodage)
      précond
      postcond l'ordre est décodé, on renvoie l'ordre correspondant
      local
         av: AVANCER, re: RECULER, dr: DROITE, ga: GAUCHE,
         le : LEVERPLUME, ba: BAISSERPLUME
         de: DEBUT, fi: FIN, ex: EXECUTER, ha: HALT
         prg: ORDREPROGRAMME,
         ses: ORDRESESSION,
         trt: ORDRETORTUE;
      définition
         selon NomOrdre =
            "avancer":    av.Créer; trt <- av
            "reculer":    re.Créer; trt <- re
            ...
            trt.Set_tortue(Tortue)
            resultat <- trt
         sinon selon NomOrdre=
            "debut":      de.Créer; prg <- de
            "fin":        fi.Créer; prg <- fi
            ...
            prg.Set_programme(Source)
            resultat <- prg
         sinon selon NomOrdre=
            "halt":       ha.Créer; ses <- ha
            ses.Set_chef(User)
            resultat <- ses
         sinon: resultat <- Traiter_autre(NomOrdre)
FIN CLASSE CONTROLETORTUE
```

Remarque: la série de choix de la méthode *traiter_ordre* est nécessaire puisque le contrôle doit décoder les ordres lui provenant de la session. Il manipule donc les ordres directement par les classes concrètes pour pouvoir créer des instances d'ordre. C'est la seule classe qui utilise les classes concrètes, les autres clients n'utilisent les ordres que par les classes virtuelles.

Dans sa version finale, le Projet Tortue contient maintenant 8 domaines et 2 applications (nous ne comptons pas les classes non définies et les domaines réutilisés).

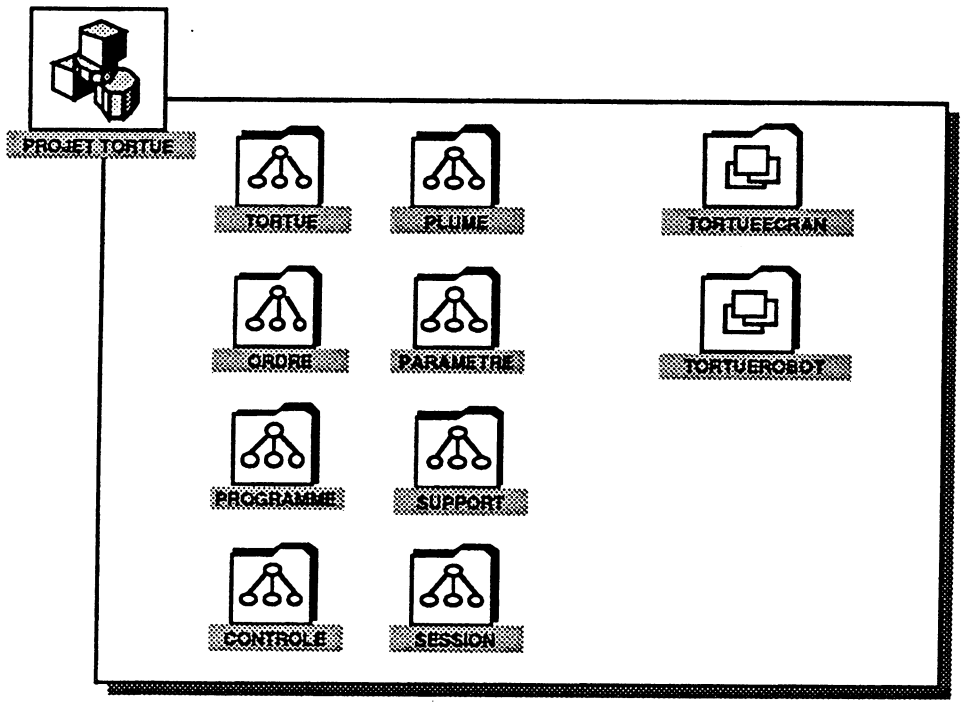


figure 25 : Le Projet TORTUE

5. MODIFICATIONS ET EVOLUTIONS

Un des intérêts majeurs d'une conception par objets est la flexibilité du logiciel obtenu. Nous indiquons, dans cette partie, comment une modification ou une évolution peut être prise en compte dans le contexte de la méthode MECANO. Nous illustrons ensuite cette approche en montrant sur l'exemple de la tortue deux évolutions et leur traduction sur la conception présentée dans la partie précédente.

5.1. Evolutions d'une conception MECANO: considérations générales

Comme nous l'avons évoqué dans le chapitre 1, la majorité des modifications et des évolutions d'un logiciel concernent les fonctionnalités. C'est d'ailleurs un des apports fondamentaux de l'approche objet que de structurer le logiciel sur les objets, parties les plus stables du système. Les modifications et les évolutions d'un logiciel suivent un processus de développement identique au processus initial. Cela signifie, entre autres, qu'une nouvelle conception doit être établie. Les entrées de cette conception seront les

spécifications de l'évolution et la conception de la première version du système. Cette activité se rapproche finalement beaucoup de la conception initiale si on considère le souci de réutilisation sous-jacent à toute conception par objet [Jacobson 91a].

Dans le cadre de la méthode MECANO, la structuration des domaines en hiérarchie de spécialisation/réalisation offre un cadre adéquat pour la mise en oeuvre des évolutions. Nous rappelons que la mise en place des domaines permet de créer des niveaux d'abstraction guidés par un triple objectif: améliorer la structuration et la clarté de la conception, rendre possible la réutilisation et anticiper les évolutions futures. L'utilisation systématique des classes abstraites dans les classes clientes, associées au polymorphisme, permet de rendre le système évolutif pendant l'exécution et donc, a fortiori, pendant le cycle de vie (cf chapitre 3, §5.2.5.).
Nous détaillons la mise en oeuvre des évolutions selon qu'il s'agit de modifier l'implantation des classes (modifications), ou bien qu'il s'agit de créer de nouvelles abstractions (évolutions).

5.1.1. Modifications

La plupart des modifications concernent des détails d'implantation. Il s'agit d'avoir une nouvelle version d'un objet. Dans ce cas, on repère le domaine contenant la classe abstraite à partir de laquelle on veut tirer une nouvelle réalisation. On doit alors créer cette nouvelle réalisation avec le processus habituel (cf §3.2.4.). Dans un deuxième temps, on s'attache à modifier toutes les classes devant manipuler directement cette réalisation (on rappelle qu'une conception MECANO tend à minimiser le nombre de telles classes). Le plus souvent, plutôt que de modifier une classe existante, il est préférable d'en créer de nouvelles versions, soit sur le même niveau que l'ancienne, soit héritant (adaptation) de l'ancienne, soit après une étape de spécialisation de la classe abstraite correspondante. Cette dernière approche sera systématiquement employée quand la nouvelle version ajoute des fonctionnalités à l'ancienne.

5.1.2. Evolutions

Un sous-ensemble des évolutions peut être traité comme les modifications. Pour les autres, il s'agit de repérer, s'ils existent, le ou les domaines concernés par l'évolution. Pour chaque domaine, on recherchera la classe virtuelle la plus spécifique pouvant convenir. On crée alors une nouvelle classe virtuelle spécialisant la première puis la ou les réalisations convenables. La définition des classes suit le processus habituel de conception. En particulier, des évolutions importantes pourront conduire à développer de nouveaux domaines.

Que ce soit pour des modifications ou des évolutions, la conception se terminera par la mise en place d'une nouvelle application et par la définition des nouvelles classes concrètes.

5.2. Exemple: évolutions de la tortue

On demande de faire évoluer le projet Tortue vu précédemment. Plusieurs modifications ou évolutions sont envisagées:

1) Ajout des macros ordres CARRE et TRIANGLE permettant de tracer un carré et un triangle. Ces ordres possèdent un paramètre qui est la longueur du coté de la figure.

2) Gestion de plusieurs Tortues, désignées par un numéro. L'utilisateur doit indiquer au préalable le nombre de tortues et le type (écran, robot) pour chacune d'elles. Il dispose ensuite de l'ordre "TORTUE", avec un paramètre entier, qui indique que les ordres

suivants doivent être exécutés par la tortue de numéro donné (jusqu'au prochain ordre TORTUE)

3) Autorisation, dans les constructions de programmes, de boucles du type "Répéter X fois". L'utilisateur donne l'ordre REPETER suivi d'un paramètre entier, puis une liste d'ordres suivie de l'ordre FINREPETER.

4) Construction d'une application pour tester la tortue écran ne nécessitant pas d'écran graphique. Pour cela, on désire tracer les déplacements de la tortue sur un écran ligne à ligne simple. Par exemple, la séquence d'ordres: reset, baisserplume, avancer(20), gauche(90), avancer(12) provoquera la trace suivante:

```
écran effacé
tortue en 0,0
cap à 0
plumebaissee
tracertrait de 0,0 à 0,20
tortue en 0,20
cap à 270
tracertrait de 0,20 à -12,20
```

Nous traitons, dans la suite, les deux premières évolutions. Les versions 3) et 4) sont laissées en exercice.

5.2.1. Macros-ordres TRIANGLE et CARRE

5.2.1.1. Ajouts de nouvelles réalisations

L'ajout de ces deux ordres ne modifie pas la classe Tortue puisqu'il peuvent être exprimés avec les méthodes connues de cette classe. Par exemple, un carré de coté **d** correspond à la séquence d'ordres suivante:

```
avancer d; droite 90
avancer d; droite 90
avancer d; droite 90
avancer d; droite 90
```

Pour prendre en compte ces deux ordres, il faut créer deux classes CARRE et TRIANGLE et les ajouter au domaine des ordres. Le rattachement se fera au niveau de la classe ORDRETORTUE qui est la classe abstraite la plus générale possible pour ces deux ordres (ils s'appliquent à une tortue quelconque). Ceci conduit au nouveau domaine ORDRE suivant:

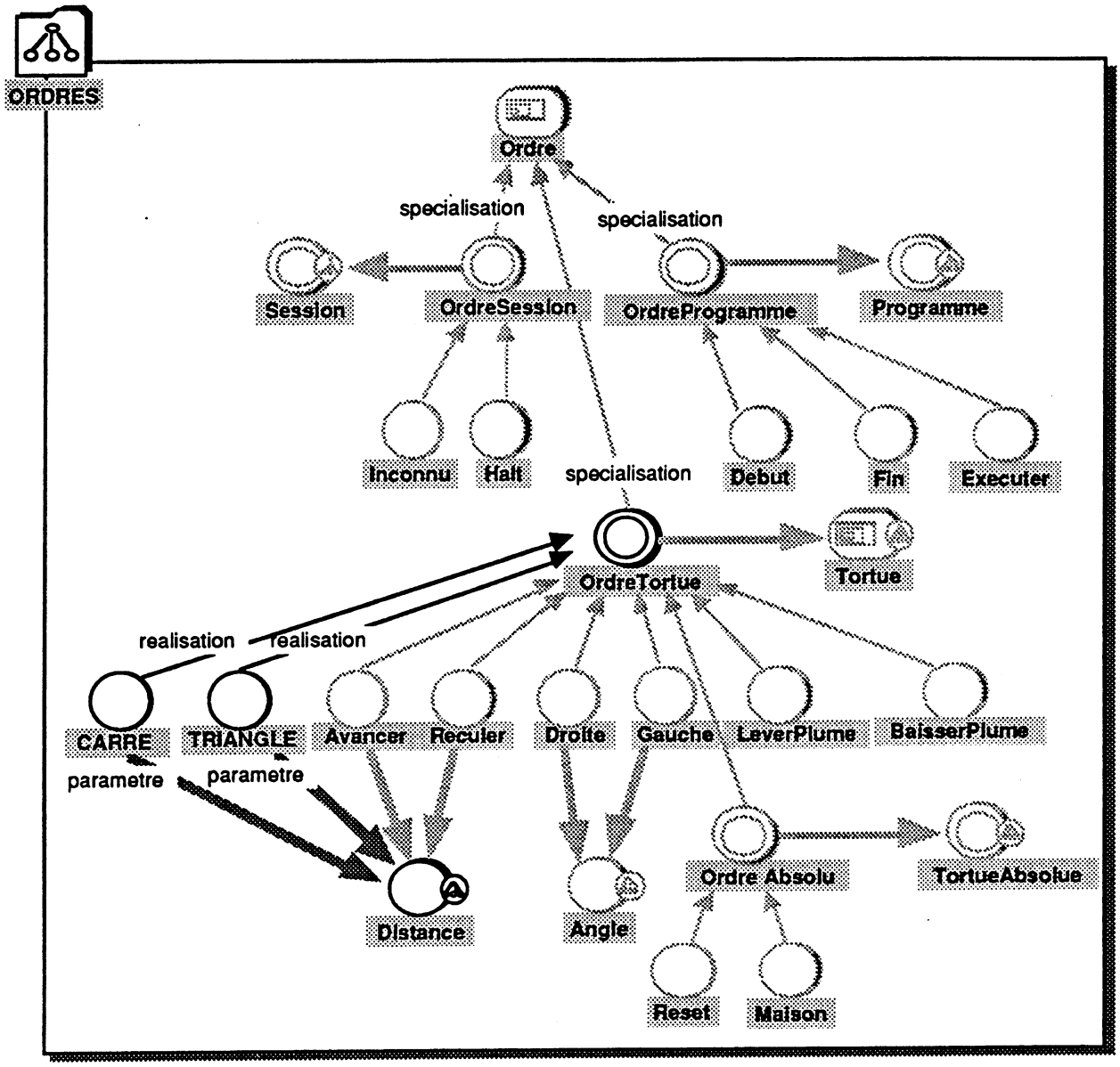


figure 26 : DOMAINE ORDRE - version 2

A titre d'exemple nous donnons ici la conception détaillée de la classe CARRE:

CLASSE CARRE
SPECIFICATION **rôle** réalisation de ORDRETORTUE (ordre carré), évolution des ordres initiaux, trace un carré de coté Paramètre **mot-clé** ordre tortue , carré **invariant**
INTERFACE exécuter, paramètre: DISTANCE, set_tortue(T: TORTUE), set_paramètre(P: DISTANCE), exige_paramètre: BOOLEEN

nom : STRING, set_nom(N: STRING), tracer(S: OUTPUT)
METHODE **action** exécuter **défini** **postcond** Tortue.plume.basse => carré tracé **local** A: angle **définition** A.créer(90) Répéter 4 fois Tortue.avancer(Param.valeur); Tortue.droite(A) **fonction** exige_paramètre: BOOLEEN **défini** **définition resultat** <- vrai
ATTRIBUT **composition** Paramètre : DISTANCE **redéfini** **FIN CLASSE CARRE**

5.2.1.2. Modifications des classes manipulant les nouvelles réalisations

Il faut définir une nouvelle classe Contrôle, apte à décoder et à créer ces nouveaux ordres. On a le choix entre modifier la classe ControleTortue initiale ou bien créer une nouvelle adaptation. Nous choisissons la deuxième solution en créant une nouvelle adaptation de ControleTortueEcran appelée ContrôleOrdreComposé. Le nouveau domaine Contrôle est représenté dans figure 27.

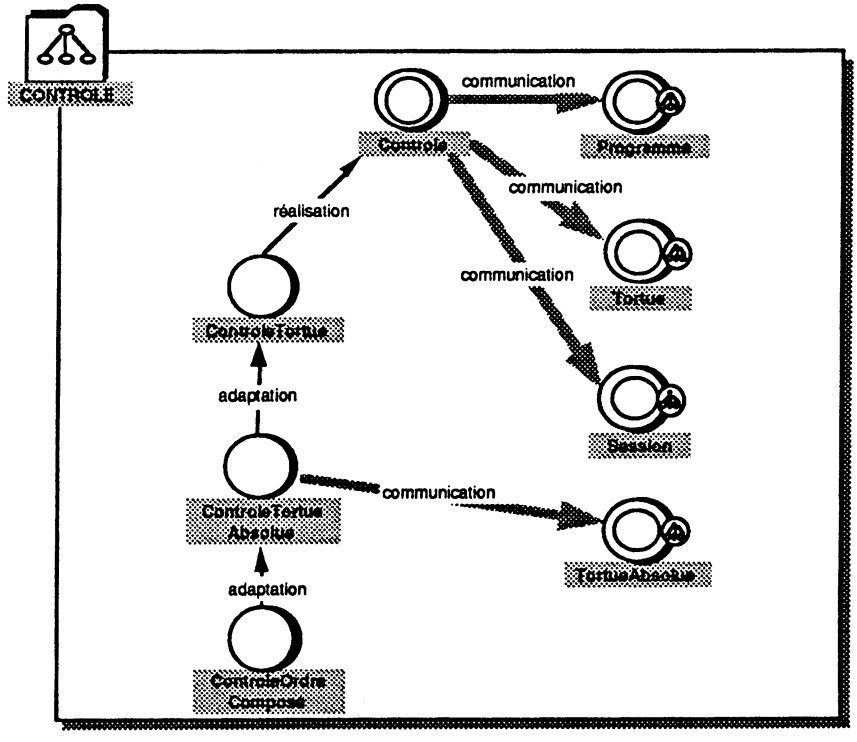


figure 27: Domaine contrôle - version 2

La conception détaillée de la classe ControleOrdreComposé est la suivante:

CLASSE CONTROLEORDRECOMPOSE
SPECIFICATION **rôle** adaptation de ContrôleTortueAbsolue pour prendre en compte l'extension Carré et Triangle **mot-clé** contrôle, version2, Carré, Triangle
INTERFACE traiter_ordre(NomOrdre:string): ORDRE, exécuter_ordre(O: ORDRE) Set_Tortue(T: TORTUEABSOLUE), Set_Programme(P: PROGRAMME), Set_Session(U: SESSION)
RENOMMAGE Traiter_autre **de** CONTROLETORTUEABSOLUE **renommé** Traiter_autre_initial
METHODE **fonction** Traiter_autre(NomOrdre:string): ORDRE **redéfini** **spécification** traite l'ordre NomOrdre comme un carré, un triangle ou un ordre inconnu appelle traiter_autre_initial si l'ordre est réellement inconnu **local** car: CARRE, tri: TRIANGLE, trt: TORTUEABSOLUE **définition** selon NomOrdre = "carré": car.creer; trt <- car "triangle": tri.creer; trt <- tri trt.Set_tortue(Tortue) **resultat** <- trt sinon: **resultat** <- Traiter_autre_initial(NomOrdre)
ATTRIBUT **communication** Tortue: TORTUEABSOLUE **redéfini**
FIN CLASSE CONTROLEORDRECOMPOSE

Remarque: la prise en compte des nouveaux ordres se fait en redéfinissant la méthode *Traiter_autre*. On renommera préalablement cette méthode, afin de conserver l'accès à l'ancienne version pour traiter l'ordre inconnu.

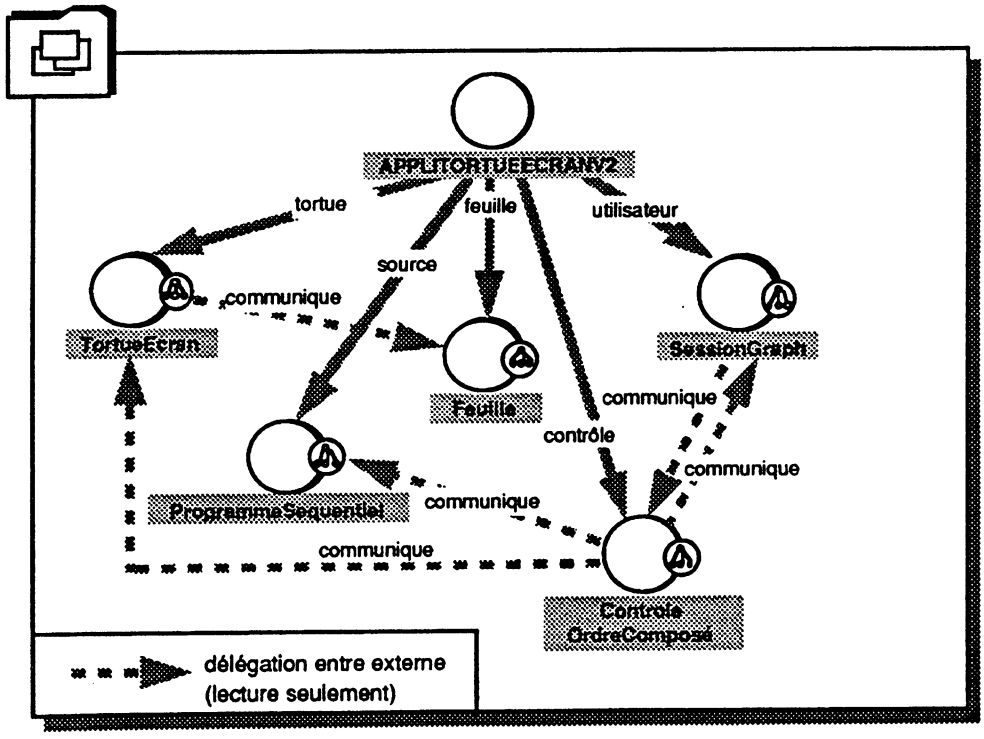


figure 28 : application avec ordres composés

5.2.1.3. Création de l'application (figure 28)

Une nouvelle application est créée pour prendre en compte la nouvelle version du contrôle.

5.2.2. Gestion de plusieurs tortues

Pour contrôler plusieurs tortues, il faut disposer d'une structure permettant de stocker et de retrouver les tortues par leur numéro. Nous choisissons un tableau.

5.2.2.1. Ajouts de nouvelles réalisations

Il faut ajouter un nouvel ordre: NOMTORTUE, avec un paramètre entier (classe distance ou création d'une nouvelle réalisation de paramètre). La classe NOMTORTUE est rattachée à la racine du domaine des ordres. En effet, cet ordre ne s'applique ni à une tortue, ni à un programme, ni à une session. Il s'adresse à un contrôle, pour lui indiquer la nouvelle tortue à prendre en compte. Le domaine Ordre devient:

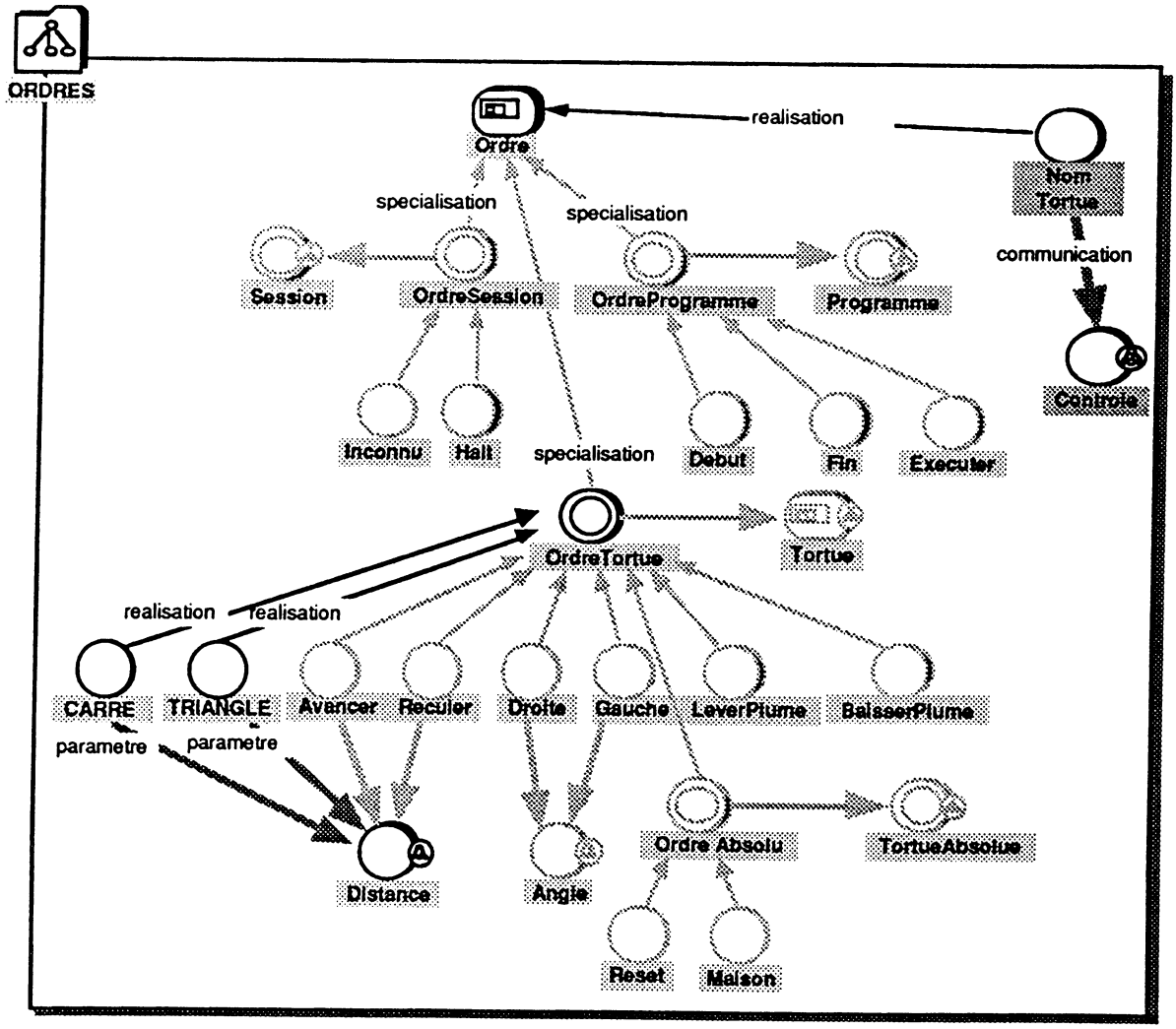


figure 29 : DOMAINE ORDRE - version 3

La classe NOMTORTUE est définie de la façon suivante:

CLASSE NOMTORTUE

SPECIFICATION
rôle réalisation de ordre, choix d'une tortue pour les ordres suivants
mot-clé ordre, multitortue
invariant .

INTERFACE
exécuter, paramètre: PARAM, exige_paramètre: BOOLEEN,
set_paramètre(P:INTEGER), set_controle(C: CONTROLE)
nom : STRING , set_nom(N: STRING), tracer(S: OUTPUT)

METHODE
action set_controle(C: CONTROLEMULTITORTUE)
définition Contrôle <- C
action exécuter **défini**
définition Contrôle.Set_Tortue_num(paramètre.valeur)
(* demande au contrôle de sélectionner la tortue numéro paramètre.valeur comme tortue courante *)
action tracer(S : OUTPUT)
spécification affiche l'ordre sur la sortie S (écran, imprimante, etc) permet de tracer l'exécution des ordres
définition S.affiche("Sélection de la tortue numéro: ", paramètre.valeur)
fonction exige_paramètre **défini**
définition resultat <- vrai

ATTRIBUT
communication
Contrôle : CONTROLEMULTITORTUE
spécification contrôle est le contrôle auqeul l'ordre doit transmettre le numéro de la nouvelle tortue courante
composition
Nom: STRING = "TORTUE"
paramètre : DISTANCE **redéfini**

FIN CLASSE NOMTORTUE

5.2.2.2. modifications des classes manipulant les nouvelles réalisations

Il faut définir un nouveau contrôle capable de gérer l'ordre "tortue"et un contexte multi-tortue. En particulier, ce sera le contrôle qui connaîtra à tout instant la tortue courante à utiliser et qui saura rechercher une tortue par son nom. Les autres classes (dont celles du domaine tortue) ne sont pas modifiées. Une nouvelle version du contrôle est créée par adaptation de ControleTortue (figure 30).

La classe ControleMultiTortue communique avec une tortue (la tortue courante) et un tableau de tortues (instanciation de la classe générique TABLEAU de la librairie par la classe TORTUE). Ce tableau pourra contenir indifféremment des tortues écrans et des tortues robots. Il est rempli initialement par la classe racine de l'application et est communiqué au contrôle lors de sa création.

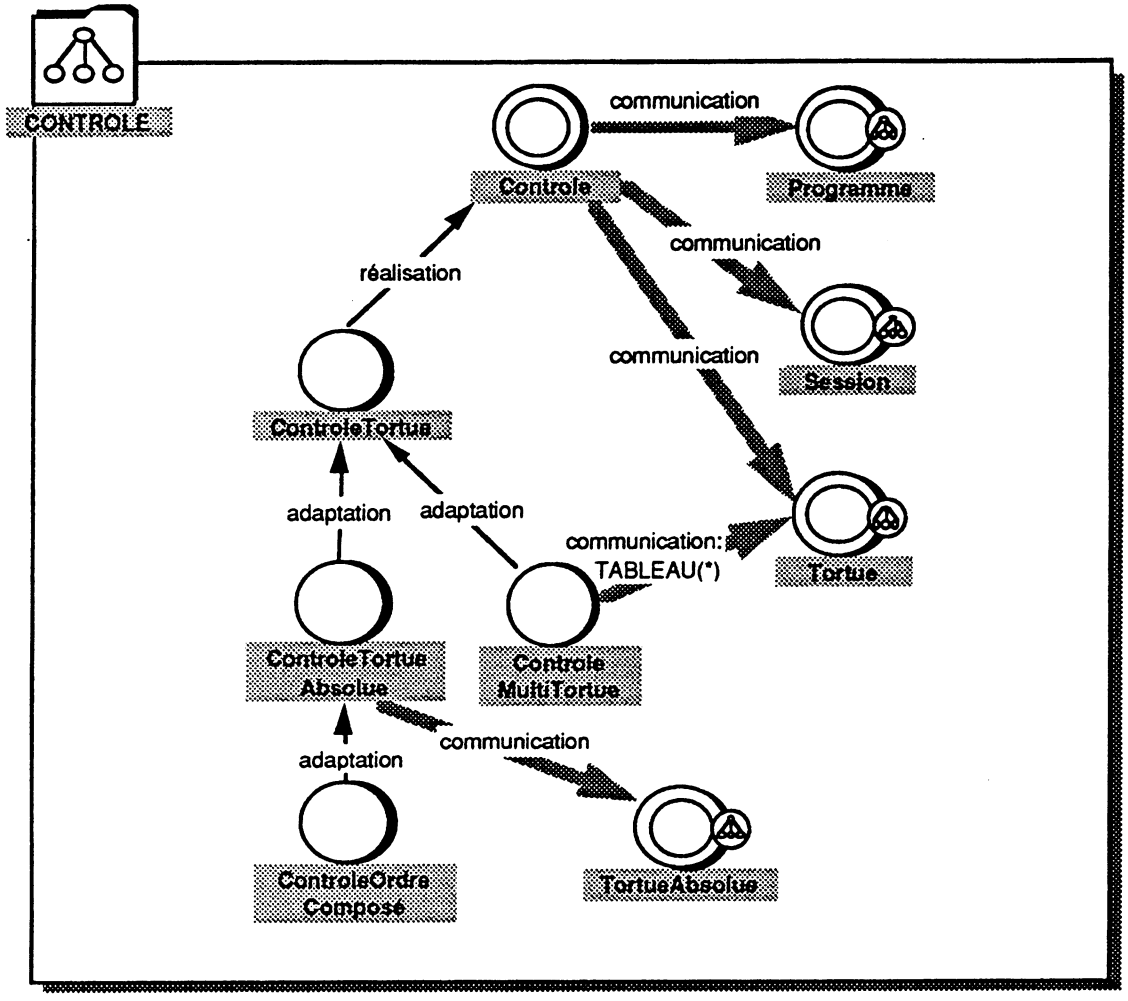


figure 30 : Domaine contrôle - version 3

CLASSE CONTROLEMULTITORTUE
SPECIFICATION **rôle** adaptation du contrôle pour prendre en compte N tortues au lieu d'une seule **mot-clé** contrôle, tortues multiples, Tortue écran
INTERFACE traiter_ordre(NomOrdre:string): ORDRE, exécuter_ordre(O: ORDRE) Set_Tortue(T: TORTUEECRAN), Set_Programme(P: PROGRAMME) Set_Ens_Tortue(E: TABLEAU(TORTUEECRAN)), Set_Tortue_Num(P: INTEGER), Set_Session(U: SESSION)
RENOMMAGE Traiter_autre **de** CONTROLE **renommé** Traiter_autre_initial
METHODE **action** Set_Ens_Tortue(E: TABLEAU(TORTUEECRAN)) **définition** Ens_Tortue <- E **action** Set_Tortue_Num(P: INTEGER) **spécification** sélectionne la tortue Numéro P qui devient la tortue courante (pour les ordres suivants) **précond** P <= Ens_Tortue.taille **définition** Set_Tortue(Ens_Tortue.get(P))

```
fonction Traiter_autre(NomOrdre:string): ORDRE redéfini
    spécification traite l'ordre NomOrdre comme l'ordre "tortue" (NOMTORTUE)
    appelle traiter_autre_initial si l'ordre est réellement inconnu
    local
        Nom: NOMTORTUE
    définition
        si NomOrdre = "tortue" alors Nom.creer; Nom.Set_controle(Courant)
        (* quand l'ordre connaîtra son paramètre, il mettra à jour la tortue correspondante dans contrôle *)
        resultat <- Nom
        sinon: resultat <- Traiter_autre_initial(NomOrdre)
ATTRIBUT
  communication
    Ens_tortue : TABLEAU(TORTUEECRAN)

FIN CLASSE CONTROLEMULTITORTUE
```

Remarque: l'attribut *Tortue* de Contrôle fait référence à la tortue courante à tout moment. L'attribut Ens_tortue maintient le tableau de tortues afin de changer de tortue courante après un ordre NomTortue.

5.2.2.3. Création de l'application

La nouvelle application est la suivante:

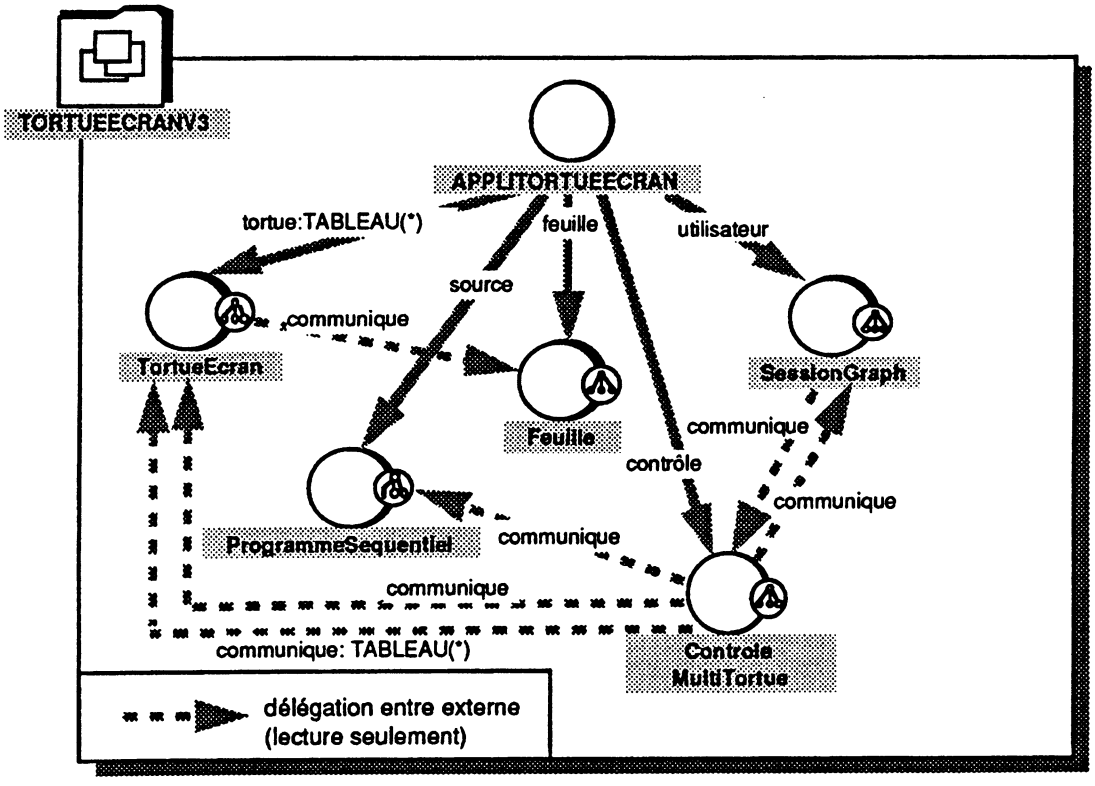


figure 31 : application avec plusieurs tortues

Les deux extensions, que nous venons de traiter, montrent la flexibilité de la conception obtenue. En ce qui concerne la réutilisabilité, il est difficile d'être démonstratif car cela implique l'existence de composants déjà construits.

On peut quand même se poser la question de la réutilisabilité de la conception de la tortue dans d'autres contextes. Sans rentrer dans les détails, un certain nombre de domaines et de classes pourront être réutilisées. Par exemple, un système de pilotage de robot programmable réutilisera sans doute les domaines Programme, Contrôle et Session, ainsi que les classes Ordre, Ordre_programme et Ordre_Session. Une application graphique peut réutiliser sans changement les classes Tortue_Ecran, Support et PlumeSimulée. Un interprète de programme Basic pourra repartir des domaines Programme, Ordre, Session et Contrôle (après spécialisation).

6. IMPLANTATION DANS UN LANGAGE DE PROGRAMMATION ET RESTRUCTURATION

6.1. Programmation d'une conception MECANO

La conception détaillée des classes MECANO (c'est à dire le codage des méthodes) est définie par un pseudo code ou la langue naturelle.
En ce qui concerne le passage de la conception MECANO au codage dans un langage de programmation, le problème est différent selon que l'on considère un langage orienté-objet ou non.

Dans le cas d'un langage orienté-objet, une traduction en EIFFEL ne posera pas de problème puisqu'on ne change pas de modèle (cf figure 32). Les concepts présents dans EIFFEL sont présents dans MECANO. On peut d'ailleurs parfaitement utiliser ce langage dans la conception détaillée. Le langage de description de classe proposé par MECANO s'inspire largement d'EIFFEL tout en étant moins formel. Les concepts de MECANO absents d'EIFFEL seront transformés éventuellement en commentaires. Par exemple, l'héritage est détypé et remplacé par l'héritage standard.
En règle générale, le passage de MECANO à la programmation s'accompagnera d'une perte de sémantique. Ceci n'a pas d'incidence puisque la cohérence et les règles de la méthode ont été assurées lors de la phase de conception.
Par contre, il faut que le langage soit assez riche pour pouvoir supporter tous les mécanismes exigés par MECANO: masquage d'information, héritage multiple, redéfinition, généricité. Si le langage ne possède pas ces concepts, il est parfois possible de les simuler. Par exemple, on peut simuler la généricité par l'héritage et réciproquement [Meyer88 pp 409-418]. La simulation des concepts inexistants doit cependant être accompagnée d'une très grande rigueur et de règles de codage précises.
L'absence des autres concepts, tels que la spécification, les assertions et les classes virtuelles, sera moins préjudiciable et il suffira d'introduire en commentaire les informations non codables. Par exemple, les méthodes retardées des classes virtuelles seront définies avec un corps vide (begin end).

Les langages non orienté-objets sont naturellement moins adaptés. Ne possédant pas l'héritage, le codage des domaines est impossible. Nous renvoyons le lecteur à [Rumbaugh 91] et [Henderson-Sellers 90] pour une étude plus approfondie des possibilités d'implantation d'une conception par objet dans un langage classique. On peut en citer les inconvénients majeurs (cf figure 32):

• Problème de traduction: la perte d'information due à la traduction entre le modèle objet et le modèle fonctionnel risque de compliquer l'implantation et de la rendre illisible.

• La COO si elle est indépendante de tout environnement d'implantation spécifique n'est pas indépendante du **paradigme** objet [Korson 90]. Pour implanter une COO dans un

langage procédural classique, on devra transformer la conception obtenue en conception basée objet: suppression de l'héritage et du niveau classe. Cette suppression, selon les langages, devra s'accompagner de conventions rigoureuses de programmation, difficiles à faire respecter.

• Problème de "reverse engineering": la consistance entre le modèle d'implantation et le modèle de conception n'est plus assurée. En conséquence, le cycle de remontée Implantation -> Conception n'étant plus possible, l'évolutivité et la maintenabilité sont compromises.

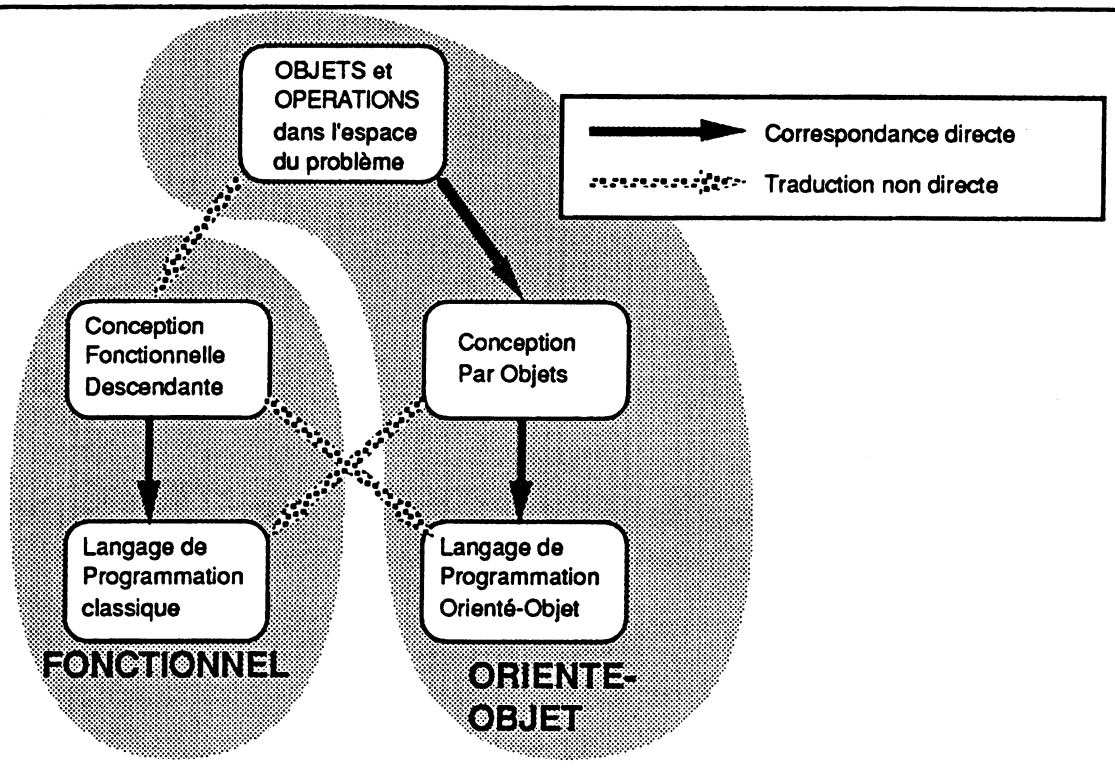


figure 32: Comparaison développement fonctionnel et orienté-objet

6.2. Restructuration

MECANO est une nouvelle méthode de conception. A ce titre, on peut s'intéresser aux problèmes de restructuration de logiciels non conçus avec MECANO. Nous nous limitons aux programmes orienté-objet à travers l'exemple de la bibliothèque de classes EIFFEL.

L'environnement du langage EIFFEL fournit une bibliothèque conséquente de classes prêtes à utiliser. Il nous a paru intéressant d'effectuer une restructuration de certaines parties de cette bibliothèque dans le modèle MECANO. Cette expérience a un double-intérêt: premièrement évaluer les possibilités de restructuration en MECANO de programmes orienté-objet classiques, ensuite évaluer la qualité de la bibliothèque à travers le modèle MECANO qui peut être vu comme un modèle plus riche que celui d'EIFFEL (et réciproquement évaluer les concepts MECANO à partir d'une bibliothèque commerciale validée).

Nous avons choisi les structures de données qui constituent une partie assez riche. La première étape est d'essayer de structurer l'ensemble de classes reliées par l'héritage en affectant un type à chaque relation mère-fille. Ensuite, nous identifions et répartissons les arbres de spécialisation dans différents domaines.

6.2.1. Etat actuel de la partie structures de données de la bibliothèque

La partie structures de données de la bibliothèque EIFFEL est représentée dans la figure 33 en utilisant le formalisme MECANO de représentation des classes (virtuelle, concrète, générique). Dans un premier temps, l'héritage est représenté par des flèches sans étiquette (héritage standard).

6.2.2. Restructuration en MECANO

6.2.2.1. Typage des liens

Les liens d'héritage doivent être typés en utilisant la taxonomie de MECANO. Les contraintes sont utilisées à l'envers comme critères de la taxonomie. Par exemple, un lien d'héritage entre une classe concrète et une classe virtuelle sera sans doute une réalisation si les interfaces sont les mêmes. Un lien entre deux classes virtuelles telles que l'interface de l'une est incluse dans l'interface de l'autre (classe mère) est un lien de spécialisation, etc....

6.2.2.2. Création des domaines

Les domaines repérés sont créés en prenant comme classe principale la racine du sous-arbre de spécialisation. Les classes reliées par des relations de spécialisation, de réalisation et d'adaptation sont introduites dans le domaine comme classes internes. Les classes reliées aux classes internes par des liens d'héritage (comportement, implantation) ou de délégation sont introduites comme classes externes.
La figure 35 montre le domaine DISPENSER qui a pour spécialisations Stack, Queue et PR_queue. Chacune des ces spécialisations est ensuite réalisée par une classe concrète (Linked_stack, Fixed_stack, SLP_queue ...). On remarquera que Fixed_stack ne vérifie pas les contraintes de MECANO car elle exporte deux méthodes de plus que sa classe mère (*duplicate* et *max_size*). Ceci peut être corrigé à l'aide d'une spécialisation intermédiaire.

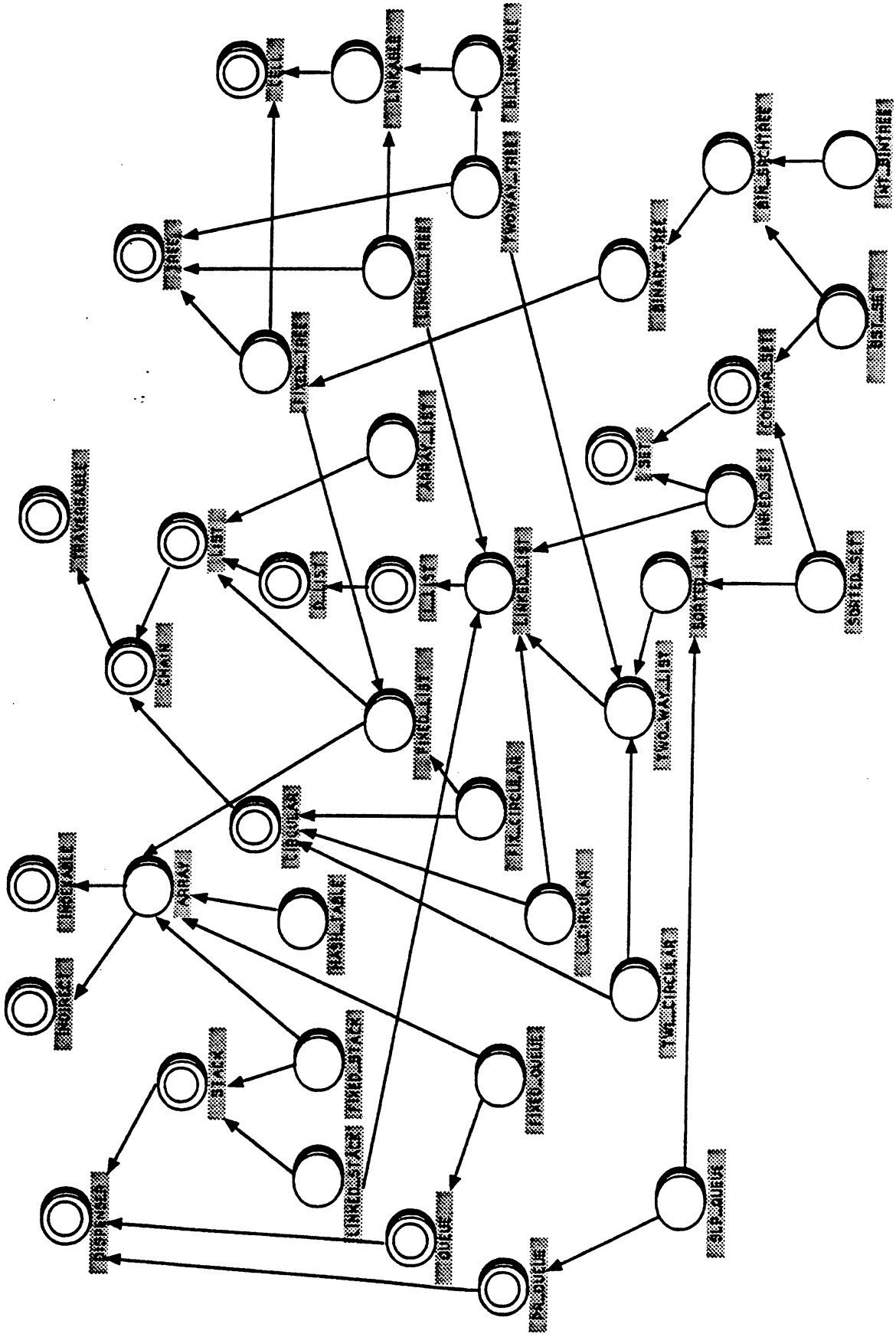


figure 33
bibliothèque des structures de données EIFFEL version 2.3 [ISE 90a]

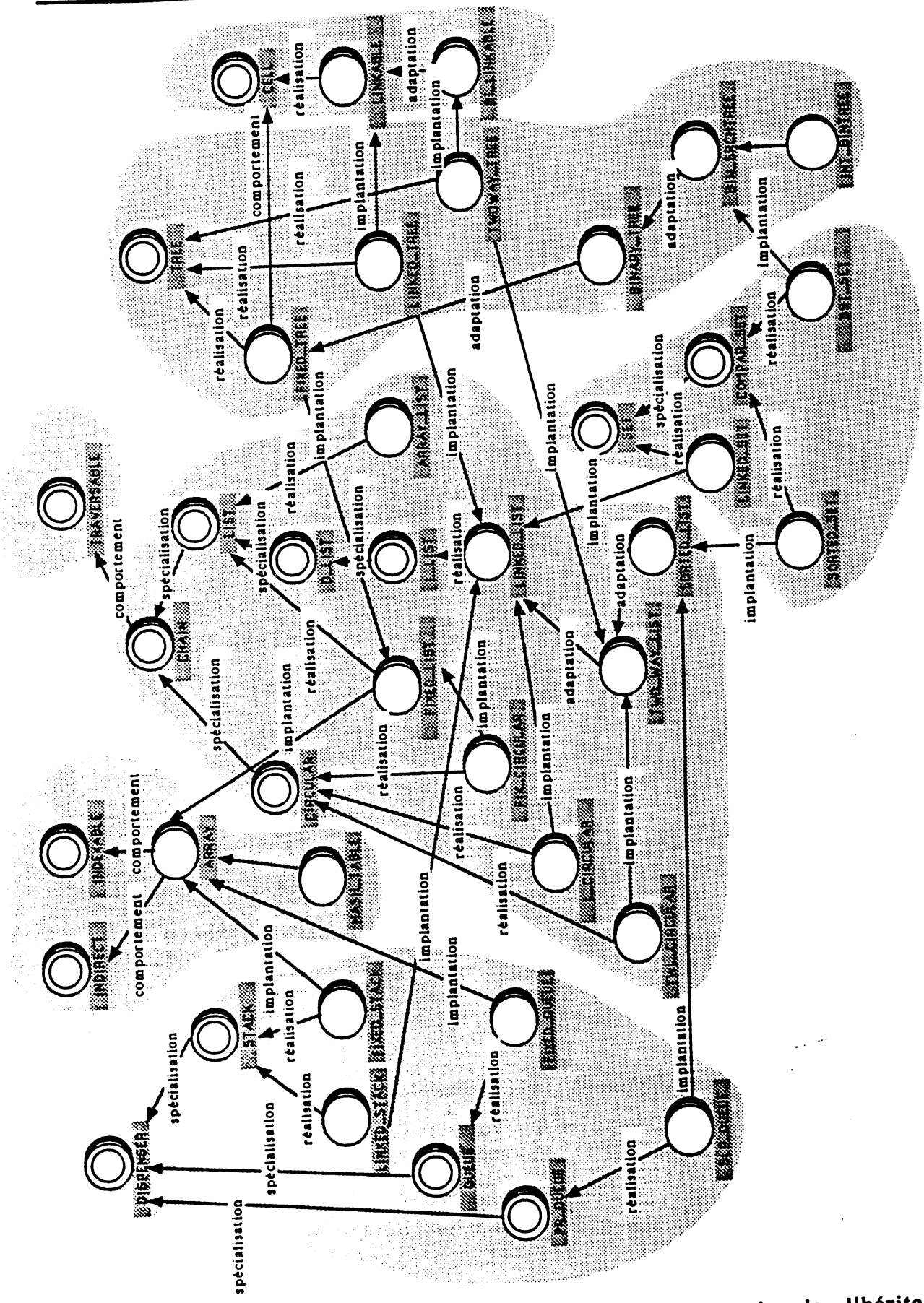


figure 34: restructuration en domaines et taxonomie de l'héritage

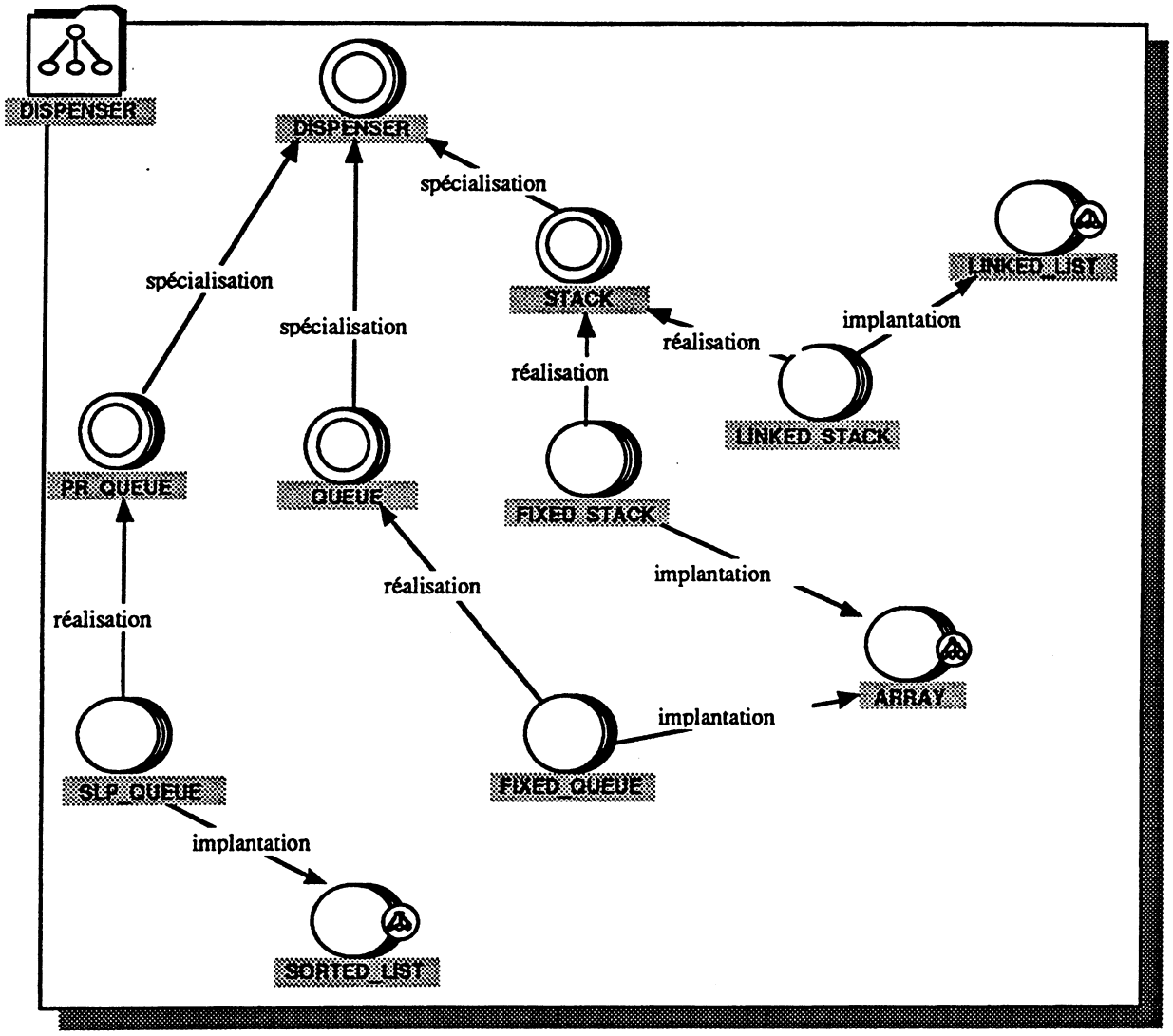


figure 35: le domaine Dispenser avec les liens externes

Dans le domaine CHAIN, la restructuration est relativement facile. Comme dans le domaine Dispenser, certaines réalisations ajoutent des caractéristiques (*extend* pour Array_list). On remarquera que les structures circulaires utilisent d'autres réalisations du domaine (Linked_list, Two_way_list et Fixed_list) avec un héritage d'implantation. Nous n'avions pas imaginé une telle situation en présentant la taxonomie de l'héritage. Elle n'entre cependant pas en conflit avec les règles de MECANO.

L'application de MECANO aux domaines des listes améliore très nettement la lisibilité. La représentation de ce domaine, privée des relations d'implantation et des liens externes, en est une illustration (figure 36)[1].
On notera que la réalisation Sorted_list de Linked_list ne vérifie pas les contraintes de la relation de réalisation puisqu'elle exporte moins de méthodes que sa classe mère. Une spécialisation intermédiaire de la classe List devra être faite pour résoudre ce problème.

---

[1] Une telle "vue de conception" pourra être produite sur demande par le poste de conception (cf chapitre 5).

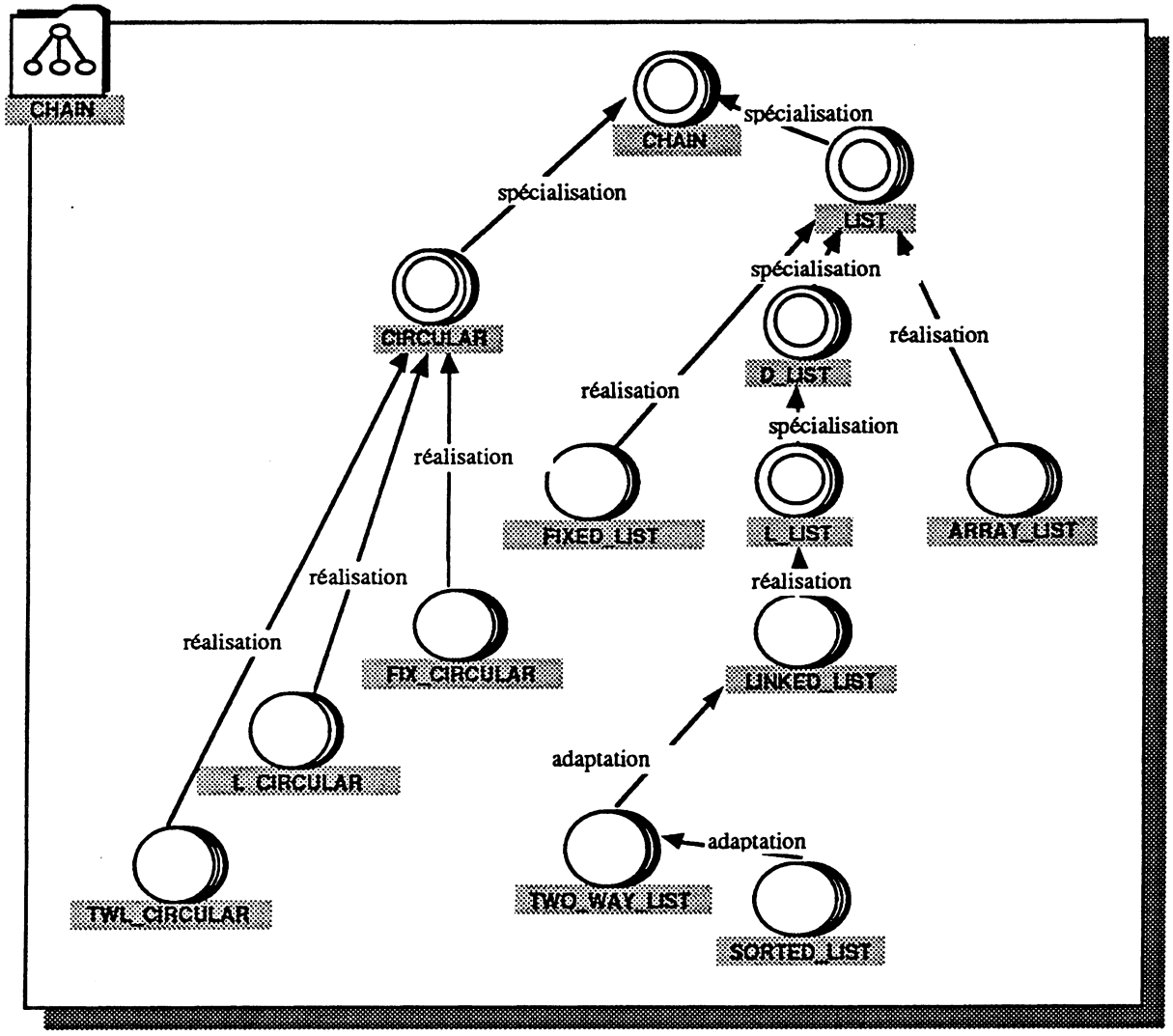


figure 36: domaine CHAIN sans lien externe et sans relation d'implantation

6.2.3. Bilan

6.2.3.1. Adéquation du modèle MECANO

Le modèle MECANO a pu être appliqué de façon quasi systématique. Les problèmes de violation de contraintes peuvent être résolus, dans la plupart des cas, en ajoutant une spécialisation virtuelle. Il ressort de cette expérience que MECANO permet d'exprimer des informations connues des concepteurs, importantes aussi bien pour la réutilisation que pour la maintenance, qui n'avaient pu être capturées par le modèle EIFFEL. La lisibilité de la restructuration obtenue est très satisfaisante et permet à un client de la bibliothèque de choisir la classe adaptée à ses besoins.

6.2.3.2. Critique de la bibliothèque

La bibliothèque, à part quelques petits problèmes, semble homogène. Le seul domaine difficile à modéliser en MECANO est le domaine des arbres. Les spécialisations et les réalisations sont correctes, mais les liens d'héritage exetrnes sur le domaine des listes ne

correspondent pas aux contraintes de la relation d'implantation, ni à celles de la relation de comportement. En fait, il semble qu'un autre type d'héritage puisse être défini, qui consiste à hériter d'une classe concrète pour l'implantation tout en exportant certaines méthodes (après un renommage éventuel afin d'adapter le nom).

7 . CONCLUSION

La réutilisation, propriété essentielle des logiciels et des processus de développement de demain trouve avec MECANO une place privilégiée, bouleversant quelque peu le cycle de développement habituel du logiciel. Ceci n'est pas anormal quand on sait que ce cycle a surtout été conçu pour industrialiser un processus de développement assurant la fiabilité, au détriment de la généralisation.
L'exemple que nous avons développé a montré les apports de la méthode en ce qui concerne l'évolutivité et la réutilisabilité des conceptions obtenues. Les entités complexes que sont les domaines, les composites et les applications, apportent des moyens de structurations complémentaires au concepteur. Les graphes d'utilisation et d'héritage sont organisés en sous-graphes d'entités possédant un critère de cohésion fort (spécialisations/réalisations, tout/parties et graphe de communication).

Dans la présentation de la méthode, nous faisons souvent référence au poste de conception. Nous sommes convaincus, en effet, qu'une méthode, même excellente, ne pourra être appliquée et donc validée que si elle est utilisée. Pour cela, il faut qu'un outil permette de la mettre en oeuvre en déchargeant le concepteur des tâches rébarbatives et en offrant une ergonomie maximale. La forme n'est pas sans incidence sur le fond, et la facilité d'utilisation, la richesse d'une représentation graphique, la présence d'outils d'édition, d'interrogation et de vérification sont des qualités au moins aussi importantes que la richesse du modèle sémantique lui-même. La programmation structurée n'est devenue ce qu'elle est que grâce à l'émergence de langages comme PASCAL ou ADA. L'intérêt que portent les programmeurs C à C++ montre bien que les concepts sont utilisés quand le support technique (langage, environnement de développement ou de conception) les intègre.
Le chapitre suivant est consacré à la présentation du poste de conception destiné à la gestion de la méthode MECANO.

1. INTRODUCTION

1.1. Présentation

Cette partie a pour but de présenter succinctement le poste de conception MECANO qui constitue un atelier de développement de logiciel pour la méthode MECANO [Girod 90] [Girod 91]. Il permet, entre autres, de gérer l'édition et l'archivage d'une conception graphique, d'assurer le respect des contraintes de la méthode, d'interroger un projet et d'obtenir directement un composant.
Dans ce chapitre nous commençons par décrire l'interface graphique ("style" de l'interaction homme-machine et objets graphiques manipulés), puis les différentes fonctionnalités du système sont exposées: navigation, édition, modification, gestion des vues, vérification de contraintes et sauvegarde.

1.2. Fonctionnalités générales

Le poste de conception a pour principale fonction de prendre en charge les conceptions effectuées au moyen de la méthode MECANO. Il nous a semblé important de développer un tel outil pour plusieurs raisons:

Prise en charge de tâches "ingrates":
L'utilisation d'une méthode implique la description d'une conception dans un formalisme particulier. Dans le cas de MECANO, ce formalisme est essentiellement graphique. L'outil doit fournir un support d'édition graphique permettant au concepteur de bâtir des conceptions avec des "objets" graphiques élémentaires, représentatifs des entités manipulées dans la méthode (classe, composite, domaine, application, projet, relation). Il faut disposer d'un éditeur de graphes pour que la redisposition des objets graphiques par l'utilisateur (ou le système) soit aisée.

Gestion des contraintes
La méthode induit un certain nombre de contraintes: règles de constitution des domaines, des composites ou des applications, règles sur les liens d'héritage et d'utilisation, règles sur les relations internes/externes. Ces règles doivent être vérifiées par l'outil. De plus, il est souhaitable que l'éditeur graphique soit dirigé par la méthode, c'est-à-dire que les contraintes structurelles (constitution des entités) soient prises en compte **a priori**.

Gestion de la cohérence et de la complétude
L'outil doit être capable de détecter des incohérences entre les entités, par exemple entre le texte d'une classe et la représentation graphique. Des vérifications de complétude sur demande sont aussi souhaitables afin de vérifier que toutes les références sont résolues.

Recherche de composant
Une autre tâche importante est la prise en compte de la navigation dans une conception. On appelle navigation la possibilité de rechercher (pour visualiser ou éditer) un ou plusieurs composants à partir d'un composant donné. La navigation est très importante, à la fois pour visualiser une conception, et aussi pour la recherche de composants réutilisables dans les bibliothèques. Nous verrons que cette navigation peut être graphique ou par critères.

Visualisation des conceptions
Enfin, nous pensons qu'il est important de fournir des moyens de personnalisation de l'affichage des conceptions afin de présenter des **vues** de celles-ci selon des critères donnés (niveau de détail, vue client créateur, vue client utilisateur, vue classes virtuelles etc...). On améliore ainsi la lisibilité en supprimant le bruit apporté par des éléments trop spécifiques ou inutiles par rapport à un niveau d'abstraction ou une vue donnés[1] (cf chapitre 4, figure 33).

2 . L'INTERFACE GRAPHIQUE

2.1. Description générale

Le principe de base de MECANO est de privilégier au maximum la représentation graphique de la conception. Le type d'interface choisi repose sur les principes simples des stations de travail actuelles. Le concepteur dispose d'un moyen d'interaction par souris pour effectuer une sélection, un "zooming"[2], une activation d'opération, un déplacement d'objet ou un accès à un menu déroulant.
D'autre part, le multi-fenêtrage est largement utilisé pour assurer la visualisation des entités avec les opérations d'iconification, de retaillage, de déplacement et de "scrolling" classiques.

2.1.1. Principes de la représentation des entités de conception

Le choix a été fait de présenter une entité de conception par fenêtre. De cette façon, le concepteur a le moyen de faire apparaître le contenu d'une entité quand il le désire sans pour autant perdre la visualisation des autres entités en cours de conception. Le passage d'une fenêtre à l'autre se fait par simple déplacement du pointeur d'écran (souris).
Une entité représentée dans une fenêtre est dite en **forme complète**. L'autre forme de l'entité est la **forme pictogramme** correspondant à une symbolisation de l'entité dans le contexte d'une entité englobante. La représentation complète est éditable par l'utilisateur et s'affiche dans une fenêtre indépendante, elle est unique pour une entité donnée. La forme pictogramme d'une entité peut apparaître dans plusieurs fenêtres. Néanmoins, elle n'apparaîtra qu'une seule fois sans plot d'appartenance (en tant qu'entité interne).

2.1.2. Les fenêtres MECANO

- **Fenêtre graphique :**

Les fenêtres graphiques sont utilisées pour les entités complexes: projet, domaine, application et composite. Elles sont constituées de représentations symboliques d'entités (pictogramme) reliées par des relations. L'image est directement manipulable par l'utilisateur en accord avec les règles de la méthode, au moyen de l'éditeur graphique et d'une palette d'édition.

---

[1] Dans le chapitre 4, le domaine de la figure 36 est représenté sans les classes externes et seules les relations de spécialisation, d'adaptation et de réalisation sont visibles.

[2] Un zooming est une action qui consiste à grossir un élément afin d'en voir le contenu. En général, ceci correspond à l'ouverture d'une fenêtre présentant une version détaillée d'un composant symbolisé (icône ou pictogramme).

• **Fenêtre texte :**
Les fenêtres textuelles servent à représenter les classes. Elles permettent de voir et de choisir un constituant (rubrique) de classe dans une liste puis d'éditer l'information relative à ce constituant.

2.1.3. Relation d'appartenance et zooming

Les différentes entités sont structurées par la relation d'appartenance définie au chapitre 4. Nous rappelons que cette relation permet de décomposer une conception MECANO (le projet) en Domaines et Applications. Chaque Domaine ou Application est ensuite décomposé en Composites et Classes. Enfin, les Composites sont constitués de Classes et de Composites.
La relation d'appartenance définit la structure arborescente stricte de la base de conception, ce qui permet de privilégier l'entité englobante par rapport aux autres entités complexes dans lesquelles une entité peut apparaître. Cette relation est exploitée par le poste MECANO dans les situations suivantes:

- Affichage de la forme pictogramme d'une entité: selon que l'entité appartient ou non à l'entité complexe représentée, le pictogramme correspondant est affiché sans plot d'appartenance ou bien avec.

- Création et destruction d'entité: pour créer ou détruire une entité, le concepteur doit obligatoirement le faire depuis l'entité englobante à laquelle elle appartient.

- Recherche et désignation d'entité: le nom complet d'une entité comprend le chemin sur la relation d'appartenance depuis la racine de la conception (le projet).

Pour obtenir la forme complète d'une entité, il suffit d'effectuer un zooming sur une forme pictogramme de l'entité (clic sur le pictogramme), une nouvelle fenêtre est créée représentant le contenu de l'entité. MECANO offre ainsi un premier moyen naturel de navigation ou "browsing" sur la relation d'appartenance.

2.1.4. Les menus

Le concepteur peut invoquer des actions sur l'entité courante en sélectionnant une opération dans les menus déroulants. Les menus sont regroupés dans une barre de menu située en haut de chaque fenêtre d'entité (invocation près de la source). Cette structure proche de celle du Macintosh nous paraît plus ergonomique que les menus fugitifs type "X-Window" se déroulant à l'endroit du curseur en fonction de la zone de positionnement. Nous suivons, en cela, la règle de la conservation de la localisation énoncée dans [Coutaz 89][1].
Il sera cependant possible d'obtenir un menu localement à un pictogramme, constitué exclusivement d'opérations sur ce pictogramme (cf Annexe C.).

2.2. Représentation graphique des entités

2.2.1. Forme complète d'une entité

Les entités MECANO sont représentées en forme complète par une fenêtre. De façon générale, cette fenêtre possède les caractéristiques de la figure 1.

---

[1] Cette règle est l'une des sept règles d'or de l'implantation d'une interface homme-machine proposées par J. Coutaz.

Composition d'une fenêtre MECANO
La fenêtre est découpée en zones qui sont immuables quel que soit le type de l'entité représentée. Les différentes zones sont la barre de titre, la barre de menu, la zone d'affichage de l'entité et la zone palette ou liste.

• Barre de titre (située en haut de la fenêtre)
La barre de titre contient le type et le nom de l'entité (par exemple "domaine tortue") ainsi que des icônes spécifiques. En plus des icônes traditionnelles de fermeture et de retaillage d'une fenêtre, la barre de titre contient des icônes d'information et des icônes d'action (Cf Annexe B).

• Icônes d'information (à gauche du titre)
Elles donnent une information visuelle rapide sur l'état de l'entité: type de l'entité, mode de travail, etc... et ne sont pas "activables".

• Icônes d'action (à droite du titre)
Elles offrent un raccourci souris pour déclencher des opérations d'ordre général: passage du mode édition au mode lecture, visualisation/masquage de la zone liste, etc.... L'opération est déclenchée par un clic souris sur l'icône correspondante.

• Barre de menus (située en haut de la fenêtre, au dessous de la barre de titre)
Elle contient les menus déroulables. Ces menus offrent toutes les opérations disponibles dans le contexte de l'entité visualisable(cf Annexe C).

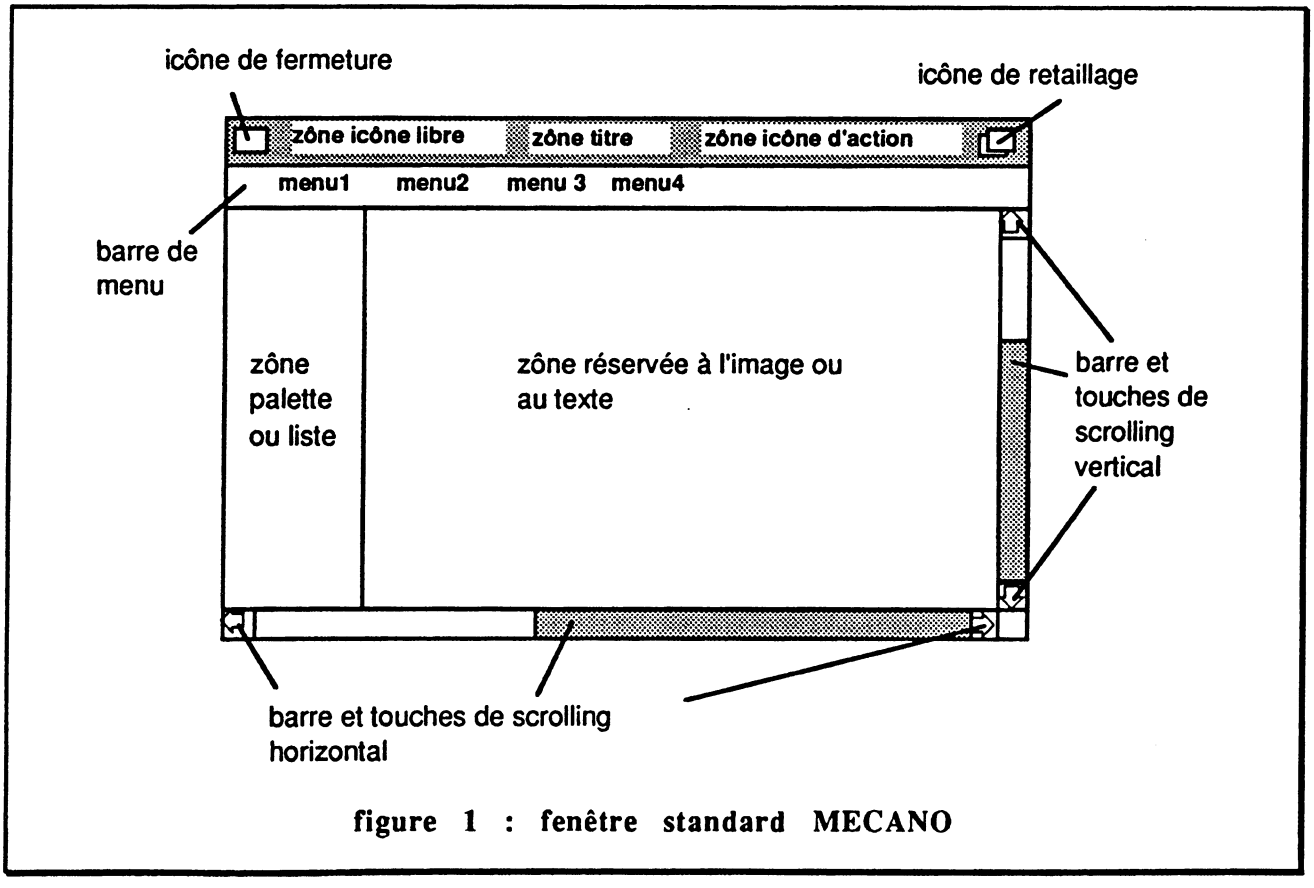


figure 1 : fenêtre standard MECANO

• Zone d'affichage de l'entité :
La zone d'affichage se subdivise en deux sous-zones. L'une, située à gauche, est la zone palette. L'autre, à droite, est la zone contenu.

• Zone palette ou liste: cette zone contient une information non éditable. Si la fenêtre est graphique, il s'agit de la palette d'édition: liste des objets graphiques utilisables dans le contexte courant. Si la fenêtre est textuelle, cette zone contient une liste de sélection, par exemple la liste des méthodes exportées d'une classe.

• Zone de contenu: la partie droite de la zone d'affichage est réservée à l'affichage d'une information sur l'entité. Cette zone est généralement éditable. S'il s'agit d'une entité graphique, les sous-entités (pictogrammes) sont visualisées avec les relations inter-entités. S'il s'agit d'une entité textuelle, la zone présente l'information correspondant à l'élément sélectionné dans la liste de droite, par exemple la définition d'un attribut. Dans les deux cas, des ascenseurs permettent le scrolling horizontal et vertical de cette zone.

Manipulation de la fenêtre
La fenêtre peut être manipulée de façon classique par le concepteur: retaillage, fermeture, déplacement et scrolling.

Entité graphique
Les entités graphiques de MECANO possèdent une zone pour la palette d'édition, à gauche de la zone d'affichage (figure 2). Cette palette contient les objets graphiques (pictogrammes et relations) utilisables pour la création de l'entité. Si la palette est plus grande que la zone, les flèches en bas de la palette permettent de faire défiler celle-ci. On peut faire disparaître la palette pour disposer d'un affichage plus grand en cliquant sur l'icône "visualisation de la palette" dans la barre de titre (pinceau).

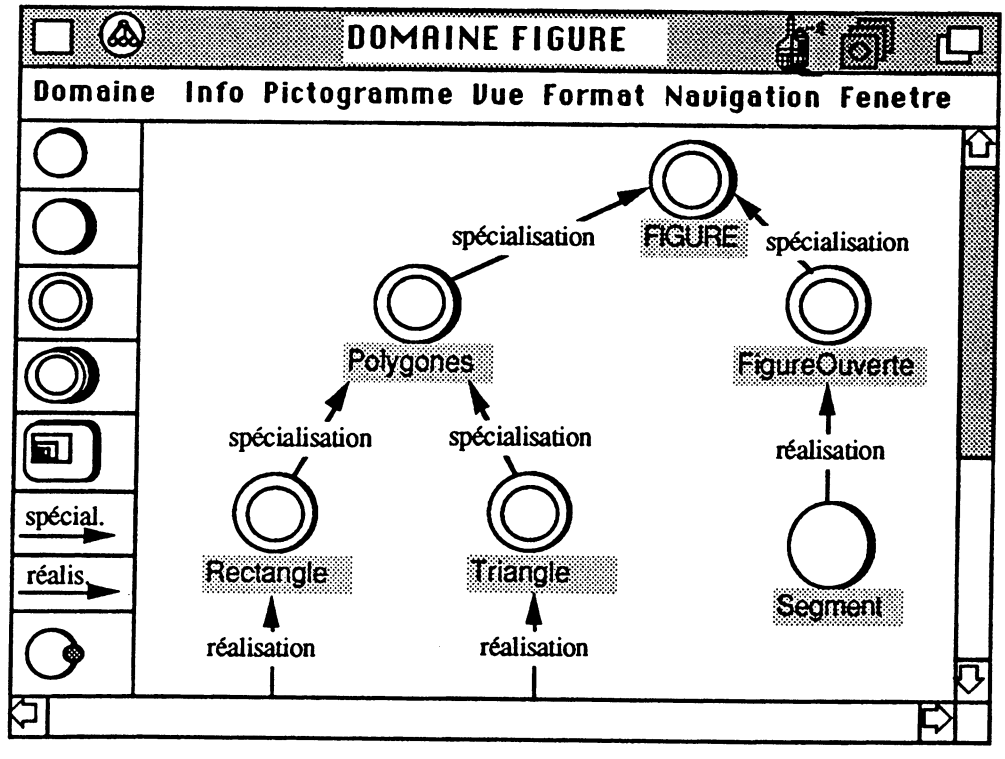


figure 2 : Fenêtre graphique

La partie droite de la zone d'affichage est réservée au graphique représentant l'entité. Les ascenseurs permettent de "dérouler" le contenu.
L'icône d'activation de l'éditeur permet de passer en mode édition. Par défaut, c'est le mode lecture qui est sélectionné.

L'icône de gauche indique le type de l'entité représentée (ici un domaine).

Entité textuelle
Les entités dont le contenu n'est pas graphique sont représentées dans une fenêtre textuelle munie d'une zone liste (figure 3). La zone de gauche contient une liste de sélections qui permet au concepteur de choisir un élément particulier d'une rubrique de l'entité. L'élément choisi est affiché dans la zone de droite sous forme textuelle pour être édité. Si le mode "lecture seule" est choisi, la sélection d'un élément provoque la création d'une fenêtre textuelle auxiliaire. L'élément est alors en mode consultation. On assure la cohérence en n'autorisant qu'une seule édition à la fois. Les fenêtres auxiliaires de consultation permettent de visualiser plusieurs informations relatives à la même entité.
Une icône d'action fait apparaître/disparaître la zone liste.

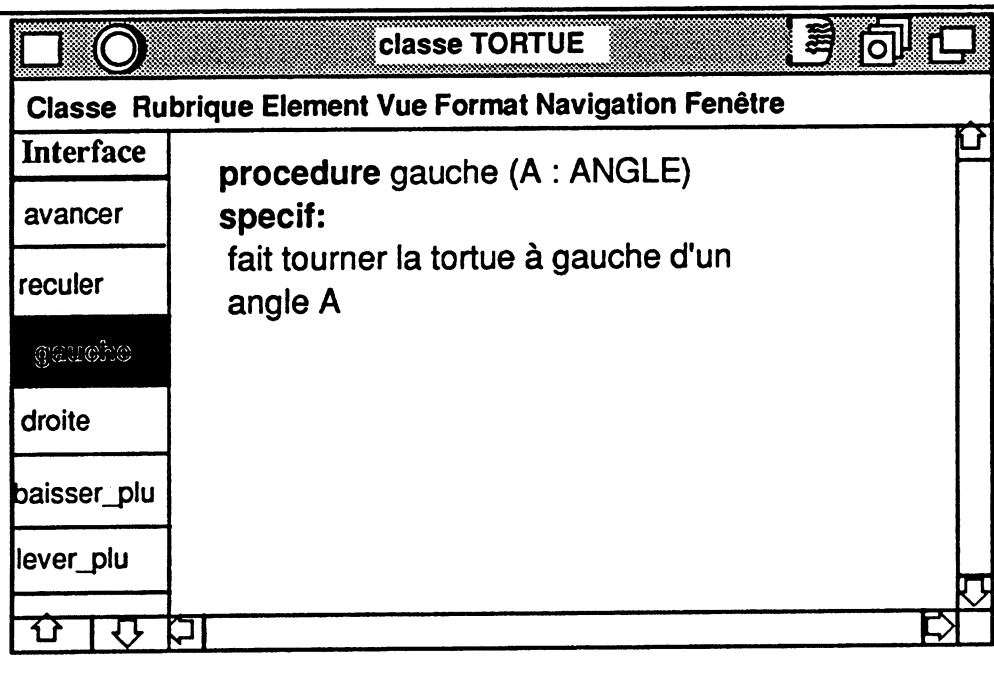


figure 3 : fenêtre textuelle

2.2.2. Les pictogrammes

Un pictogramme permet de représenter symboliquement une entité de conception dans le contexte d'une entité englobante.
Il est constitué d'une icône (symbolisation du type de l'entité), d'un label (nom de l'entité) et éventuellement d'un "plot" (cf chapitre 4). Le plot est une mini-icône d'action permettant d'atteindre rapidement, par simple clic, l'entité structurante à laquelle appartient l'entité symbolisée par le pictogramme.
Un menu associé au pictogramme peut être obtenu en cliquant avec le bouton central. Il offre des informations rapides sur l'entité, ainsi que des opérations de destruction, de sélection pour transfert ou d'édition du nom (cf Annexe C).

• **Les relations**

Les relations permettent de relier les entités MECANO en accord avec la méthode. Elles possèdent un label, généralement non éditable, indiquant le type de relation; seul le label de la relation de composition est éditable (nom de l'attribut). Les relations disponibles pour l'entité courante sont présentes dans la palette d'édition.
Une relation peut être sélectionnée en vue de sa destruction. Par contre, on ne peut modifier la destination ou la source d'une relation. Le déplacement d'un pictogramme mis en jeu dans la relation provoque le déplacement de la flèche correspondante.
Les relations entre entités externes peuvent être visibles mais ne sont pas éditables.

• **Opérations directes sur les pictogrammes**

Sur une fenêtre graphique, l'utilisateur peut appliquer aux pictogrammes les opérations suivantes en agissant directement sur le dessin au moyen de la souris.

Ouverture (ou Zooming)

Un clic sur le pictogramme avec le bouton droit (hors zone plot) permet d'obtenir la forme complète de l'entité. La fenêtre correspondante est ouverte et s'affiche en premier plan.

Sélection

Un clic avec le bouton gauche sur une entité de la fenêtre courante permet de sélectionner (inverse vidéo) un pictogramme. Une entité sélectionnée devient argument pour les fonctions que l'utilisateur désire appliquer : détruire, déplacer, transférer, recopier etc... La sélection peut être multiple en glissant la souris pendant la sélection [Robert 91].

Création

La palette d'édition graphique indique les pictogrammes et les relations utilisables pour la conception de l'entité courante. En cliquant sur un pictogramme de la palette puis sur un point du dessin, on crée une entité à l'endroit indiqué. Une boîte de dialogue permet de saisir les informations complémentaires relatives à cette entité.

Etablir une relation :

Pour positionner une relation entre deux pictogrammes, le concepteur sélectionne cette relation dans la palette, puis le pictogramme source et enfin le pictogramme destination (dans cet ordre). Une boîte de dialogue apparaît pour saisir le label dans le cas des relations de composition. Si la relation est interdite, l'utilisateur en est informé et sa commande est annulée.

Destruction

On sélectionne l'entité à détruire et on lance la commande *supprimer* dans le menu *pictogramme*. Pour détruire une relation, il suffit de la sélectionner et de lancer l'opération détruire du menu entité. Selon les cas, MECANO interdit l'action ou demande la confirmation après information sur les effets de bord générés par la destruction.

Déplacer

Pour déplacer un pictogramme, il suffit de le sélectionner puis de faire glisser la souris en maintenant le bouton enfoncé, et de relâcher le bouton à l'endroit désiré. Cette opération s'applique aussi à une sélection multiple. Les relations ne sont pas déplaçables, elles suivent les déplacements des entités sources et destinations.

Accès à l'entité d'appartenance

Un clic avec le bouton droit dans le plot d'appartenance permet d'ouvrir ou de réactiver la fenêtre de la représentation complète de l'entité englobante. La fenêtre correspondante s'affiche (ou se réaffiche) en premier plan.

Menu
Après sélection d'un pictogramme ou d'une relation et clic sur le bouton central, on obtient alors le menu déroulant des opérations sur l'entité (identique au menu *pictogramme* de la barre de menu, cf Annexe C).

2.2.3. Le projet

Le projet est l'entité racine de l'arbre d'appartenance d'une conception MECANO. Sa forme complète contient des pictogrammes d'applications et de domaines. La palette d'édition n'offre que ces deux types d'entité. Aucune relation n'est permise entre les entités d'un projet (figure 4).

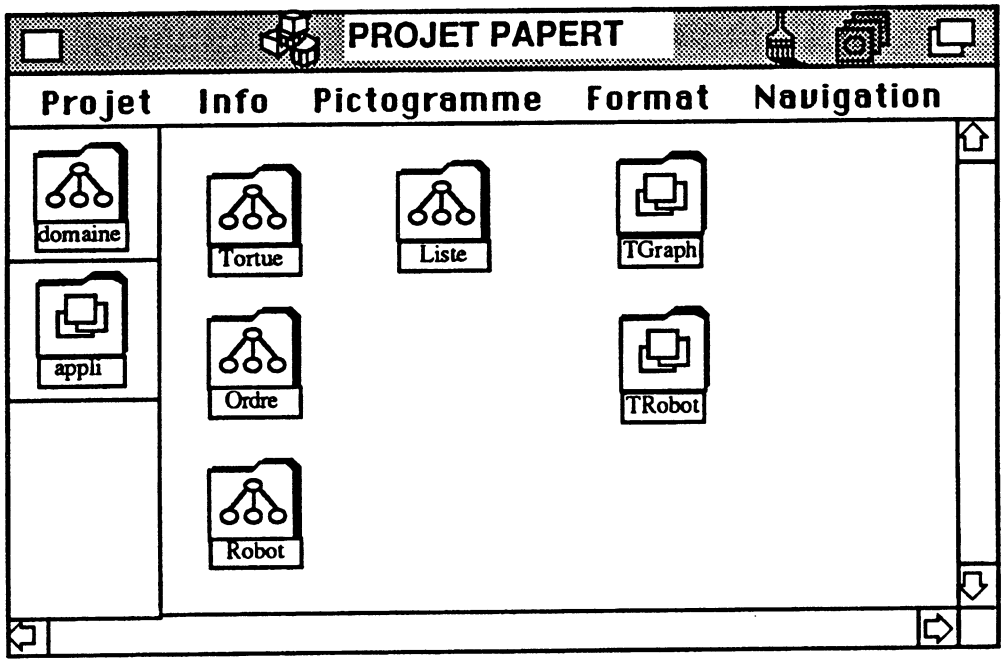


figure 4 : fenêtre projet

3. FONCTIONNALITES DU POSTE DE CONCEPTION

Nous présentons dans cette partie les fonctionnalités du poste de conception. Les outils proposés assurent les fonctions principales de navigation et d'édition de la conception. La prise en compte de vues de conception et la gestion des contraintes de la méthode sont décrites. Les opérations offertes par les menus associés à chaque type d'entité sont détaillées en Annexe C.

3.1. Les grandes fonctions

Le poste de conception a pour principale fonction la construction et la visualisation d'une conception selon le formalisme et les règles de la méthode MECANO. Il va décharger le concepteur des tâches d'archivage, de construction graphique et de recherche d'informations. Le premier outil que nous décrivons est le navigateur

CHAPITRE 5:

LE POSTE DE CONCEPTION MECANO

People are most adept at visual thinking,
text should be replaced by graphic images as far as possible[1] *[Cox 83]*

ENVIRONNEMENT GRAPHIQUE

---

1 Les gens sont avant tout experts en pensée visuelle, les textes doivent être remplacés par des images graphiques autant que possible

(browser en anglais) qui permet de se déplacer dans l'ensemble de la conception et d'obtenir la visualisation de l'entité désirée.

3.2. Navigation

La navigation ou "browsing" est à la base du poste de conception. Elle permet d'atteindre une nouvelle entité de conception à partir d'une entité courante afin de l'éditer ou simplement d'en obtenir une information. Dans le cadre de la conception objet, cette fonctionnalité est très importante. En effet, la brique de base -la classe-, bien que relativement autonome, interagit dans un système complexe faisant intervenir des relations d'héritage et des relations client. Dans le premier cas, les classes ancêtres fournissent un patrimoine disponible pour la classe et donc pour son implantation. Dans le second cas, une classe cliente s'appuie sur les opérations offertes par une classe fournisseur pour réaliser ses propres services. Dans les deux cas, il faut pouvoir accéder facilement à la classe cible de la relation.

Navigation graphique
Le browser graphique (ou navigateur) de MECANO s'appuie sur plusieurs principes:

• appartenance: constituant d'une entité
La relation d'appartenance est privilégiée puisqu'elle permet de passer d'un niveau de détail au suivant. Une entité en forme complète se représente par un ensemble de pictogrammes symbolisant les entités qu'elle contient. On peut obtenir la visualisation de l'une de ces entités par simple zooming.

• appartenance: accès à l'entité englobante
Réciproquement, on peut obtenir, à partir d'une entité sous forme pictogramme, l'entité englobante à laquelle elle appartient. Si ce pictogramme n'a pas de plot d'appartenance, l'entité englobante est celle représentée par la fenêtre. Si le pictogramme possède un plot d'appartenance, un simple clic sur ce plot fera apparaître l'entité englobante correspondante.
Cette deuxième façon de naviguer permet de parcourir les conceptions transversalement et fait jouer la loi de la proximité: on peut atteindre une entité structurante voisine à travers les entités référencées dans l'entité courante.

• relation de délégation et d'héritage
Les relations de délégation et d'héritage sont visualisées sur les formes complètes. Le concepteur peut alors parcourir les différents graphes et atteindre de cette façon des entités appartenant au contexte d'une entité donnée (éventuellement après scrolling de la fenêtre ou accès aux entités complexes voisines).

Recherche de composants par critère
Ces premiers moyens de navigation purement graphiques sont complétés d'une recherche par critères permettant d'obtenir l'entité recherchée (menu *navigation*). Les critères s'appliquent à l'entité courante dont la forme complète est visualisée dans la fenêtre. Les critères retenus sont décrits en Annexe C, en voici quelques exemples:

- tous les critères graphiques précédents
- recherche par nom
- recherche par rubrique: méthode, attribut, interface ...
- recherche sur les attributs du menu INFO
- classes clientes de la classe courante
- classes héritières de la classe courante
- entités référençant une entité donnée

remarque : si plusieurs entités répondent aux critères, une liste de sélection contenant l'ensemble des entités possibles est présentée à l'utilisateur pour lui permettre de choisir celle qu'il désire.

3.3. Edition

Le poste MECANO est un atelier de conception d'applications et propose à ce titre des activités de création, de modification et de destruction des entités en cours de conception. Un éditeur graphique est proposé pour créer de nouvelles entités et des relations dans l'entité courante. Il y a deux modes d'édition selon que l'on crée une nouvelle entité (entité interne) ou que l'on ajoute une entité existante au sein d'une entité complexe (entité externe).

Edition locale (entité interne)
Afin de concevoir un projet, l'utilisateur doit créer de nouvelles entités. Pour cela, il dispose d'un éditeur spécialisé à chaque niveau d'entités graphiques. L'éditeur est composé d'une palette de pictogrammes et de liens correspondant aux entités et relations qu'il est possible de créer dans le contexte de l'entité en cours de conception.
Un graphe d'entités est élaboré en ajoutant des pictogrammes et des liens.
Pour créer un pictogramme, il suffit de pointer l'icône correspondante dans la palette et de désigner l'endroit où le nouveau pictogramme doit être inséré. L'éditeur prend en charge les demandes d'informations complémentaires : nom de l'entité ou label de la relation.
Une entité ainsi créée sera considérée comme appartenant (interne) à l'entité englobante courante.

Edition externe (entité externe) ou transfert d'entité
Le concepteur peut avoir besoin d'une entité déjà créée mais appartenant à une autre entité complexe, par exemple la classe TRIANGLE du domaine FIGURE pour implanter la TORTUE_ECRAN (Cf Chapitre 4). Cette classe est une entité externe mais il est possible de la faire apparaître dans l'entité courante par le mécanisme de transfert.
Le transfert se déroule en plusieurs étapes:

Etape 1)
Sélectionner l'entité dans son contexte d'appartenance soit par navigation graphique, soit par recherche par critère;
Etape 2)
Appliquer l'opération de transfert ***exporter*** *du menu local au pictogramme ou du menu pictogramme de l'entité.*
Etape 3)
Revenir dans le contexte de l'entité en cours de conception et lancer l'opération d'acquisition externe ***importer*** *après avoir indiqué l'endroit ou doit s'insérer l'entité.*

Remarque : le transfert possède deux modes opératoires:
• Transfert simple: on transfère une référence à une entité externe, l'entité reste interne à son ancien contexte et sera externe dans son nouveau contexte.
• Transfert d'appartenance: c'est l'entité qui est transférée, elle devient interne à l'entité courante et externe à son ancienne entité englobante. Le transfert complet se déroule en deux temps : transfert simple + changement d'appartenance.
• Afin de gérer les incomplétudes passagères (classes référencées dans l'étape B du processus de conception), MECANO offre la possibilité de référencer des entités externes sans qu'elles aient été définies préalablement. Dans ce cas, le plot d'appartenance est grisé, indiquant ainsi que la référence n'est pas résolue (voir la palette du domaine de la figure 2).

La destruction
Pour détruire une entité, le concepteur doit sélectionner l'entité et appliquer l'opération *supprimer* du menu déroulant de l'entité.
Selon que l'entité est interne ou externe, la destruction n'a pas le même effet:
1) entité externe:
C'est la référence qui est détruite, l'entité existe toujours dans son contexte d'appartenance et pour les autres références.
2) entité interne:
L'entité est détruite complètement après information des effets de bord et demande de confirmation à l'utilisateur. Dans ce cas, toutes les informations relatives à l'entité sont perdues, les références dans les autres entités sont détruites.

Dans les deux cas, les relations qui atteignaient le pictogramme sont détruites. La liste des effets de bord est présentée à l'utilisateur avant destruction, elle comprend toutes les entités qui référencent l'entité à détruire (héritage et délégation).

L'édition des classes
Les rubriques et les éléments d'une classe (cf Chapitre 4) sont édités dans une fenêtre textuelle. L'élément à éditer est représenté dans la fenêtre après sélection de celui-ci dans la liste de gauche (exemple: méthode "avancer" de l'interface de la classe TORTUE). Le concepteur peut alors modifier le texte correspondant à chaque item de l'élément. Pour que la modification soit prise en compte, il faut lancer la commande *enregistrer* ou repasser en mode non-éditable.

3.4. Gestion des vues de conception

L'affichage des images graphiques se fait dans un format prédéfini selon le type d'entité (format standard). Pour les domaines par exemple, l'affichage ne concerne que l'arbre des classes ou composites reliés par l'héritage de spécialisation et de réalisation. L'utilisateur peut cependant "voir" la conception sous différents aspects. Pour cela, il dispose du menu VUE permettant de masquer ou au contraire d'afficher certaines entités ou relations. Il y a deux sortes de vues:

• **Vue calculée** : il s'agit d'un masquage ou démasquage de pictogrammes satisfaisant une contrainte exprimée sur le "type" des entités et des relations.
- Pour les entités, les types suivants sont disponibles:
classe concrète, classe générique, classe virtuelle, domaine, composite, application, entité externe, entité interne.
- Pour les relations, les types suivants sont disponibles:
spécialisation, réalisation, comportement, implantation, fusion, adaptation, composition, communication, utilisation secondaire, relation externe/externe.
Par exemple, le concepteur peut demander à voir un domaine privé des relations d'implantation et des classes externes ou masquer les classes concrètes (vision client non créateur de classe).

Remarque: le masquage d'une classe entraîne le masquage des relations adjacentes. Le masquage d'une relation efface la flèche correspondante.

• **Vue définie** : une vue définie est construite directement par le concepteur par multi-sélection sur la fenêtre courante. Après avoir sélectionné les entités à masquer, l'opération **masquer sélection** est lancée. Le système efface de la représentation graphique toutes les entités sélectionnées et les relations adjacentes.

A tout moment, l'utilisateur peut revenir à l'une des deux vues suivantes :

- Standard : vue standard pour le type d'entité de conception. Par exemple, la vue standard d'un domaine ne présente que les classes internes et les liens de spécialisation, réalisation et adaptation.
- Complète : toutes les relations et les entités internes et externes sont visibles

Remarque: toutes les manipulations de vues ne génèrent aucun effet de bord sur la conception. Seul l'affichage et la possibilité de sélection sont modifiés.

3.5. Gestion des contraintes de la méthode

Le poste de conception permet de gérer les contraintes de la méthode MECANO.

Syntaxe:
Les règles de syntaxe sont prises en compte pour une bonne partie a priori c'est-à-dire que l'utilisateur ne peut concevoir que par les formalismes proposés. Le formalisme graphique pour la représentation des entités complexes et le formalisme textuel (rubriques et éléments de classe) sont gérés par l'éditeur.

Sémantique
Les règles plus précises concernant la composition des entités (domaine, composite ...) sont prises en compte par l'éditeur. La palette d'édition offre à priori les pictogrammes disponibles pour l'entité en cours de conception, ainsi que les relations préétiquetées autorisées. Lors de l'ajout d'une relation, MECANO vérifie qu'elle est autorisée en fonction des entités sources et destinations.
Les contraintes sur les liens d'héritages énoncées au chapitre 4 sont dans la mesure du possible prises en compte à priori.
De même, l'outil prend en compte le contrôle sur les éléments composites (visibilité) et sur le raccordement des entités externes.
Certaines règles ne sont contrôlées qu'après un événement précis (fin d'édition, fermeture de fenêtre). Enfin, des vérifications peuvent être exécutées sur demande (complétude, vérification de certaines cohérences coûteuse).

Il nous semble important de ventiler la gestion des règles parmi ces différents modes (a priori, a posteriori et sur demande) afin de préserver un équilibre "liberté de création/ assistance automatique" nécessaire à l'activité de conception. On doit noter ici que certaines actions peuvent induire des effets de bord sur d'autres entités qui seront pris en compte par MECANO en fin d'édition de l'entité courante après message d'avertissement.

3.6. Sauvegarde des entités et gestion du contexte

Le poste de conception assure des fonctions de sauvegarde de conception et de contexte.

Sauvegarde
MECANO conserve à tout moment les entités de conception et leurs inter-relations en mémoire. Après chaque édition, les entités créées, modifiées ou détruites sont sauvegardées sur fichier avec leur contexte. Au démarrage d'une session, le concepteur retrouve le projet et les entités de conception déjà créées.

Contexte de conception
L'utilisateur compose les entités de manière graphique. Le positionnement des entités est sauvegardé par MECANO afin de pouvoir présenter le même graphe avec le même agencement des pictogrammes lors d'une prochaine ouverture. Pour cela, un contexte de conception est géré par MECANO et contient pour chaque entité:
- le positionnement des pictogrammes dans l'image

- la taille de la fenêtre et sa position dans l'écran
- la zone de l'image affichée dans la fenêtre
- la vue demandée par l'utilisateur
- les modes de travail sélectionnés : édition, vue concepteur, ...

Un contexte est associé à une entité. A la fermeture de l'entité, le contexte est sauvegardé. Il sera restauré à la prochaine ouverture, y compris si elle intervient dans une autre session.
On notera que la notion de contexte est purement externe à la conception. En particulier, on peut toujours obtenir une entité dans son contexte par défaut : taille de fenêtre standard, masquage lié au type d'entité, mode non éditable, mode utilisateur simple, positionnement automatique des entités. L'enregistrement d'un contexte (fermeture) détruit le contexte précédent. L'utilisateur ne peut pas sauvegarder de contexte intermédiaire qui nécessiterait une gestion des versions de conception.

4. CONCLUSION

4.1. Conception en MECANO

Nous concluons ce chapitre par un aperçu de l'implantation du système MECANO.
Le poste de conception est en cours de réalisation. L'interface graphique est en phase d'implantation. La partie application est en phase de conception.
Nous avons porté un intérêt tout particulier à la conception du poste MECANO. Cette conception est menée au moyen de la méthode MECANO elle-même, ce qui nous a permis de valider et d'améliorer nos idées de départ. Parallèlement, elle nous a conduit à affiner nos besoins en termes de fonctionnalités du poste de conception. Malheureusement, ne disposant pas du poste pour le concevoir, nous avons dû nous résigner à un travail sur papier (et traitement graphique) fastidieux. Nous pensons porter notre conception sur le poste MECANO dès que celui-ci sera prêt afin d'être à même de le faire évoluer dans les meilleures conditions.

Au niveau de la méthode, le typage de l'héritage a été très utilisé sauf peut être la fusion qui reste une situation peu courante. Notre argumentation sur l'héritage multiple s'avère juste: la plupart des classes ont plus d'un parent, spécialement celles de l'interface graphique. La spécialisation et la structure associée, le domaine, ont joué un rôle central. Toute l'interface du poste est structurée en domaines. Certains ont jusqu'à 4 niveaux d'abstraction. Des exemples tirés de la conception du poste sont présentés en Annexe D.

4.2. Adéquation du modèle objet

La réalisation de l'interface graphique a montré, une fois de plus, que ce domaine d'application est le terrain de prédilection du modèle objet. Comme le dit J. Coutaz: "le modèle objet permet de décentraliser le contrôle de l'interaction, un objet connait son état, l'utilisateur peut passer librement d'objet en objet ce qui permet de laisser le contrôle sous la responsabilité de l'utilisateur". Nous ajouterons que l'héritage, en permettant de structurer des concepts relativement complexes tels que les fenêtres, les menus, les événements, les boites, les figures et les icônes, conduit à des architectures souples et évolutives. Le polymorphisme est utilisé systématiquement quand cela est possible.

4.3. Considérations techniques

Le poste est réalisé au moyen de la version 2.2 du langage EIFFEL. Nous avons fait un grand usage de la bibliothèque. Mais nous n'avons pas pu utiliser la bibliothèque graphique proposée dans la version 2.2 de EIFFEL. En effet cette bibliothèque fournit un ensemble de Widget propre à EIFFEL et trop restreint pour nos besoins. Nous avons développé entièrement une nouvelle interface complète avec X-WINDOWS 11.4 sous forme de couches superposées. Cette interface permet de manipuler aussi bien les fonctions de la Xlib que les Widgets ATHENA et les Widgets HP .

Pour plus d'informations sur le poste de conception, nous renvoyons le lecteur à [Girod 90a] et [Robert 91].

CONCLUSION

L'approche objet permet d'unifier les phases d'analyse, de conception et d'implantation dans un modèle unique. La plupart des travaux dans ce domaine montrent que cette approche apporte une réponse adaptée aux problèmes de la qualité des logiciels. La flexibilité et la réutilisabilité des programmes sont particulièrement concernées, d'autant plus qu'il est reconnu que la crise du logiciel est largement due à l'absence de ces deux facteurs de qualité dans les logiciels.
L'approche objet, avec les concepts de classe, d'héritage, de polymorphisme et de généricité rend les programmes plus généraux et améliore la stabilité des architectures élaborées non plus sur les fonctions mais sur les objets que manipule le système.

Notre étude a porté sur la phase de conception, c'est à dire sur la mise en place d'une architecture de composants décrivant une solution informatique d'un problème du monde réel. Nous avons appliqué le paradigme objet à cette étape en partant du modèle déjà très riche du langage EIFFEL. Les objectifs de la conception sont, malgré tout, différents de ceux de l'implantation et notre réflexion a porté sur l'utilisation des concepts du modèle objet dans le processus de création d'un logiciel.
Nous avons élaboré une taxonomie de l'héritage en examinant les différentes intentions qui conduisent le concepteur à la création des relations classe/super-classe. Cette taxonomie, comprenant six types d'héritage, peut être décrite au moyen de contraintes sur les classes mises en jeu. Elle permet d'améliorer la lisibilité de la conception en capturant les décisions de conception. De façon réciproque, cette taxonomie favorise l'utilisation de l'héritage, y compris de l'héritage multiple, tout en restant dans un cadre conceptuel bien délimité par les règles de la méthode.
La classification de l'héritage, au travers de la spécialisation et de la réalisation, favorise la mise en place de structures polymorphes assurant l'évolutivité et la réutilisabilité, et permet la création de différentes versions de composants et de systèmes. Ces structures, nommées Domaines, forment l'ossature d'une conception MECANO, elles encapsulent les hiérarchies de classes décrivant un concept de l'application.

Les classes sont aussi reliées par des relations de délégation, supportant le flux des envois de messages à l'exécution. Nous avons étudié cette deuxième relation inter-classes de la même façon que l'héritage, en nous penchant sur les différentes sémantiques associées. Notre propre expérience, confirmée par l'étude des méthodes d'analyse et de conception par objets, a permis de distinguer deux types de délégation: la communication et la composition. Pour prendre en compte ces deux relations et afin d'organiser le graphe de délégation d'une application, MECANO propose deux nouvelles structures: le Composite, bâti sur la composition, et l'Application, bâtie sur la communication. La première offre un concept d'abstraction verticale en encapsulant les structures Tout/parties. Les applications décrivent les différentes versions d'un système comme des réseaux d'objets communiquants.
Le processus de conception que nous avons exposé tire parti de ces différents concepts et indique la façon de mettre en place une conception avec MECANO.

Une partie des résultats de ce travail a fait l'objet d'une publication au congrès TOOLS'90. De nombreuses méthodes d'analyse et de conception par objets ont vu le jour depuis. La plupart sont très récentes, certaines n'ont pas encore été publiées. Le chapitre 2 a montré les avantages de ces méthodes par rapport aux méthodes issues de l'OOD, plus anciennes. C'est d'ailleurs ce constat qui nous avait conduit à entreprendre le développement de notre propre méthode. Nous cependant avons cependant voulu, par honnêteté scientifique, comparer notre étude aux nouvelles méthodes. Certains concepts sont similaires, tels que la communication et les composites, malgré la disparité des termes utilisés. En revanche, la méthode MECANO reste originale sur d'autres points comme la taxonomie de l'héritage, les domaines et le mécanisme d'entités internes/externes.

Un poste de conception a été développé afin d'automatiser la méthode. Il offre un éditeur graphique dirigé par la méthode, des outils de navigation, de recherche de composants, de vérification de contraintes et de présentation de vues.
Un tel outil a été conçu pour permettre d'utiliser intensivement la méthode. Cette utilisation ne sera pas sans retour sur les concepts eux-mêmes et sur les extensions souhaitables. Notre travail futur consiste donc à appliquer et faire appliquer notre méthode, à rassembler les conceptions obtenues et les critiques des concepteurs. Ces résultats devront permettre d'évaluer et de raffiner MECANO. En effet, une méthode favorisant la réutilisabilité et l'évolutivité des logiciels ne peut être validée que sur des projets suffisamment gros et suffisamment longs.

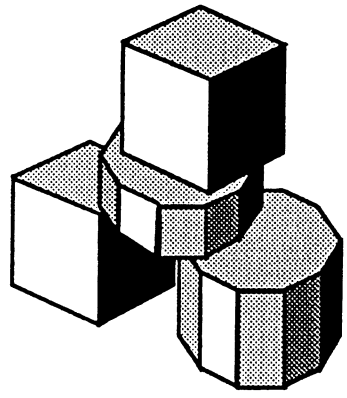
Au cours de ce rapport, un certain nombre d'idées ont été émises et constituent autant d'axes de recherche possibles pour améliorer ou étendre la méthode. Par exemple, une intégration de l'analyse objet dans la méthode est prévue, d'autant plus que le modèle MECANO semble suffisamment riche pour s'adapter à cette phase du cycle de vie. Le travail portera essentiellement sur le processus de conception qui devra expliciter la démarche d'analyse.
Nous prévoyons, par ailleurs, de continuer le travail sur la restructuration des logiciels, avec l'objectif d'intégrer dans le poste de conception des outils d'aide à la restructuration.
Des extensions du poste de conception sont prévues dans d'autres directions:

- passage à la programmation et génération automatique de code,
- gestion de projet (équipes, bibliothèques, configurations ,versions),
- extensions des vérifications de contraintes,
- intégration de processus de conception et de stratégie de développement.

Les monde des objets est passionnant et le nombre de plus en plus grand de personnes qui décident de s'y investir montre bien l'évolution que semble prendre notre façon d'appréhender le développement des logiciels.
Le temps ne paraît plus très loin où le rapport entre la part des programmes originaux et la part des programmes réécrits s'inversera, libérant ainsi le programmeur vers des champs d'investigation nouveaux.

C'est du moins mon intime conviction.

ANNEXES ET BIBLIOGRAPHIE



ANNEXE A

PASSAGE DES TYPES ABSTRAITS AUX OBJETS: EXEMPLE

Nous présentons un exemple de passage d'une spécification de type abstrait vers la ou les classes EIFFEL correspondantes. Le type considéré est le type abstrait GRAPHE et les deux types importés ARC et NOEUD.
ARC et NOEUD donnent chacun naissance à une classe.
GRAPHE conduit à la création de deux classes liées par une relation d'héritage[1]. Une classe virtuelle *GRAPHE* décrit le type abstrait indépendamment d'une réalisation. La classe concrète *GRAPHE_CHAINE* hérite de *GRAPHE* et fournit une réalisation particulière de la classe GRAPHE au moyen de deux listes chainées. Ceci est rendu possible grâce aux méthodes retardées d'EIFFEL. D'autres réalisations peuvent être envisagées, par exemple avec la technique de la matrice d'adjacence (matrice carrée n x n de booléens pour un graphe à n sommets) ou bien par les listes d'adjacences (liste des successeurs associée à chaque noeud).

Remarque
Les axiomes et préconditions dont le numéro est en gras ont été traduits en assertions EIFFEL. Les autres axiomes n'ont pas pu être traduits:
certains concernent des transformateurs composés, par exemple
(31) retirer_arc(a1,ajouter_arc(a2,g)) == si a1=a2 alors g
d'aurtres utilisent la logique du premier ordre, par exemple
(6) possede_arc(a,vide) == faux[2].

I) Type abstraits NOEUD, ARC et GRAPHE

```
type NOEUD
sorte
    noeud
importe
    Entier
operateur
    constructeur
    nd : entier -> noeud
    acces
    numero : noeud -> entier
axiome
    (1) numero(nd(i)) == i
fin type NOEUD
```

```
type ARC
sorte
    arc
importe
    Noeud
operateur
    constructeur
    arc : noeud x noeud -> arc
    acces
    orig : arc -> noeud
    dest : arc -> noeud
    egal : arc x arc -> booleen
precondition
```

[1] Cet héritage est qualifié de **Réalisation** en MECANO.
[2] L'équation (6) se lit "pour tout arc a, possede_arc(a,vide) == faux"

(1) precond(arc(n1,n2)) : n1 <> n2 *par convention*

axiome

(1) orig(arc(n1,n2)) == n1
(2) dest(arc(n1,n2)) == n2
(3) egal(arc(no1,nd1),arc(no2,nd2)) == si no1=no2 et nd1=nd2 alors vrai
(4) egal(arc(no1,nd1),arc(no2,nd2)) == si no1<>no2 alors faux
(5) egal(arc(no1,nd1),arc(no2,nd2)) == si nd1<>nd2 alors faux

fin type ARC

Type GRAPHE

sorte graphe

importe Noeud, Arc, Booleen, Entier, Liste(T)

rq: Liste(T) est un type générique qui fournit des opérations d'énumération: premier, suivant, est_dernier et les opérations d'ajout et de suppression d'un élément dans la liste. T sera instancié à NOEUD et ARC

operateur

constructeur

vide :	-> graphe
ajouter_noeud:	noeud x graphe -> graphe
ajouter_arc:	arc x graphe -> graphe

acces

est_vide:	graphe -> booleen
possede_noeud:	noeud x graphe -> booleen
possede_arc:	arc x graphe -> booleen
degre+:	noeud x graphe -> entier
degre-:	noeud x graphe -> entier
degre:	noeud x graphe -> entier
successeur:	noeud x graphe -> liste(noeud)
predecesseur:	noeud x graphe -> liste(noeud)
arcs_sortant:	noeud x graphe -> liste(arc)
arcs_entrant:	noeud x graphe -> liste(arc)

transformateur

retirer_noeud:	noeud x graphe -> graphe
retirer_arc:	arc x graphe -> graphe

conventions utilisées:

- un noeud ajouté est isolé (pas d'arc entrant ni sortant),
- les extrémités d'un arc ajouté sont elles-mêmes ajoutées si elles ne sont pas dans le graphe,
- lors du retrait d'un arc, les extrémités ne sont pas retirées,
- lors du retrait d'un noeud, tous les arcs entrants et sortants sont supprimés,
- l'ajout d'un noeud (resp. un arc) n'est pas défini pour un noeud (resp. un arc) déjà présent.
- le retrait d'un noeud (resp. un arc) n'est pas défini pour un noeud (resp. un arc) absent.

precondition

(1) precond(ajouter_noeud(n,g)) :
possede-noeud(n,g) = faux
(2) precond(ajouter_arc(a,g)):
possede_arc(a,g) = faux
(3) precond(degre+(n,g)) : possede_noeud(n,g) = vrai
(4) precond(degre-(n,g)) : possede_noeud(n,g) = vrai

(5) precond(degre(n,g)) : possede_noeud(n,g) = vrai
(6) precond(successeur(n,g)) : possede_noeud(n,g) = vrai
(7) precond(predesseur(n,g)) : possede_noeud(n,g) = vrai
(8) precond(arcs_entrant(n,g)) : possede_noeud(n,g) =. vrai
(9) precond(arcs_sortant(n,g)) : possede_noeud(n,g) = vrai
(10) precond(retirer_noeud(n,g)) : possede_noeud(n,g) = vrai
(11) precond(retirer_arc(a,g)) : possede_arc(a,g) = vrai

axiome

définition des observateurs sur les constructeurs

(1) possede_noeud(n,vide) == faux
(2) possede_noeud(n1,ajouter_noeud(n2,g)) == si n1=n2 alors vrai
(3) possede_noeud(n1,ajouter_noeud(n2,g)) ==
si n1<>n2 alors possede_noeud(n1,g)
(4) possede_noeud(n,ajouter_arc(a,g)) ==
si n=orig(a) ou n=dest(a) alors vrai
(5) possede_noeud(n,ajouter_arc(a,g)) ==
si n<>orig(a) ou n<>dest(a) alors possede_noeud(n,g)
(6) possede_arc(a,vide) == faux
(7) possede_arc(a,ajouter_noeud(n,g)) == possede_arc(a,g)
(8) possede_arc(a1,ajouter_arc(a2,g)) == si egal(a1,a2) alors vrai
(9) possede_arc(a1,ajouter_arc(a2,g)) ==
si non egal(a1,a2) alors possede_arc(a1,g)
(10) degre+(n1,ajouter_noeud(n2,g)) == si n1<>n2 alors degre+(n1,g)
(11) degre+(n1,ajouter_noeud(n2,g)) == si n1= n2 alors 0
(12) degre+(n,ajouter_arc(a,g)) ==
si n <> orig(a) et n <> dest(a) alors degre+(n,g)
(13) degre+(n,ajouter_arc(a,g)) ==
si n=orig(a) et possede_noeud(n,g)=vrai alors degre+(n,g) + 1
(14) degre+(n,ajouter_arc(a,g)) ==
si n=orig(a) et possede_noeud(n,g)=faux alors 1
(15) degre+(n,ajouter_arc(a,g)) ==
si n=dest(a) et possede_noeud(n,g)=vrai alors degre+(n,g)
(16) degre+(n,ajouter_arc(a,g)) ==
si n=dest(a) et possede_noeud(n,g)=faux alors 0

définition des observateurs sur les transformateurs

(17) possede_noeud(n1,retirer_noeud(n2,g)) == si n1=n2 alors faux
(18) possede_noeud(n1,retirer_noeud(n2,g)) ==
si n1<>n2 alors possede_noeud(n1,g)
(19) possede_noeud(n,retirer_arc(a,g)) == possede_noeud(n,g)
(20) possede_arc(a,retirer_noeud(n,g)) ==
si n=orig(a) ou n=dest(a) alors faux
(21) possede_arc(a,retirer_noeud(n,g)) ==
si n<>orig(a) et n<>dest(a) alors possede_arc(a,g)
(22) possede_arc(a1,retirer_arc(a2,g)) == si egal(a1,a2)=vrai alors faux
(23) possede_arc(a1,retirer_arc(a2,g)) ==
si non egal(a1,a2)=vrai alors possede_arc(a,g)
(24) degre+(n1,retirer_noeud(n2,g)) ==
si possede_arc(arc(n1,n2),g) = vrai alors degre+(n1,g) -1
(25) degre+(n1,retirer_noeud(n2,g)) ==
si possede_arc(arc(n1,n2),g) =faux et possede_noeud(n1,g) = vrai
alors degre+(n1,g)
(26) degre+(n,retirer_arc(a,g)) == si n=orig(a) alors degre+(n,g) - 1
(27) degre+(n,retirer_arc(a,g)) == si n <> orig(a) alors degre+(n,g)

autres
(28) degre(n,g) == degre+(n,g) + degre-(n,g)
(29) possede_noeud(n1,g) == si possede_arc(arc(n1,n2),g) alors vrai
(30) possede_noeud(n2,g) == si possede_arc(arc(n1,n2),g) alors vrai
(31) retirer_arc(a1,ajouter_arc(a2,g)) == si a1=a2 alors g
(32) retirer_noeud(n1,ajouter_noeud(n2,g)) == si n1=n2 alors g
(33) est_vide(vide) == vrai
(34) est_vide(ajouter_noeud(n,g)) == faux
(35) est_vide(ajouter_arc(a,g)) == faux
(36) est_vide(retirer_arc(a,g)) == faux

fin type GRAPHE

Remarques
Pour que la spécification soit complète il faudrait ajouter les axiomes concernant *predecesseur* , *successeur*, *arcs_sortant* et *arcs_entrant*. On pourra aussi ajouter, suivant les besoins, des opérateurs tels que:

sous_graphe: liste(noeud) x graphe -> graphe
rend le sous-graphe construit sur un ensemble de noeuds donné

graphe_partiel : liste(arc) x graphe -> graphe
rend le graphe partiel construit sur un ensemble d'arcs donné

est_arbre: graphe -> booleen
détermine si la graphe est un arbre

racine: graphe -> noeud
rend la racine du graphe considéré comme un arbre

relié: noeud x noeud x graphe -> booléen
existe-t-il un arc entre deux noeuds donnés

nb_noeud, nb_arc etc...

Exemple de réécriture à partir de la spécification de GRAPHE:

possede_noeud(n1,ajouter_noeud(n2,retirer_arc(a,retirer_noeud(n1,vide))))
== possede_noeud(n1,retirer_arc(a,retirer_noeud(n1,vide))) *équation (3)*
== possede_noeud(n1, retirer_noeud(n1,vide) *équation (19)*
== faux *équation (17)*

II) Classes EIFFEL équivalentes

Voici la traduction en EIFFEL des types abstraits présentés. En face des préconditions et postconditions (clause *require* et *ensure*) on trouvera le numero des préconditions ou axiomes équivalents du type abstrait.
La classe GRAPHE est une classe virtuelle décrivant le type GRAPHE indépendamment d'une représentation. La classe GRAPHE_CHAINE décrit une représentation au moyen de deux listes chainées (classe LINKED_LIST de la bibliothèque EIFFEL).

```
class NOEUD
-- noeud d'un graphe, un noeud porte une information numerique
export
  nd,numero,egal
feature
  Create(I: INTEGER) is
    -- remplace le constructeur nd
    do numero := I
    ensure
      numero = I                                          -- axiome (1)
    end;

  numero: INTEGER;
    -- remplace l'accès numero

  egal(Other: like Current) is
    do Result := Current.numero=Other.numero;
    end;

end -- class NOEUD
```

```
class ARC
export
  orig,dest,egal
feature
  Create(No,Nd: NOEUD) is
    -- remplace le constructeur arc
    require No/=Nd                                        -- precondition (1)
    do orig:= No; orig:= Nd;
    ensure
      Current.orig = No;                                  -- axiome (1)
      Current.dest = Nd;                                  -- axiome (2)
    end;

  orig, dest : NOEUD;
    -- implantation des accès orig et dest comme des attributs

  egal(Other : like Current): BOOLEAN is
    do Result := Current.orig.egal(Other.orig) and Current.dest.egal(Other.dest)
    ensure
      (Current.orig.egal(Other.orig) and Current.dest.egal(Other.dest))
      implies Result=true;                                -- axiome (3)
      not Current.orig.egal(Other.orig) implies Result=false;   -- axiome (4)
      not Current.dest.egal(Other.dest) implies Result=false;   -- axiome (5)
    end;

end -- class ARC
```

```
deferred class GRAPHE
  -- gestion d'un graphe oriente a base de noeuds et d'arcs
export
  ajouter_noeud, ajouter_arc, retirer_noeud, retirer_arc,
  est_vide,possede_noeud, possede_arc,
  degre_pos, degre_neg, degre,
  successeur, predecesseur, arcs_sortant, arcs_entrant,
  sous_graphe, graphe_partiel
feature

  -- Vide sera remplacé par le Create de la classe réalisation

  est_vide : BOOLEAN is deferred end;

  ajouter_noeud(N : NOEUD) is
    -- ajoute le noeud N dans le graphe
    require not possede_noeud(N)                          -- precondition (1)
    deferred
    ensure
      possede_noeud(N);                                     -- axiome (2)
      degre_pos(N) = 0;                                    -- axiome (11)
      not est_vide;                                        -- axiome (34)
    end;

  ajouter_arc(A:ARC) is
    -- ajoute l'arc au graphe
    require not possede_arc(A)                            -- precondition (2)
    deferred
    ensure
      possede_arc(A);                                       -- axiome (8)
      possede_noeud(A.orig) and possede_noeud(A.dest);      -- axiome (4)
      old current.possede_noeud(A.orig) implies
        degre_pos(A.orig)=degre_pos(old A.orig) +1;        -- axiome (13)
      not old current.possede_noeud(A.orig)
        implies degre_pos(A.orig)=1;                       -- axiome (14)
      old current.possede_noeud(A.dest) implies
        degre_pos(old A.dest)=degre_pos(A.dest);           --axiome (15)
      not old current.possede_noeud(A.dest) implies
        degre_pos(A.dest)=0;                               -- axiome (16)
      not est_vide;                                        -- axiome (35)
    end;

  possede_noeud(N : NOEUD) : BOOLEAN is
    -- le noeud N fait-il partie du graphe
    deferred
    end;

  possede_arc(A : ARC) : BOOLEAN is
    -- l'arc A fait-il partie du graphe
    deferred
    ensure
      Result implies possede_noeud(A.orig);                -- axiome (29)
      Result implies possede_noeud(A.dest);                -- axiome (30)
    end;
```

```
retirer_noeud(N : NOEUD) is
   -- retire le noeud N du le graphe
   -- les arcs adjacents sont retirés
   require possede_noeud(N)                          -- precondition (10)
   deferred
   ensure
     not possede_noeud(N);                                -- axiome (17)
     -- + équations (29)(30)(37)(38)
   end;

retirer_arc(A : ARC) is
   -- retire l'arc A du le graphe
   -- les noeuds correspondant ne sont pas atteints
   require possede_arc(A)                            -- precondition (11)
   deferred
   ensure
     not possede_arc(A);                                  -- axiome (22)
     degre_pos(A.orig) = degre_pos(old A.orig) -1         -- axiome (26)
     not est_vide;                                        -- axiome (36)
   end;

degre_pos(N: NOEUD) : INTEGER is
   -- nombre d'arcs partant du noeud N
   require possede_noeud(N)                          -- precondition (3)
   deferred
   end;

degre_neg(N) : INTEGER is
   -- nombre d'arcs arrivant au noeud N
   require possede_noeud(N)                          -- precondition (4)
   deferred
   end;

degre(N) : INTEGER is
   -- nombre d'arcs adjacents au noeud N
   require possede_noeud(N)                          -- precondition (5)
   deferred
   ensure
     Result = degre_pos(N) + degre_neg(N)                 -- axiome (28)
   end;

successeur(N: NOEUD) : LIST[NOEUD] is
   require possede_noeud(N)                          -- precondition (6)
   deferred
   end;

predecesseur(N: NOEUD): like successeur is
   require possede_noeud(N)                          -- precondition (7)
   deferred
   end;

arcs_sortant(N: NOEUD): LIST[ARC] is
   require possede_noeud(N)                          -- precondition (9)
   deferred
   end;
```

```
  arcs_entrant(N: NOEUD): like arcs_sortant is
    require possede_noeud(N)                          -- precondition (8)
    deferred
    end;

  graphe_partiel(EnsA: like arcs_entrant) : like Current is deferred end;

  sous_graphe(EnsN: like successeur) : like Current is deferred end;

end -- class GRAPHE
```

```
class GRAPHE_CHAINE
-- implantation de la classe GRAPHE
-- les arcs et les noeuds sont stockés dans des listes chainées

export
  repeat GRAPHE

inherit
  GRAPHE
  define
    ajouter_noeud, ajouter_arc, retirer_noeud, retirer_arc,
    est_vide,possede_noeud, possede_arc,
    degre_pos, degre_neg, degre,
    successeur, predecesseur, arcs_sortant, arcs_entrant,
    sous_graphe, graphe_partiel

feature

  Noeuds : LINKED_LIST[NOEUD];
    -- l'ensemble des noeuds du graphe est implanté par une liste chainée
    -- de NOEUD

  Arcs : LINKED_LIST[ARC];
    -- l'ensemble des arcs du graphe est implanté par une liste chainée d'arcs

  Create is
    -- création des deux listes Arcs et Noeuds initialisée à vide
    -- remplace le constructeur vide
    do
      Noeuds.Create; Arcs.Create;
    ensure
      est_vide;                                       -- axiome (33)
      Noeuds.empty;
      Arcs.empty;
    end;

  est_vide : BOOLEAN is
    do
      Result := Noeuds.empty        -- and Arcs.empty;
    end;
```

```
ajouter_noeud(N : NOEUD) is
  -- ajoute le noeud N dans le graphe
  require not possede_noeud(N)
  do
    Noeuds.put_right(N,Noeuds)   -- put_right: ajout d'un élément dans une liste
  ensure
    possede_noeud(N);
    degre_pos(N) = 0;
    not est_vide;
  end;

ajouter_arc(A:ARC) is
  -- ajoute l'arc au graphe
  require not possede_arc(A)
  do
    Arcs.put_right(A);
    if not possede_noeud(A.orig) then ajouter_noeud(A.orig);
    if not possede_noeud(A.dest) then ajouter_noeud(A.dest);
  ensure
    possede_arc(A);
    possede_noeud(A.orig) and possede_noeud(A.dest);
    old current.possede_noeud(A.orig) implies
       degre_pos(A.orig)=degre_pos(old A.orig) +1;
    not old current.possede_noeud(A.orig) implies
       degre_pos(A.orig)=1;
    old current.possede_noeud(A.dest) implies
       degre_pos(old A.dest)=degre_pos(A.dest);
    not old current.possede_noeud(A.dest) implies
       degre_pos(A.dest)=0;
    not est_vide;
  end;

possede_noeud(N : NOEUD) : BOOLEAN is
  -- le noeud N fait-il partie du graphe
  do
    Result := Noeuds.has(N);
  end;

possede_arc(A : ARC) : BOOLEAN is
  -- l'arc A fait-il partie du graphe
  do
    Result := Arcs.has(A);
  ensure
    Result implies possede_noeud(A.orig);
    Result implies possede_noeud(A.dest);
  end;

retirer_noeud(N : NOEUD) is
  -- retire le noeud N du graphe, les arcs adjacents sont retirés
  require possede_noeud(N)
  local arc_out, arc_in: like Arcs
  do
    Noeuds.start;            -- retrait du noeud du graphe
    Noeuds.search_same(N);
    Noeuds.remove;
```

```
      from        -- retrait des arcs sortant du graphe
         arc_out:= arcs_sortant(N); arc_out.start;
      until arc_out.offright
      loop
         retirer_arc(arc_out.item);
         arc_out.forth;
      end;
      from        -- retrait des arcs entrant du graphe
      arc_in:= arcs_entrant(N); arc_in.start;
      until
         arc_in.offright
      loop
         retirer_arc(arc_in.item);
         arc_in.forth;
      end;
   ensure
      not possede_noeud(N);
   end;

retirer_arc(A : ARC) is
   -- retire l'arc A du le graphe
   -- les noeuds correspondant ne sont pas atteints
   require possede_arc(A)
   do
      Arcs.start;          -- retrait de l'arc du graphe
      Arcs.search_same(A);
      Arcs.remove;
   ensure
      not possede_arc(A);
      degre_pos(A.orig) = degre_pos(old A.orig) -1
      not est_vide;
   end;

degre_pos(N: NOEUD) : INTEGER is
   -- nombre d'arcs partant du noeud N
   require possede_noeud(N)
   do
      Result := arcs_sortant(N).count;
   end;

degre_neg(N) : INTEGER is
   -- nombre d'arcs arrivant au noeud N
   require possede_noeud(N)
   do
      Result := arcs_entrant(N).count;
   end;

degre(N) : INTEGER is
   -- nombre d'arcs adjacents au noeud N
   require possede_noeud(N)
   do
      Result := arcs_sortant(N).count + arcs_entrant(N).count;
   ensure
      Result = degre_pos(N) + degre_neg(N)
   end;
```

```
  successeur(N: NOEUD) : like Noeuds is
    require  possede_noeud(N)
    do ...
    end;

  predecesseur(N: NOEUD): like successeur is
    require  possede_noeud(N)
    do ...
    end;

  arcs_sortant(N: NOEUD): like Arcs is
    require  possede_noeud(N)
    do ...
    end;

  arcs_entrant(N: NOEUD): like arcs_sortant is
    require  possede_noeud(N)
    do ...
    end;

  graphe_partiel(EnsA:  like Arcs) : like Current is
    local GP: like  Current
    do
    from
      GP.Create;  EnsA.start
    until
      EnsA.offright
    loop
      GP.ajouter_arc(EnsA.item) -- les noeuds sont ajoutés automatiquement
      EnsA.forth
    end;
    ensure
      GP.Arcs = EnsA
    end;

  sous_graphe(EnsN:  like Noeuds) : like Current is
    do ...
    end;

invariant
  est_vide implies  Arcs.empty;
  est_vide implies  Noeuds.empty;
  Arcs.count <= Noeuds.count * (Noeuds.count - 1) ;
  -- invariant de tout graphe

  -- axiome (29) : pour tout a de Arcs, a.orig appartient à Noeuds
  -- axiome (30) : pour tout a de Arcs, a.dest appartient à Noeuds

end -- class GRAPHE_CHAINE
```

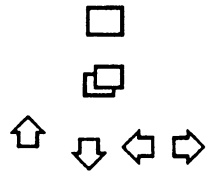
Remarques

le domaine de sortie de successeur et predecesseur dans GRAPHE est LIST[NOEUD]. Ce domaine est specialise en LINKED_LIST[NOEUD] dans la réalisation GRAPHE_CHAINE (covariance)

Il a été ajouté un invariant dans la classe GRAPHE_CHAINE qui permet d'exprimer la relation entre le nombre Nbn de noeuds d'un graphe et le nombre Nba d'arcs: Nba <= Nbn * (Nbn - 1).

ANNEXE B

SYMBOLES UTILISES EN MECANO



fermeture de l'entité (fenêtre)

retaillage de la fenêtre

flèches de scrolling

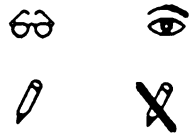
début / fin edition

visualisation/masquage zône liste

visualisation/masquage palette d'édition

vérification de complétude

Icônes d'action



mode concepteur / mode utilisateur

fenêtre éditable/non editable

Icônes "mode de travail"

classe concrète
classe concrète générique
classe virtuelle
classe virtuelle générique
composite
domaine
application
projet

Icône "type de l'entité"

plot d'appartenance à un composite

plot d'appartenance à un domaine

plot d'appartenance à une application

Plots d'appartenance

ANNEXE C

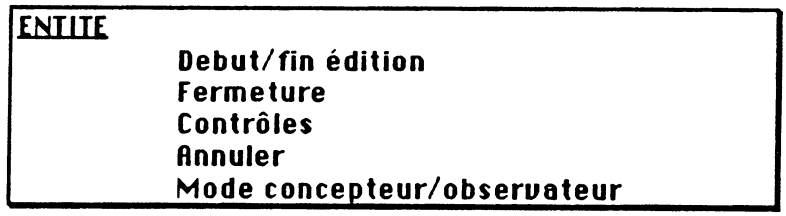
MENUS ET OPERATIONS SUR LES ENTITES

Nous détaillons dans cette partie l'ensemble des opérations activables depuis une entité. Une barre de menu est associée à chaque type d'entité. La barre de menu contient une liste de sous-menus déroulants permettant d'activer les opérations voulues dans le contexte de cette entité. Les menus des entités complexes (projet, domaine, application, composite) sont similaires.

I) MENU GENERAL DES ENTITES COMPLEXES

Les menus suivants sont présents dans toutes les entités graphiques (complexes) : projet, domaine, composite, application. Les choix proposés dans certains menus peuvent ne pas apparaître dans un type d'entité, à l'inverse il peut y avoir des opérations spécifiques à un type d'entité, elles seront décrites dans la suite.
La barre de menu d'une entité complexe se présente sous la forme suivante:

MENU ENTITE
Ce menu offre des opérations générales sur l'entité représentée dans la fenêtre. Le titre de ce menu sera le type de l'entité: **PROJET**, **DOMAINE**, **COMPOSITE** ou **APPLICATION**.



début/fin édition
l'entité passe en debut ou fin d'édition. En fin d'édition, les changements sont pris en compte.

fermeture
fermeture de l'entité, la fenêtre disparait après enregistrement des changements.

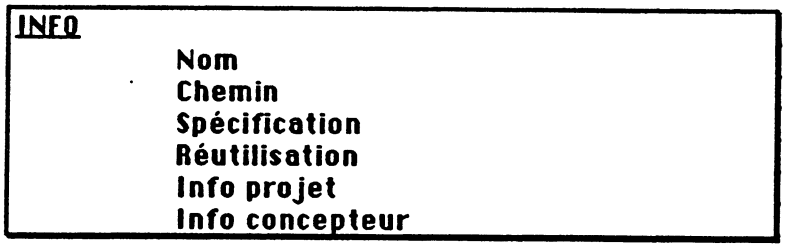
contrôles
apparition d'une boite de dialogue pour demander le type de contrôle à effectuer. Il s'agit de la vérification de certaines règles de la méthode et du contrôle de complétude.

annuler
annule la dernière opération effectuée si c'est possible

mode concepteur/observateur
l'entité passe en mode concepteur (accès complet) ou observateur (accès restreint, édition interdite). Le mode utilisateur ne permet pas, par exemple, de voir les classes externes des domaines, ni la partie définition des méthodes et attributs d'une classe.

MENU INFO

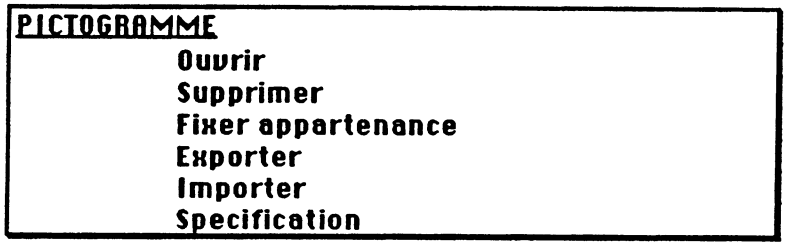
Ce menu permet la visualisation et l'édition d'informations relatives à l'entité.
Chaque alternative produit l'affichage d'une boite de saisie ou d'une fenêtre de texte simple éditable ou non. Aucune information n'est éditable en mode observateur.



nom	nom de l'entité (éditable)
chemin	liste des entités depuis la racine (le projet) - non éditable
spécification	ouvre une fenêtre éditable - rôle de l'entité dans la conception
réutilisation	information sur la (ré)utilisation de l'entité ou d'une sous-entité
info projet	date, auteur, version (sous forme attribut/valeur)
info concepteur	possibilité d'ajout d'attributs personnel (utilisable dans la navigation)

MENU PICTOGRAMME

Il permet de lancer des opérations liées au pictogramme courant (sélectionné en inverse vidéo). Ce même menu est disponible directement sur le pictogramme (bouton central).



ouvrir
ouvre l'entité en forme complète (zooming)

supprimer
supprime le pictogramme de l'image et de l'entité courante, ainsi que les liens associés. S'il s'agit d'une entité complexe, les sous-entités internes sont aussi supprimées. Il y a demande de confirmation de MECANO, après affichage des effets de bord.

fixer appartenance
ne s'applique qu'à une entité externe. Transforme l'entité externe en entité interne, l'entité devient externe dans son contexte d'origine et reste externe dans tous les autres contextes. Les règles de constitution doivent être respectées.

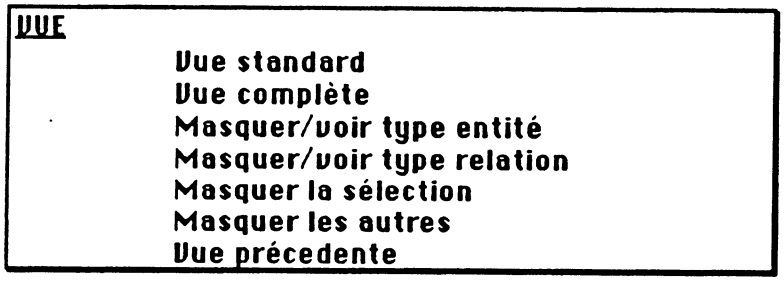
exporter
copie l'entité en vue d'un transfert

importer
ajoute en tant qu'entité externe, une entité préalablement copiée en transfert dans un autre contexte

specification
visualisation non éditable de la spécification de l'entité sélectionnée

MENU VUE

Contient les opérations modifiant l'affichage: masquage de relations et d'entités. Seules les options de masquage sont activables en mode observateur.



vue standard
retour à la vue standard relative à l'entité. Cette vue dépend du type de l'entité

vue complete
toutes les entités externes et internes sont représentées ainsi que toutes les relations (non disponible en mode observateur)

masquer / voir type entité
boite de dialogue demandant les types des entités à masquer ou à afficher

masquer/voir type relation
boite de dialogue demandant les types de relations à masquer ou à afficher

masquer la selection
masque les pictogrammes sélectionnés

masquer les autres
masque les pictogrammes non sélectionnés

vue precedente
retour à la vue précédente (avant la dernière demande de modification)

MENU FORMAT

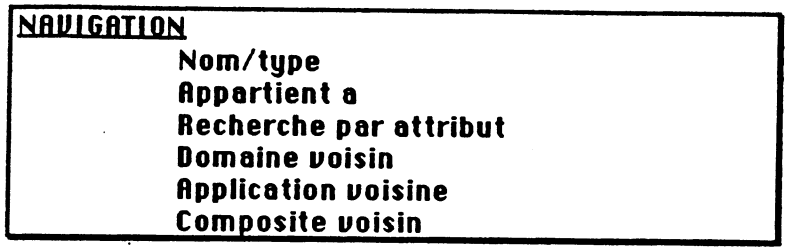
Sélection des formats de caractères (taille, police et style) et de la taille des pictogrammes. Le format s'applique à la sélection courante ou à l'ensemble des pictogrammes si rien n'est sélectionné.

Taille, Police, Style
le format est appliqué aux noms des pictogrammes et aux labels de relations

Picto / Mini picto
passe en mode pictogramme réduit ou normal

MENU NAVIGATION

Recherche (browsing) des entités à partir de l'entité sélectionnée. Si aucun pictogramme n'est sélectionné, la recherche s'applique, quand c'est possible, à l'entité courante (la fenêtre).
En général une liste de noms d'entités est obtenue dans une boite spéciale de choix, l'utilisateur peut alors en sélectionner un et ouvrir l'entité correspondante.



nom/type
entité recherchée par son nom et son type. La recherche est circonscrite à l'entité courante et aux sous-entités

appartient à
recherche de l'entité d'appartenance de l'entité courante

recherche par attribut
permet de rechercher une entité par les attributs prédéfinis ou personnels présents dans le menu INFO (info concepteur/ info projet) ou dans le menu RUBRIQUE pour les classes

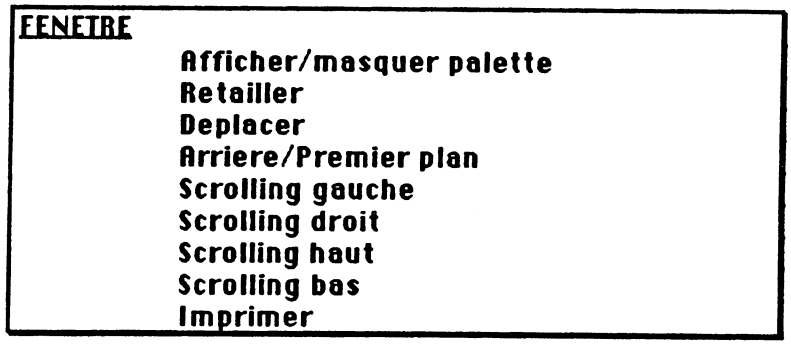
domaine voisin
liste des domaines possèdant une entité externe de l'entité courante (domaine utilisateur)

application voisine
liste des applications possèdant une entité externe de l'entité courante (application utilisatrice)

composite voisin
liste des composites possèdant une entité externe de l'entité courante

Les autres critères de recherche dépendent du type de l'entité courante.

MENU FENETRE
Opérations sur la fenêtre



II) SPECIALISATION SELON LES TYPES D'ENTITE COMPLEXE

Toute entité possède un menu local à sa représentation pictogramme ainsi qu'une série de menus présents dans la barre de menu de sa représentation complète. Nous détaillons ici les menus et opérations spécifiques à chacune des entités :

• PROJET
Le projet est l'entité complexe de plus haut niveau. Il n'y a qu'une seule fenêtre projet. Les fonctions disponibles dans cette fenêtre regroupent les opérations liées au projet ainsi que celles liées à l'application MECANO (concernant l'environnement de travail).

Menu Projet (menu entité)

La clé du MECANO	information sur le poste MECANO
nouveau	crée un nouveau projet
ouvrir	ouvre un nouveau projet (remplace le projet courant)
enregistrer	enregistre le projet actuel sous son nom
enregistrer sous	enregistre le projet actuel sous un autre nom
quitter	ferme la fenêtre projet et quitte l'application MECANO

Menu Pictogramme
fixer appartenance et *exporter* ne sont pas disponibles
importer permet d'obtenir la **copie** d'une entité d'un autre projet (fenêtre de dialogue)
voir racine ouvre la racine du domaine ou de la classe sélectionnée

Menu Vue
pas de menu vue

• **APPLICATION**

Menu Application
idem Menu entité présenté plus haut

Menu Info
Pas de rubrique *reutilisation*
Rubriques supplémentaires

info application	informations statistiques sur l'application: nombre de classes internes nombre de classes externes nombre de lien d'héritage, etc...
classe démarrage	nom de la classe qui sera instanciée au début de l'exécution de l'application, cette classe doit être la racine de l'application

Menu Vue
La vue standard présente les classes et composites internes et externes reliés par des relations clients (masquage des classes héritées).

• **DOMAINE**

Menu Domaine
idem Menu entité

Menu Info
Rubriques supplémentaires

info domaine	informations statistiques sur le domaine
classe racine	nom de la classe racine du domaine

Menu Vue
La vue standard présente l'arbre de classes (composites) internes sur les relations de spécialisation, réalisation et adaptation.

En plus, certaines vues très utiles aux domaines sont proposées directement:
masquer/voir utilisation
permet de ne pas surcharger l'image par les liens d'utilisation
masquer/voir classe concrète
vue utilisateur créateur de classe (Cf Chapitre 4)
masquer/voir classe virtuelle
vue utilisateur simple

• **COMPOSITE**

Menu Composite
idem Menu entité

Menu Info
Rubriques supplémentaires
classe racine type (concrète,virtuelle,générique) de la classe racine du composite
Menu Vue
La vue standard présente l'arbre de classes internes en relation de composition.

Remarque: d'autres possibilités de navigation (classe utilisatrice, classe héritière etc...) sont disponibles dans le menu navigation de la classe racine du composite.

III) MENUS DES CLASSES

Une fenêtre de classe possède la barre de menu suivante:

CLASSE	RUBRIQUE	ELEMENT	VUE	FORMAT	NAVIGATION	FENETRE

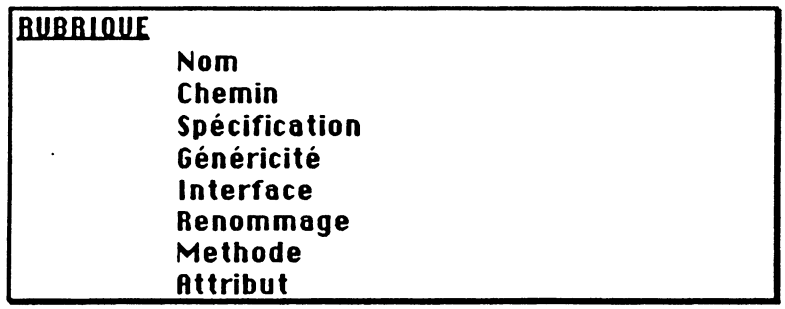
MENU CLASSE
Ce menu propose des opérations générales sur la classe. Il est identique au menu *entite* des entités complexes.

Le lancement de la requête de contrôle affiche une boite de choix sur les contrôles possibles:
- la complétude:
  complétude de conception détaillée: les méthodes de l'interface sont réalisées.
  tous les liens d'utilisation se traduisent par des attributs de type correct (classe destination).
- cohérence
  généricité graphique et textuelle
- autre vérifications

Le mode observateur interdit toute édition ainsi que l'accès aux rubriques méthodes, attributs et renommage.

MENU RUBRIQUE
Ce menu fournit l'accès aux différentes rubriques constituantes d'une classe. Les rubriques sont des informations textuelles décrivant la sémantique de la classe (cf Chapitre 4). La sélection d'une rubrique fait apparaître dans la zone liste l'ensemble des

éléments de la rubrique pour la classe courante. L'utilisateur peut alors sélectionner, ouvrir, créer ou détruire des éléments de cette liste (menu **element**).



nom	nom de la classe courante
chemin	chemin complet d'appartenance depuis le projet (non éditable)
specification	liste des éléments de la rubrique spécification
genericite	liste des paramètres génériques formels
interface	liste des méthodes et attributs exportés.
renommage	liste des caractéristiques renommées
methode	liste des méthodes de la classe et liste des classes héritées pour l'accès aux méthodes héritées.
attribut	liste des attributs de la classe et liste des classes héritées pour l'accès aux attributs hérités.

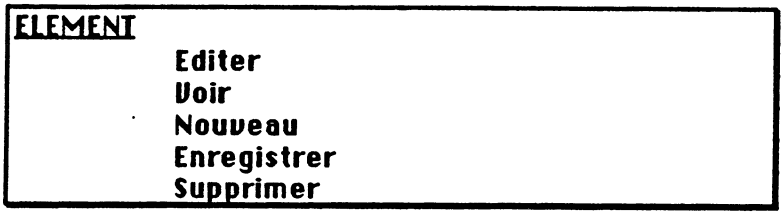
Fonctionnement des rubriques interface, renommage
Lors de la création d'un nouvel élément, une fenêtre de choix apparait pour proposer la liste des éléments possibles.
Par exemple, à la création d'un nouvel export, une liste contenant les méthodes et attributs de la classe déjà créés (rubrique méthode et attribut), ainsi que ceux des classes héritées apparait. L'utilisateur peut sélectionner un ou plusieurs éléments de cette liste qui viendront s'ajouter aux exports de la classe courante ou bien il clique sur **nouveau** pour créer une nouvelle méthode.
Pour le renommage, une fenêtre donne accès aux caractéristiques héritées via à la liste des ancêtres de la classe. Le concepteur choisit une classe puis la caractéristique qu'il désire renommer. Une boite de dialogue apparait pour saisir le nouveau nom.

Fonctionnement des rubriques attribut et méthode
La liste propose les méthodes ou attributs de la classe ainsi qu'un accès (recopie) aux méthodes héritées via les classes parentes. Ceci permet, par exemple, de travailler directement sur le code d'une méthode redéfinie, ou bien de visualiser une méthode héritée particulière.
Lors de la création d'une nouvelle méthode, le système présente une boite de sélection avec la liste des méthodes actuellement exportées. Le concepteur peut alors en sélectionner une, ce qui permet de récupérer les informations de l'interface (fonction ou procédure, nom, paramètres, type résultat, spécification informelle).

Le concepteur peut donc aussi bien créer de nouvelle caractéristique depuis la rubrique *interface* (création de l'abstraction d'abord) que depuis les rubriques *attribut* et *methode* (construction de l'implantation d'abord).

MENU ELEMENT
Il s'agit des opérations sur la liste d'éléments affichés pour une rubrique donnée. Ce menu permet de créer, détruire ou voir un élément de la liste. L'élément apparait dans la partie droite de la zone texte de la classe pour être édité. La zone texte s'efface à chaque visualisation d'un nouvel élément. Afin de visualiser simultanément plusieurs

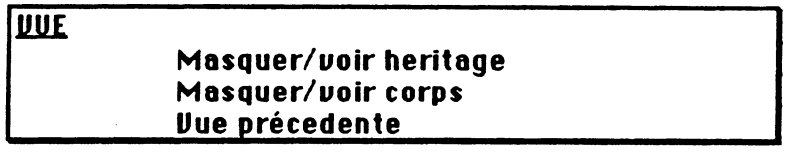
informations concernant la même classe, il est possible de faire apparaître l'élément dans une fenêtre auxiliaire non éditable. Cette fenêtre (dépendante de l'entité courante) reste présente jusqu'à la fermeture de la classe ou de la fenêtre elle-même.



editer
affiche en édition l'élément selectionné dans la zone texte à droite
voir
affiche l'élément sélectionné dans une fenêtre dépendante auxiliaire
nouveau
crée un nouvel élément dans la liste (cf §Fonctionnement des rubriques, ci-dessus)
enregistrer
prend en compte les modifications effectuées sur l'élément, sinon elle sont perdues à la sélection d'un autre élément après confirmation sur boite de "warning".
detruire
supprime l'élément selectionné.

remarque: *voir* et *supprimer* peuvent s'appliquer à une multi-sélection

MENU VUE
Les opérations du menu vue s'appliquent à la liste des éléments visibles dans la partie gauche ou à la visualisation des éléments eux mêmes.

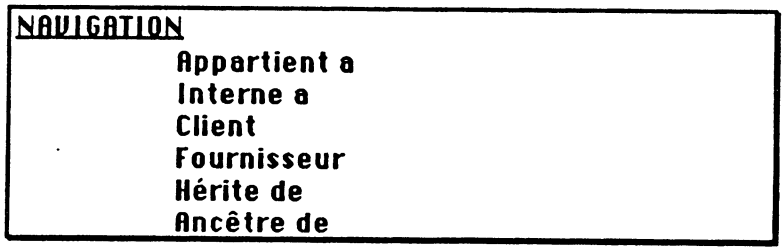


masquer héritage
la liste de caractéristiques affichée par les rubriques *méthode* et *attribut* ne contient que les caractéristiques propres (non héritées).
masquer corps
les méthodes sont visualisées (opération voir) sans le corps (définition et local). N'a pas d'effet sur les éléments déjà visualisés.
vue précédente
retour à la vue précédente (avant la dernière demande de modification)

MENU FORMAT
choix des formats de caractères et application au texte sélectionné (inverse vidéo)

MENU NAVIGATION

Recherche des entités à partir de la classe.



appartient à	accès à l'entité englobante de la classe
interne à	liste des entités complexes possèdant une forme externe de l'entité (référence)
client	liste des classes clientes
fournisseur	liste des classes fournisseurs
hérite de	liste des classes dont hérite la classe courante
ancêtre de	liste des classes qui héritent de la classe courante

Pour les alternatives *client*, *fournisseur*, *hérite de* et *ancêtre de*, le système demande à l'utilisateur les types d'héritage ou d'utilisation désirés.

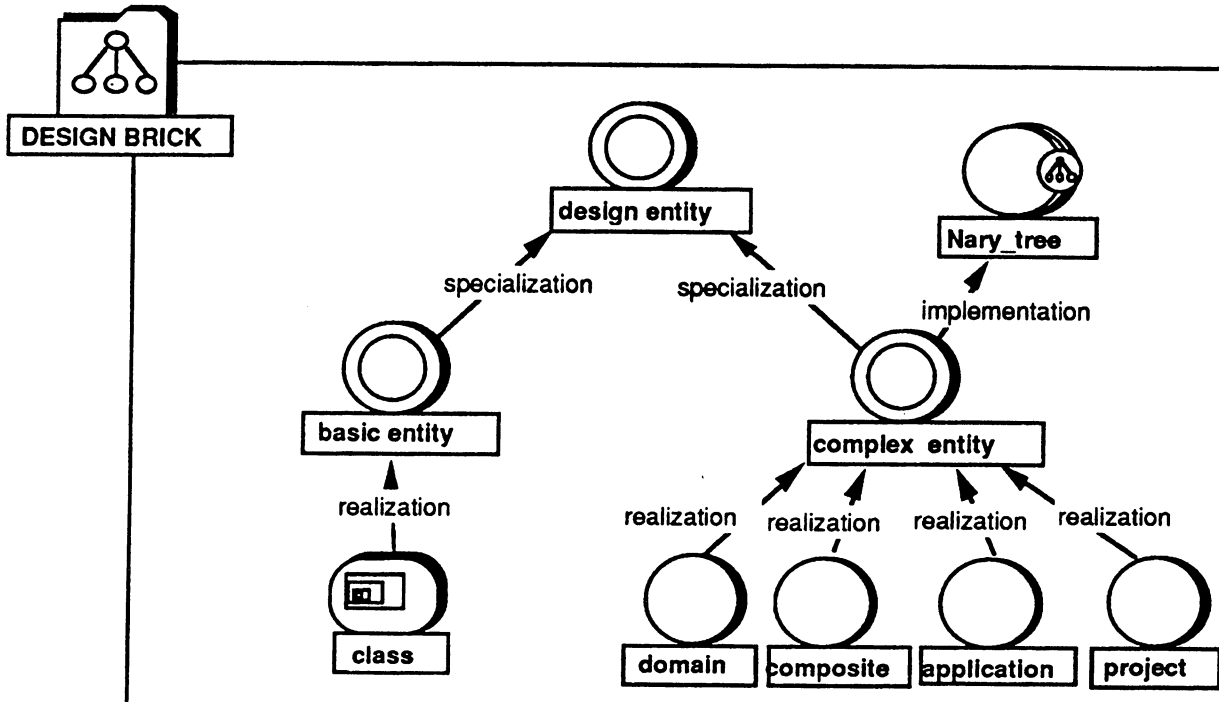
MENU FENETRE

même menu que pour les entités complexes

ANNEXE D

CONCEPTION DU POSTE: EXEMPLES

Les schémas suivants sont tirés de la conception du poste de conception MECANO.



SPECIFICATION

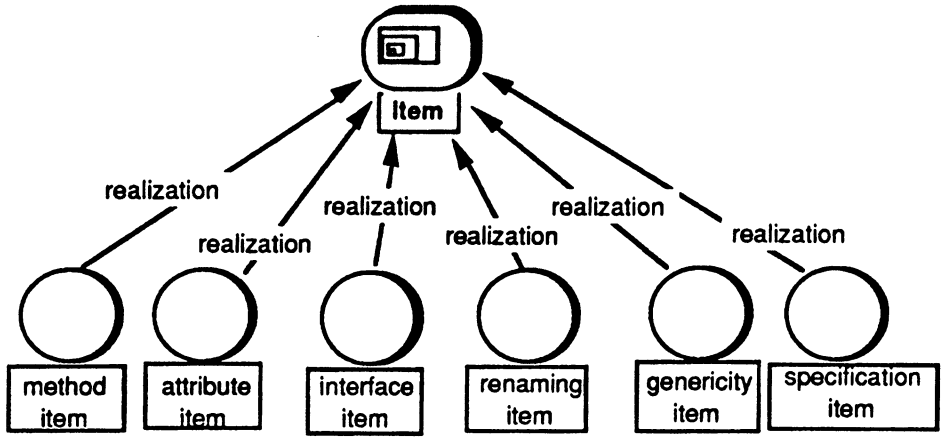
le domaine DESIGN BRICK contient les entités manipulées par la methode MECANO (représentation interne). En particulier toutes les informations saisies par le concepteur (graphique ou textuelle) sont encapsulées par l'entité. Chaque entité de design créée par un concepteur MECANO donnera naissance à une instance d'une des classes réalisation du domaine (class, composite, ...) .

Les entités complexes sont implantées par une structure d'arbre en liaison avec le concept d'appartenance de MECANO. Elles donneront lieu à une représentation externe graphique.

Les entités basiques sont terminales (au sens de l'arborescence d'appartenance), la représentation externe est textuelle (rubriques structurées) et se decompose dans un composite.

La classe NARY_TREE provient du domaine des structures arborescentes arbres.

ITEM

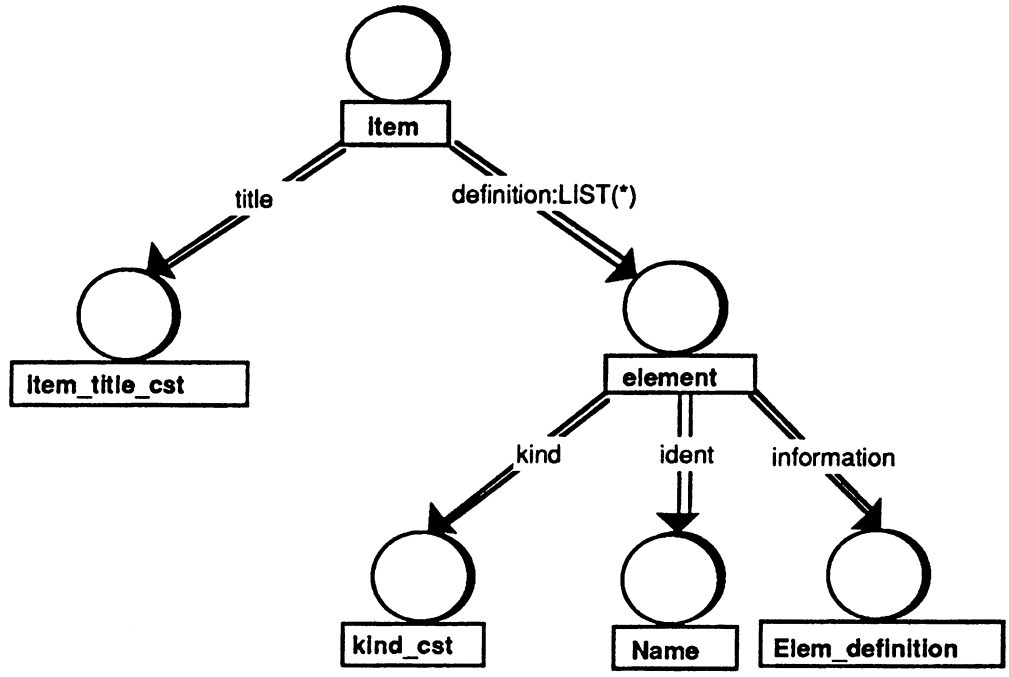


SPECIFICATION

le domaine ITEM decrit les rubriques (items) possibles pour une entité basique. Un item est toujours composé d'un titre et d'une definition. La definition est une liste d'élément.

L'item est specialisé en diverses rubriques, constituant les informations associées a une classe

ITEM

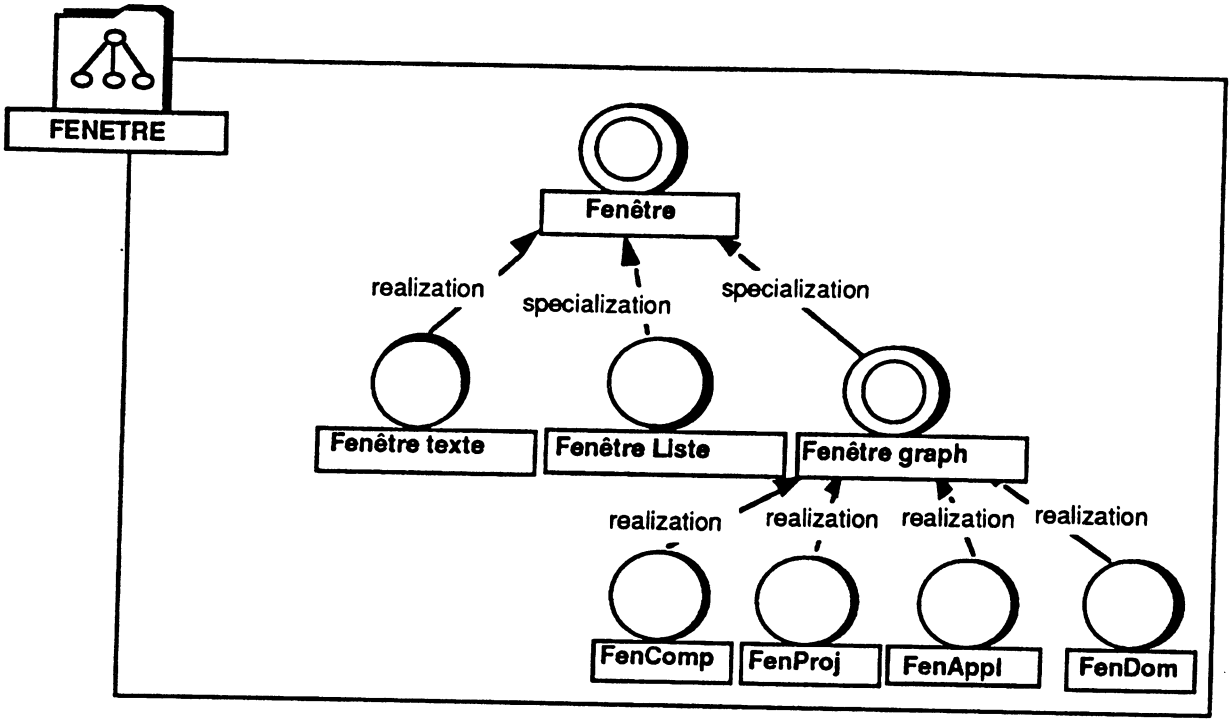


SPECIFICATION

le composite ITEM, permet de décrire les rubriques constituantes des entites basiques. Il possède un titre (specification, method, attribute, ...) et une definition. La definition est une liste d'elements (instanciation de la classe generique des listes par le parametre effectif "ELEMENT").

L'élément est decomposé en un genre (kind) , un identificateur (ident), et une definition (information). Exemple: element : method / Avancer / param: p1:T1;specif: ...

L'information peut être une liste de définition ou une definition seule (selon l'élément considéré).



Domaine de fenêtre (tiré de la partie interface graphique)

BIBLIOGRAPHIE

[Adam 87]
J.M. Adam. "Non-functional Constraints in ASPIS Context", *L.G.I. Contribution to ASPIS Project,* Contrat ESPRIT 401, 1987.

[Adam 88]
J.M. Adam. "A Selection of Software Metrics to Evaluate the Quality of a Design", *Rapport final,* annexe A, Contrat ESPRIT 401, 1988.

[Adiba 89a]
M. Adiba & C. Collet."L'Approche Objet pour les Bases de Données", *Survey - SUR001, Rapport de Recherche IMAG,* Avr. 1989.

[Adiba 89]
M. Adiba. "Les Nouveaux SGBD, Extensibilité et Orientation Objet", *Séminaire de Formation Aerospatiale,* Montrouge, dec. 1989.

[ANSI 83]
ANSI/-MIL-std 1815-a. "Reference Manual for the Ada Programming Language", United States Department of Defense, 1983.

[Bezivin 90]
J. Bézivin. "Teaching Object-Oriented Methods",*Panel in Technology of Object-Oriented Languages and Systems*, second international conference, Paris, Jun 1990.

[Blake 87]
E. Blake & S. Cook. "On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk",*European Conference on Object-Oriented Programming*, pp 41-50, 1987.

[Bobrow 83]
D.G. Bobrow & M. Stefik. "The LOOPS Manual: A Data and Object-oriented Programming System for Interlisp", *Knowledge-Based VLSI Design Group* Memo KB-VLSI-81-13, Xerox PARC, Palo Alto, California, 1983.

[Bois 89]
H. Bois. "Specification d'une Méthode Fondée sur le Concept d'Objets", *3èmes Journées "Pratique des méthodes et outils d'aide à la conception des systèmes d'information"*, Nantes, sept. 1989.

[Booch 83]
G. Booch. "Object-Oriented Design", *Tutorial on Software Design Techniques*, 4th ed, 1983 & "Software Engineering with Ada", *Benjamin/Cummings Pub. Company*, 1983.

[Booch 86]
G. Booch. "Object-oriented Development", *IEEE Transaction on Software Engineering*, Vol SE-12, Feb. 86.

[Borland 89]
Borland International. "Programmation Orientée Objets" et "Manuel de Reference", *Turbo Pascal version 5.5, Borland International*, 1989.

[Boussard 90]
J.C. Boussard, C. Michel, J.P. Partouche, R. Rousseau & M. Rueher. "Une Evaluation du Langage Eiffel", *Technique et Science Informatique*, vol. 9, n° 4, 1990.

[Brien 87]
P.D. O'Brien, D.C. Halbert & M.F. Kilian. "The Trellis Programming Environment", *Object-Oriented Programming Systems, Languages and Applications Proceedings*, pp. 91-102, oct. 1987.

[Caromel 90]
D. Caromel. "Concurrency: An Object-Oriented Approach".*Technology of Object-Oriented Languages and Systems*, second international conference, Paris, Jun. 1990.

[CCM 87]
Cri / Cisi-Ingéniérie / Matra . "HOOD manual 2.0", *internal report*, Jun. 1987.

[Coad 91]
P. Coad & E. Yourdon. "Object-Oriented Analysis", *Second Edition, Yourdon Press*, 1991.

[Cointe 87]
P. Cointe. "Metaclasses are First Class: The ObjVlisp Model", *Proceedings of the 2nd OOPSLA*, Orlando, Florida, 1987.

[Coutaz 89]
J. Coutaz. "Interface Homme-Ordinateur: Conception et Réalisation", *Thèse d'Etat, Université J. Fourier*, Grenoble, 1989.

[Cox 86]
B.J. Cox. "Object Oriented Programming, An Evolutionary Approach", *Addison-Wesley*, 1986.

[Cox 89]
B.J. Cox. "Planning the Software Industrial Revolution", *Technology of Object-Oriented Languages and Systems*, first international conference, Paris, Nov 1989.

[Dahl 66]
O.J. Dahl & K. Nygaard. "SIMULA - An Algol-Based Simulation Language", *Communications of the ACM*, 1966.

[Dahl 84]
O.J. Dahl, B. Myhrhaug & K. Nygaard. "SIMULA 67 Common Base Language", *Norwegian Computing Center*, Feb. 1984.

[De Mare 90]
C. De Mare. "Parallelisme dans les Langages à Objets", *Probatoire en Informatique CNAM*, mai 90.

[Dijkstra 76]
E. Dijkstra. "Structured Programming", *Software Engineering, Concepts and Techniques*, Van Nostrand Reinhold, 1976.

[Di Maio 89]
A. Di Maio, F. Bott, I. Sommerville, R. Bayan & M. Wirsing. "The DRAGOON Project", *présenté à ESPRIT Conference*, 1989.

[Ducournau 88]
R. Ducournau. "YAFOOL", *Version 3.22, Manuel de référence.* SEMA.METRA, Montrouge, 1988.

[Dugerdil 88]
P. Dugerdil. "Contribution à l'Etude de la Représentation des Connaissances Fondée sur les Objets - Le Langage OBJLOG", *Thèse de L'Université d'Aix-Marseille II*, 1988.

[Estublier 88]
J. Estublier & N. Belkhatir. "NOMADE: Noyau de Maintenance et de Developpement", *Journées Internationales, le Génie Logiciel et ses Applications*, Toulouse, Dec. 1988.

[Froidevaux 90]
C. Froidevaux, M.C. Gaudel & M. Soria. "Types de Données et Algorithmes", *McGraw-Hill*, Paris, 1990.

[Fukunaga 86]
K. Fukunaga & S. Hirose. "An Experience with a Prolog-based Object-Oriented Language", *Proceeding of the 1st OOPSLA*, Portland, Oregon, 1986.

[Gendre 90]
P. Gendre & H. Bitteur. "HOOD Version 3.0: l'Age de Raison", *Génie Logiciel et Systèmes Experts*, N° 19, juin 1990.

[Girod 87a]
X. Girod & P. Morat. "Design Methods and Tools", *L.G.I. Contribution to ASPIS Project*, Contrat ESPRIT 401, 1987.

[Girod 87b]
X. Girod. "AMPHI : Abstract Machine and Process Hierarchies", *L.G.I. Contribution to ASPIS Project*, Contrat ESPRIT 401, 1987.

[Girod 88]
X. Girod. "La Dynamique de Conception: Actions et Stratégies", *L.G.I. Contribution to ASPIS Project*, Contrat ESPRIT 401, 1988.

[Girod 89]
X. Girod. "MECANO: Méthode de Conception d'Applications par Objets". *Rapport contrat L.G.I.-H.P.* , dec. 1989.

[Girod 90a]
X.Girod. "MECANO: Spécification du poste de conception". *Rapport contrat L.G.I.-H.P.* , mar. 1990.

[Girod 90b]
X. Girod. "MECANO: A Method for Object-Oriented Software Construction". *Technology of Object-Oriented Languages and Systems*, second international conference, Paris, Jun. 1990.

[Girod 90c]
X. Girod. "Classification de l'Héritage et Structuration pour la Conception Objet". *Congrès DI 90*, Dijon, Oct. 1990.

[Girod 91]
X. Girod & P. Morat. "The MECANO Design Station: A Case Tool For Object-Oriented Construction". *Article soumis à EurOOPe'91*, Bratislava, sep. 1991.

[Goldberg 83]
A. Goldberg & D. Robson. "Smalltalk-80: The Language and it's Implementation", *Addison-Wesley*, 1983.

[Guttag 77]
J.V. Guttag. "Abstract Data Types and the Development of Data Structures", *Communications of the ACM*, vol. 20, no. 6, Jun. 1977.

[Halbert 87]
D.C. Halbert & P.D. O'Brien. "Using Types and Inheritance in Object-Oriented Programming". *IEEE Software*, pp 71-79, Sep. 1987.

[Heitz 87]
M. Heitz. "HOOD, une Méthode de Conception Hiérarchisée Orientée Objets pour le Développement des gros Logiciels Techniques et Temps-Réel. *Journées ADA France*, Bigre n°57, Dec. 1987.

[Henderson-Sellers 90]
B. Henderson-Sellers & J.M. Edwards. "The Object-Oriented Systems Life-Cycle", *Communications of the ACM* , vol.33, N°9, Sep. 1990.

[Henry 89]
C. Henry & M. Lott. "La Méthode de Conception MACH2", *Congrès de Génie Logiciel CGL4*, Toulouse, 1989.

[Hullot 85]
J.M. Hullot "CEYX, version 15. II - Programmer en CEYX", *Rapport Technique 45 INRIA*, Rocquencourt, 1985.

[IGL 82]
"Introduction à SADT", *IGL Technology*, 1982.

[IGL 84]
"Introduction à la méthode MACH", *IGL Technology*, 1984.

[ISE 90a]
"EIFFEL: The Libraries", *Interactive Software Engineering*, Version 2.3, Octobre 1990.

[ISE 90b]
"EIFFEL: The Language", *Interactive Software Engineering*, Version 2.3, 1990.

[Jackson 83]
M.A. Jackson. "System Development", *Prentice-Hall International, Englewood Cliffs*, 1983.

[Jacobson 88]
I. Jacobson & P. Jonsson. "Object Oriented System Development with ObjectOry: an Example", *Objective Systems*, SF AB 1988.

[Jacobson 89]
I. Jacobson. "An Object Oriented System Architecture with ObjectOry", *Objective Systems*, SF AB 1989.

[Jacobson 91a]
I. Jacobson. "Industrial Development of Software with an Object-Oriented Technique", *Journal of Object-Oriented Programming*, Mar/Apr. 1991.

[Jacobson 91b]
I. Jacobson. "Object-Oriented Software Engineering", *Addison-Wesley,* à paraître, automne 1991.

[Jacquet 78]
P. Jacquet. "Les Types Génériques, Propositions pour un Mécanisme d'Abstraction dans les Langages de Programmation", *Thèse de 3ème cycle,* USTMG,1978.

[Jalote 89]
P. Jalote. "Functional Refinement and Nested Objects for Object-Oriented Design", *IEEE Transactions on Software Engineering*, vol. 15, N° 3, mar. 1989.

[Keene 89]
S.E. Keene. "Object Oriented Programming in Common Lisp. A Programmer's Guide to CLOS", *Addison-Wesley*, Reading, Massachusetts, 1989.

[Kirkerud 89]
B. Kirkerud. "Object-Oriented Programming With Simula", *Addison-Wesley*, 1989.

[Korson 90]
T. Korson & J.D McGregor. "Understanding Object-Oriented: A Unifying Paradigm", *Communications of the ACM* , vol.33, N°9, Sep. 1990.

[Krakoviak 87]
S. Krakoviak, R. Balter, M. Meysembourg, C. Roisin, X. Rousset de Pina, R. Scioville & G. Vandôme. "Principes de conception du système d'exploitation réparti Guide".*Rapport Guide-R1, LGI* , 1987.

[Kristensen 87a]
B.B. Kristensen, O.L. Madsen, B. Moller-Pedersen, K. Nygaard. "Classification of Actions or Inheritance also for Methods", *European Conference on Object-Oriented Programming, ECOOP'87 Proceedings*, pp 98-107, Paris, jun. 1987.

[Kristensen 87b]
B.B. Kristensen, O.L. Madsen, B. Moller-Pedersen, K. Nygaard. "The Beta Programming Language". *B. Schriber & P. Wegner, editors, Research Directions in Object Oriented Programming,* MIT Press, Cambridge, Massachusetts, 1987.

[Larousse 77]
Larousse Encyclopédique en Couleurs, *Librairie Larousse*, 1977.

[Lienz 80]
B. Lientz & E. Swanson. "Software Maintenance Management", *Addison-Wesley*, 1980.

[Liskov 74]
B. Liskov & S.N Zilles. "Programming with Abstract Data Types", *SIGPLAN Notices,* 9, 4, Apr. 1974.

[Liskov 77]
B. Liskov, A. Snyder, R. Atkinson & C. Schaffert. "Abstraction Mechanisms in CLU", Communications of the ACM, 20, aug. 1977.

[Liskov 81]
B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, R. Scheiffler & A. Snyder. "CLU Reference Manual", *Springer-Verlag*, 1981.

[Loomis 87]
M.E.S. Loomis, A.V. Shah & J.E. Rumbaugh. "An Object Modeling Technique for Conceptual Design", *European Conference on Object-Oriented Programming, ECOOP'87 Proceedings*, pp 192-202, Paris, jun. 1987.

[Malverti 88]
C. Malverti. "Connaissances et Stratégies de Diagnostic dans une Représentation Centrée-Objet", *Rapport de DEA* , Grenoble, juin 1988.

[Masini 89]
G. Masini, A. Napoli, D. Colnet, D. Léonard, K. Tombre. "Les Langages à Objets", *InterEdition* , Paris 1989.

[McCall 77]
J. McCall, P. Richards & G. Walters,. "Factors in Software Quality", 3 vols., NTIS AD-A049-014, 015, 055, Nov. 1977.

[Mevel 87]
A. Mevel & T. Gueguen. "SMALLTALK-80", *Ed. Eyrolles*, 1987.

[Meyer 78]
B. Meyer & C. Baudoin. "Méthodes de Programmation", *Eyrolles, Collection de la Direction des Etudes et Recherche d'EDF*, 1978.

[Meyer 87a]
B. Meyer. "Software Reusability: The Case for Object-Oriented Design", *IEEE Software*, pp. 50-64, Mar. 1987.

[Meyer 87b]
B. Meyer. "Eiffel: A Language and Environment for Software Engineering", *Report TR-EI-2/BR*, Interactive Software Engineering, Inc.,1987.

[Meyer 88]
B. Meyer. "Object-Oriented Software Construction",*Series in Computer Science, Prentice Hall International*, 1988, et "Conception et Programmation par Objets", *traduction française, InterEditions*, 1990.

[Meyer 89a]
B. Meyer. "You Can Write, But Can You Type?", *Journal of Object-Oriented Programming*, Mar/Apr. 1989.

[Meyer 89b]
B. Meyer. "The New Culture of Software Development: Reflections on the Practice of Object-Oriented Design", *Technology of Object-Oriented Languages and Systems*, first international conference, Paris, Nov. 1989.

[Meyer 90a]
B. Meyer. "Sequential and Concurrent Object-Oriented Programming". *Technology of Object-Oriented Languages and Systems*, second international conference, Paris, Jun. 1990.

[Meyer 90b]
B. Meyer. "Lessons from the Design of the Eiffel Libraries", *Communications of the ACM*, vol.33, N°9, Sep. 1990.

[Michel 88]
C. Michel, R. Rousseau & M. Rueher. "Expérimentation d'Eiffel: un premier bilan", *4ème colloque AFCET de génie logiciel,* Paris, 1988.

[Miller 63]
G.A. Miller. "The magical number seven, plus or minus two save limits on our capacity for processing information", *Psychologocal Review* , Mar. 1963.

[Moon 86]
D. Moon. "Object Oriented Programming with FLAVORS", *Proceedings of the 1st OOPSLA,* Portland, Oregon, 1986.

[Morat 87]
P. Morat. "Une Expérience de Programmation avec le Langage Eiffel", *L.G.I. Contribution to ASPIS Project,* Contrat ESPRIT 401, 1987.

[Mühlhäuser 88]
M. Mühlhäuser, A. Schill & L. Heuser. "Project DOCASE: Software Engineering for Distributed Applications: an Object-Oriented Approach", *International workshop proceedings: "software engineering and its applications"*, vol 1, Toulouse, dec. 1988.

[Myers 78]
G.J. Myers. "Composite / Structured Design", *Van Nostrand Reinhold,* New York, 1978.

[Nerson 87]
J. M. Nerson. "Extending EIFFEL Toward O-O Analysis and Design" et "Case Studies In Object-Oriented Analysis and Design" , à paraître en 1991.

[Nierstrasz 91]
O.M. Nierstrasz. "Hybrid - A Language for Programming with Active Object", *D. Tsichritzis editor, Objects and Things,* Université de Genève, 1987.

[Papathomas 89]
M. Papathomas "Concurrency Issues in Object-Oriented Programming Langages", *??.*

[Papert 81]
S. Papert. "Jaillissement de l'Esprit: Ordinateur et Apprentissage", *Flammarion,* 1981.

[Parnas 72]
Parnas. "On Criterion to be Use to Decompose Systems into Modules", *ACM vol. 15*, pp 1053-1058, dec. 1972.

[Pressman 88]
R.S. Pressman. "Software Engineering: A Practitioner's Approach", *McGraw-Hill International*, 1987.

[Riguidel 87]
E. Riguidel. "La Méthode MAC_ADAM", *TH-CSF/SDC Direction Technique*, oct. 1987.

[Robert 91]
S. Robert De St Victor. "Interface Graphique pour le Poste de Conception MECANO", *Rapport de Thèse CNAM*, soutenance juin 1991.

[Ross 85]
D.T. Ross. "Application and Extensions of SADT", *IEEE Trans. Software Engineering*, Avr. 1985.

[Rumbaugh 91]
J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy & W. Lorensen "Object Oriented Modeling and Design", *Prentice Hall, Englewood Cliffs*, 1991.

[Sakkinen 89]
M. Sakkinen "Disciplined Inheritance", *??* .

[Saunders 89]
J.H. Saunders. "A survey of Object-Oriented Programming Languages", *Journal of Object-Oriented Programming*, mar/apr. 1989.

[Schaffert 86]
C. Schaeffert, T. Cooper, B. Bullis, M. Kilian & C. Wilpolt. "An Introduction to Trellis-Owl", *OOPSLA 1986*, pp 9-16, 1986.

[Shlaer 88]
S. Shlaer, S.J. Mellor, D.Ohlsen & W. Hywari. "The Object-Oriented Method for Analysis", *Proceedings of the Tenth Structured Development Forum*, 1988.

[Schmucker 86]
K. Schmucker. "Object Oriented Programming on the Macintosh", *Hayden Books, Hasbrouck Heights*, New Jersey,1986.

[Scholl 83]
P.C. Scholl, M. Lucas & J.P. Peyrin. "Algorithmique et Représentation des Données", *tome1,2 et 3, Masson*, 1983.

[Snyder 86]
A. Snyder. "Encapsulation and Inheritance in Object-Oriented Programming", *Object-Oriented Programming Systems, Languages and Applications Conference*, pp. 38-45, 1986.

[Stay 76]
J.F. Stay. "HIPO and Integrated Program Design", *IBM System Journal*, 15, n° 2, pp 143-154, 1976.

[Stevens 74]
W.P. Stevens, G.J. Myers & L.L. Constantine. "Structured Design", *IBM System Journal*, 13, n° 2, pp 115-139, 1974.

[Stroustrup 86]
B. Stroustrup,. "The C++ Programming Langage", *Addison-Wesley Series in Computer Science*, Reading, Massachusetts,1986.

[Stroustrup 87]
B. Stroustrup. "What is Object-Oriented Programming?", *European Conference on Object-Oriented Programming*, pp 51-70, 1987 (voir aussi *IEEE Software*, May 88)

[Tardieu 82]
Tardieu, Rochfeld & Coletti. "La Méthode Merise", 1982.

[Tesler 85]
L. Tesler. "Object Pascal Report", *Structured Language World*, Springer Verlag, New York, 1985.

[Warnier 81]
J.D. Warnier. "Logical Construction of Systems", *Van Nostrand Reinhold*, 1981.

[Wegner 87]
P. Wegner. "Dimension of Object-Based Language Design", *Proceedings of the 2nd OOPSLA*, Orlando, Florida, 1987.

[Wirfs-Brock 89]
R. Wirfs-Brock, B. Wilkerson. "Object-Oriented Design, A Responsability-Driven Approach", *OOPSLA '89 Proceeding* , Oct. 1989.

[Wirfs-Brock 90a]
R. Wirfs-Brock & R.E. Johnson. "Surveying Current Research in Object-Oriented Design", *Communications of the ACM* , vol.33, N°9, Sep. 1990.

[Wirfs-Brock 90b]
R Wirfs-Brock, B. Wilkerson & L. Wiener. "Designing Object-Oriented Software", *Prentice Hall, Englewood Cliffs* ,1990.

[Wirth 71]
N. Wirth. "Program Development by Stepwise Refinement", *Communications of the ACM,*, vol.14, N°4, 1971.

[Wirth 82]
N. Wirth. "Programming in Modula-2", *Springer-Verlag*, New York, 1982.

[Wolf 89]
W. Wolf. "A Practical Comparison of Two Object-Oriented Languages", *IEEE Software,* Septembre 1989.

[Xerox 86]
Xerox . "Programming in LOOPS", *Composite Object - lecture 7*, Xerox Corporation, 1986.

[Yourdon 79]
E. Yourdon & L.L. Constantine. "Structured Design", *Englewood Cliffs, Prentice-Hall*, 1979.

AUTORISATION DE SOUTENANCE

T 3 01160

DOCTORAT DE L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

Vu les dispositions de l'Arrêté du 5 juillet 1984,

Vu les rapports de Monsieur Jean BEZIVIN

Monsieur Pierre COINTE

Monsieur Xavier GIRAD est autorisé(e)
à présenter une thèse en vue de l'obtention du Doctorat de l'Université Joseph Fourier - Grenoble 1

Grenoble, le 5 Juin 1

Le Président de l'Université
Joseph Fourier - Grenoble 1

A. NEMOZ

Résumé:
Nous présentons la méthode MECANO de conception d'applications logicielles par objets. Partant des concepts du modèle de la programmation par objets, et du constat que l'approche objet peut améliorer l'évolutivité et la réutilisabilité des logiciels, MECANO propose des concepts novateurs adaptés à la phase de conception. La taxonomie de l'héritage apporte une sémantique à la relation liant une classe à sa super-classe, permettant d'améliorer la lisibilité des graphes d'héritage produits. Cette taxonomie sert de base à la notion de domaine, entité structurante offrant des composants réutilisables, basée sur une hiérarchie de spécialisation/ réalisation. Deux autres concepts structurant sont définis, les composites et les applications, permettant d'organiser l'ensemble des classes produites sur des axes complémentaires aux domaines. La donnée d'un formalisme graphique et d'un processus de conception complètent la définition de la méthode. A travers un exemple, nous illustrons notre méthode et montrons qu'elle permet d'obtenir des conceptions plus flexibles et plus réutilisables.

Mots clés:
paradigme objet, génie logiciel, qualité du logiciel, méthode de conception par objets, processus de conception, atelier logiciel, formalisme graphique

Abstract:
We present a method for Object Oriented Software Design called MECANO. MECANO is based on the object-oriented programming model and provides new concepts for the design phasis in order to improve evolutivity and reusability. The inheritance taxonomy provides semantic to the class/super-class relationship and improves the understandability of the inheritance graph. The structuring entity called Domain is based on this taxonony. Domains are specialization/realization hierarchies containing reusable components. Two other structuring entities are also introduced: Composites and Applications which provide complementary ways for class organisation. A graphic formalism and a design process are given in order to define other aspects of the MECANO design method. We finally present an exemple, showing how the method leads to more flexible and reusable designs.

Keywords:
object-oriented paradigm, software engineering, software quality, object-oriented design method, design process, computer assisted software engineering, graphical formalism