Contribution à la conception logicielle de systèmes d'applications : la méthode MOSAIC dans le projet Aristote

Frédéric Brissaud

THÈSE

présentée par

Frédéric BRISSAUD

pour obtenir le titre de

Docteur de l'Université Joseph Fourier - Grenoble 1

(Arrêtés ministériels du 5 Juillet 1984 et du 23 Novembre 1988)

Spécialité : **INFORMATIQUE**

Contribution à la conception logicielle de systèmes d'applications

La méthode MOSAÏC dans le projet Aristote

Thèse soutenue le 13 Mai 1993

Composition du jury :

Président :	Yves CHIARAMELLA
Rapporteurs :	Colette ROLLAND
	Jean BÉZIVIN
Examinateurs :	Joëlle COUTAZ
	Mauricio LOPEZ
Directeur :	Jean-Pierre GIRAUDIN

Thèse préparée au sein du Laboratoire de Génie Informatique de Grenoble

Ce travail n'aurait pu aboutir sans l'aide et la participation de nombreuses personnes à qui je tiens à témoigner ma reconnaissance :

Yves Chiaramella, Professeur à l'Université Joseph Fourier et Directeur du Laboratoire de Génie Informatique qui me fait l'honneur de présider le jury,

Colette Rolland, Professeur à l'Université Panthéon Sorbonne, pour l'attention et le soutien chaleureux qu'elle porte à ce travail depuis son début et pour avoir bien voulu en être rapporteur,

Jean Bézivin, Professeur à l'Université de Nantes, qui a accepté d'être un de mes rapporteurs et qui a, par ses commentaires constructifs, manifesté un vif intérêt pour ce travail,

Joëlle Coutaz, Professeur à l'Université Joseph Fourier, pour avoir accepté de participer au jury de cette thèse et dont l'expérience et les recherches en IHM ont largement contribué au contenu de ce travail,

Jean-Pierre Giraudin, Professeur à l'Université Pierre Mendès France et directeur de cette thèse, qui m'a toujours accordé une confiance totale et une aide précieuse,

Mauricio Lopez, Ingénieur et responsable de l'équipe bases de données du Centre de Recherche Bull de Grenoble, pour avoir accepté de participer au jury de cette thèse,

La région Rhône-Alpes qui a financé ce travail mené conjointement au Laboratoire de Génie Informatique et au Centre de Recherche BULL,

Les responsables (Michel Adiba, Marie-France Bruandet et Mauricio Lopez) et les membres du projet Aristote pour leur soutien permanent pendant cette recherche,

Les membres de l'ORC Conception Orientée Objet (PRC-BD3) initialisée et dirigée par Colette Rolland avec qui nous avons pu périodiquement confronter fructueusement nos idées,

Les équipes Génie Logiciel et Interface Homme-Machine du LGI qui m'ont permis d'élargir ma vision des méthodes de conception,

Nicolas Pudda qui a contribué à donner corps à mes réflexions et dont la bonne humeur constante a été un soutien précieux,

Enfin, tous ceux qui de près ou de loin m'ont supporté, soutenu ou encouragé tout au long de ces années : Laurence, Christophe, Jean-Pierre, Hervé, Philippe, Christine Luis, Jean-Louis, Monique...

Sommaire

Chapitre 1 Introduction 1

Chapitre 2 Programmation, Bases de données et Systèmes interactifs 11

Chapitre 3 Techniques à objets et conception dirigée par les objets 55

Chapitre 5 Classes applicatives, classes interactives et classes projets 115

Annexe A Types abstraits de données, sous-typage et héritage A-1

Annexe B Langages et modèle MOSAÏC B-1

Annexe C Application «Gestion de Compétitions Sportives» C-1

Liste des figures

Chapitre 1 Introduction 1

Chapitre 2 Programmation, Bases de données et Systèmes interactifs 11

Chapitre 3 Techniques à objets et conception dirigée par les objets 55

Chapitre 4 MOSAÏC : Modèle d'architecture 93

Chapitre 5 Classes applicatives, classes interactives et classes projets 115

Chapitre 6 Outils de conception et éléments d'une démarche 159

Chapitre 1

Introduction

Face à l'augmentation constante de la complexité des applications informatiques, les méthodes de conception sont aujourd'hui reconnues comme une aide essentielle pour le développement de logiciels de qualité. Une méthode représente un savoir-faire de plus en plus indispensable aux concepteurs comme aux programmeurs. Ce savoir-faire se présente généralement sous la forme de quatre composants [RFB88] :

- Les modèles fixent les concepts et leurs règles d'utilisation pour décrire l'application en cours d'étude et de développement.
- Les langages permettent l'utilisation pratique des modèles en proposant un vocabulaire de base et une syntaxe.
- La démarche organise le processus de développement d'une application en différentes étapes et fournit un guide chronologique et décisionnel au concepteur.
- Les outils s'appuient sur les modèles, les langages et la démarche pour faciliter le travail du concepteur. Ce sont principalement des outils de représentation, de stockage, de documentation, d'évaluation, de simulation, de transformation, d'aide à la production de code, etc.

A la fin des années 60, le coût relatif des logiciels a commencé à dépasser celui du matériel marquant le début de ce que l'on a appelé la crise du logiciel. Depuis, des critères de qualité et des méthodes de conception ont été élaborés pour tenter d'apporter une solution à ce problème qui reste toujours d'actualité. Au cours des années 60, les premières méthodes de conception de systèmes d'information ont fait leur apparition. Ces méthodes ont ensuite évolué au fil des progrès technologiques et des différentes modélisations de la notion de système d'information. Plus récemment, au début des années 80, le développement de la micro-informatique et des interfaces graphiques a bouleversé les habitudes des utilisateurs. La complexité de la mise en œuvre de dialogues conviviaux a conduit au développement de techniques de conception spécifiques aux applications interactives et a été une des origines des approches par les objets.

Le degré de spécialisation d'une méthode de conception conditionne la qualité de l'aide qu'elle apporte au concepteur. Pour offrir une aide efficace, il est nécessaire de limiter le champ d'application d'une méthode sans pour autant mésestimer l'intérêt d'une méthode générale. La section suivante présente le cadre de notre travail qui a

pour objectif d'étudier une technique de conception qui soit adaptée au développement d'un ensemble d'applications interactives.

1 Champ de notre étude

1.1 Systèmes d'applications

Depuis la généralisation de l'outil informatique, différents facteurs ont contribué au morcellement des systèmes informatiques : la montée en puissance des micro-ordinateurs, l'extension des réseaux, la décentralisation des organisations, etc. L'informatique traditionnelle, centralisée et monolithique, laisse progressivement place à des systèmes constitués d'unités autonomes plus proches des utilisateurs et qui communiquent de façon intensive. Parallèlement, le logiciel a évolué en mettant à profit l'augmentation de la puissance des ordinateurs, des capacités de stockage des mémoires secondaires et la prise de conscience de la nécessité d'offrir des interfaces conviviales. Parmi les différentes applications informatiques existantes, nous nous intéressons à une catégorie d'applications que nous appelons les systèmes d'applications. Un **système d'applications** possède les caractéristiques suivantes :

- Il est composé de différents programmes que nous appelons **applications** et qui peuvent être déclenchés simultanément par les utilisateurs. La durée de vie d'une application est limitée dans le temps par un début et une fin.
- Il gère des informations persistantes, complexes, volumineuses, variées et fortement interconnectées. Les informations persistantes peuvent être manipulées par les différentes applications du système d'applications.
- Une application peut réaliser des fonctions compliquées qui ne se résument pas à une combinaison simple d'opérations de base sur les informations persistantes (ajout, suppression ou modification).
- Chaque application échange avec l'utilisateur suivant un protocole riche dans lequel l'initiative du dialogue doit être laissée à l'utilisateur.
- Les applications ne sont pas soumises à des contraintes particulières de délais de réponse à l'inverse des applications dites «temps réel». Les temps de réponse doivent cependant être conformes à l'attente de l'utilisateur en interaction avec les applications.

Parmi les applications répondant à ces critères, on trouve par exemple : des systèmes d'information de gestion, les environnements de développement de logiciels, les applications CAO, etc.

1.2 Logiciels de base cibles

Nous appelons **logiciels de base** l'ensemble des logiciels utilisés pour program-

mer un système d'applications. Les langages de programmation, les systèmes de gestion de fichiers ou de bases de données et les boîtes à outils graphiques sont des exemples de logiciels de base. Avec l'avènement des techniques à objets, la traditionnelle séparation entre les données persistantes et les programmes tend à disparaître au profit de logiciels de base «intégrés» offrant à la fois un langage de programmation et des mécanismes de gestion de la persistance. De tels systèmes font encore l'objet de recherches, aussi nous nous fixons les caractéristiques minimales des logiciels de base que nous avons pris en compte et que nous appelons Langages de Programmation de Bases d'Objets. Un **Langage de Programmation de Bases d'Objets** (LPBO) intègre les concepts de base des techniques orientées objets [Wegn87] et offre les caractéristiques suivantes :

- Il est bâti sur la notion d'objet encapsulant des données et des traitements et accessible seulement par une interface limitée.
- Il intègre un mécanisme de classification par lequel une classe décrit un modèle d'objets et tel que tous les objets sont obtenus par instanciation d'une classe.
- Il supporte la notion d'héritage pour factoriser des caractéristiques communes à plusieurs classes et permet des relations de sous-typage.
- Il offre un langage de description des données persistantes et un langage de programmation pour manipuler ces données persistantes tous deux fondés sur les notions d'objet, de classe et d'héritage.
- Il doit permettre la cohabitation d'objets persistants et d'objets temporaires. La durée de vie d'un objet temporaire est limitée à celle de l'application dans laquelle il a été créé alors qu'un objet persistant doit être détruit explicitement.
- Il offre des classes spécialisées pour gérer le dialogue établi entre le programme et l'utilisateur.

Parmi les logiciels de base répondant à ces critères, on trouve en général les SGBD orientés objets et en particulier le système O_2 [ODeu89], mais également les langages de programmation à objets offrant des mécanismes élémentaires de gestion de la persistance comme Eiffel [Meye88] ou Guide [KMR*88]. Ces systèmes appartiennent à la catégorie des langages de classes et ils sont à distinguer des langages de frames ou des langages d'acteurs [MNC*89].

1.3 Cycle de développement

Le développement d'un système informatique est un processus long et complexe. Son bon déroulement conditionne l'efficacité, l'adéquation aux besoins exprimés et même la disponibilité du futur système. Le processus de développement d'un système informatique est traditionnellement décomposé en quatre phases principales :

- La spécification fixe les services que doit remplir le système d'applications indépendamment d'une technique de réalisation de ces fonctions.

- La conception conduit, par le choix d'un ensemble de techniques informatiques, à élaborer une architecture de composants logiciels permettant de réaliser les spécifications du système d'applications (choix des algorithmes, de la représentation des données, etc.).
- La réalisation, dans laquelle les composants retenus pendant le conception sont programmés à l'aide d'un ou de plusieurs logiciels de base pour obtenir le système d'applications opérationnel (exécutable).
- La maintenance qui assure, après la mise en place du système dans l'organisation, son adaptation aux évolutions des besoins des utilisateurs et de l'organisation.

On notera que cette décomposition en quatre phases ne préjuge en rien de l'organisation temporelle précise du cycle de vie. L'enchaînement des phases n'est pas fixé et peut être de différentes natures : cascade, V, spirale, etc.

Une méthode complète doit assister le concepteur d'un système informatique tout au long du cycle de vie de ce système et notre objectif à long terme est de proposer une telle méthode. Cependant, dans le cadre de ce travail, nous nous sommes limités à l'étude de la phase de **conception**.

Notre objectif est donc de proposer une approche pour faciliter la conception d'un système d'applications programmé à l'aide d'un langage de programmation de bases d'objets. Une telle approche doit notamment apporter une réponse aux deux questions complémentaires suivantes :

Comment organiser le code d'un système d'applications programmé à l'aide d'un Langage de Programmation de Bases d'Objets ?

Quels concepts peut-on proposer au concepteur pour faciliter le déroulement de l'étape de conception ?

1.4 Méthodes de conception

Des études ont montré qu'il n'existe pas actuellement de méthodes de conception bien adaptées à notre contexte [Gira90]. On peut relever deux raisons principales à ce constat :

- Les LPBO combinent les fonctionnalités des systèmes de gestion de bases de données et des langages de programmation. Ils apportent un modèle de données riche, une puissance de structuration voisine de celle des langages à objets et

permettent souvent la réalisation d'interfaces conviviales. Ces systèmes, par l'encapsulation de données et de traitements et par la cohabitation d'objets persistants et d'objets temporaires, font disparaître la séparation entre les données persistantes et les programmes en vigueur dans les systèmes conventionnels et sur laquelle s'appuient toutes les méthodes classiques de conception.

- Les différentes méthodes de conception existantes sont très spécifiques et trop partielles pour prendre en compte de façon homogène et intégrée les trois domaines concernés par un système d'applications : les bases de données, les programmes et les interfaces homme-machine. Les méthodes d'analyse et de conception de systèmes d'information (MACSI) sont les seules à aborder un système dans sa globalité. Cependant, l'émergence des logiciels de base permettant la réalisation d'interfaces conviviales et des développements par objets est postérieure à la définition de la majorité des MACSI qui n'ont pas pu prendre en compte cette dimension importante des systèmes d'applications ni les services offerts par les LPBO.

Il est donc nécessaire de définir de nouvelles méthodes de conception mieux adaptées à l'évolution des techniques.

2 Notre approche

2.1 Projet Aristote

Notre travail prend place dans le cadre du projet de recherche Aristote mené conjointement par le Laboratoire de Génie Informatique de Grenoble et le centre de recherche BULL de Grenoble. L'objectif de ce projet co-dirigé par M. Adiba, M-F. Bruandet et M. Lopez est d'offrir un ensemble d'outils intégrés pour la conception d'applications dans un environnement hétérogène constitué de systèmes de gestion de bases de données orientés objets (type O_2), relationnels étendus (type ESQL) ou relationnels (type Oracle). Quatre grands objectifs ont été fixés :

- intégrer les données et les traitements dans un formalisme uniforme pour réduire la dichotomie entre un système de gestion de données et un langage de programmation ;
- fournir un langage de définition d'applications de haut niveau, avec le plus de déclarativité possible ;
- offrir des capacités de structuration des applications, de façon à pouvoir augmenter la réutilisation de tout ou partie des applications ;
- pouvoir supporter des applications hétérogènes et notamment pouvoir faire coopérer des fragments d'applications écrits dans des langages différents.

La solution technique expérimentale retenue est d'implanter le langage comme un générateur d'applications. Dans notre contexte, un générateur peut être vu comme un compilateur générant du code pour des logiciels de base particuliers. Le projet Aristote est articulé autour de trois activités principales :

- une activité «générateur» a pour objectif de définir un modèle et un langage et de réaliser un générateur multicible ;
- une activité «poste de travail» propose et réalise un ensemble homogène d'outils d'aide au développement d'applications Aristote ;
- une activité «applications» valide les différentes propositions sur des applications concrètes prises dans les domaines du génie logiciel, et des systèmes médicaux.

Notre travail doit contribuer d'une part à l'élaboration du poste de travail d'Aristote et d'autre part à enrichir le modèle et le langage Aristote pour une meilleure structuration des applications dans la phase de conception.

2.2 Démarche adoptée

Pour mener à bien nos travaux, nous nous sommes appuyés sur trois constats complémentaires :

- Le développement des systèmes d'applications auxquels nous nous intéressons est une activité qui fait intervenir trois domaines de compétences à prendre en compte. Tout d'abord le domaine des bases de données pour la gestion des données persistantes, ensuite la programmation pour l'élaboration des fonctions des applications et enfin le domaine des interfaces homme-machine pour la mise en œuvre des dialogues entre les applications et les utilisateurs.
- Des méthodes et des techniques de conception spécifiques ont été proposées et expérimentées dans ces trois domaines. Cependant elles ont généralement été développées en réponse à des préoccupations spécifiques et sans souci particulier d'intégration. Ce manque d'homogénéité entraîne notamment une surcharge de travail pour le concepteur qui doit combiner des techniques de conception récentes dont la synthèse n'existe pas actuellement.
- Les techniques à objets issues des langages de programmation sont actuellement reprises et adaptées dans la plupart des domaines de l'informatique. En particulier, après les langages de programmation orientés objets, des logiciels de base spécialisés dans la gestion de l'interaction ainsi que des modèles d'architecture ont intégré ces techniques. De même, les premiers SGBD orientés objets commencent à être commercialisés [DLR91].

Nous utilisons cette notion d'objet comme vecteur d'intégration des techniques récentes en base de données, en programmation et en interface homme-machine pour donner les bases d'une méthode de conception d'applications adaptée aux LPBO. Nous voulons que ce travail ne soit pas spécifique dans l'un des trois domaines mentionnés

mais une contribution au domaine des méthodes de conception.

2.3 Difficultés rencontrées

Pour développer notre méthode nous avons tenté d'apporter une réponse aux questions suivantes :

Doit-on remettre en cause la décomposition base de données / programme / interface pendant l'étape de conception ?

Sur quel modèle d'architecture doit-on s'appuyer pour concevoir un système d'applications ?

Qu'est-ce qu'un objet : une donnée passive, un module logiciel, un agent interactif ou une entité dont le rôle reste à définir ?

Qu'est-ce qu'un ensemble d'objets : une base de données, un programme, un gestionnaire de dialogue ou une entité multiforme ?

Apporter une réponse à ces questions essentielles contribue à répondre à la question d'actualité :

Comment trouver les «bons» objets ?

L'étude de ces questions a mis en évidence une difficulté importante. La synthèse que nous avons tentée de réaliser sur les trois domaines (programmation, bases de données et interfaces homme-machine) a très vite posé un problème de vocabulaire. Par ailleurs, l'intérêt du consensus réalisé autour de la notion d'objet est pondéré par les différentes interprétations qui sont faites de cette notion. En effet, sous une uniformité de vocabulaire, les techniques à objets sont interprétées de façons différentes selon qu'elles sont utilisées dans le domaine de la programmation, des bases de données ou des interfaces homme-machine. Nous avons dû fixer notre propre vocabulaire pour intégrer ces différents points de vue.

2.4 Notre méthode de conception : MOSAÏC

MOSAÏC (Méthode de cOnception de Systèmes d'ApplICations) propose, pour concevoir l'organisation globale d'un système d'applications, un **modèle d'architecture général** qui fixe les principaux composants et les relations entre ces composants. Pour concevoir pratiquement le système d'applications, le concepteur dispose d'un **modèle** et de deux **langages** (un langage graphique et un langage textuel) cohérents avec le modèle d'architecture général retenu. Un **outil** graphique d'aide à la conception a été développé à l'aide du générateur d'ateliers de génie logiciel GraphTalk [RX91]. Nous avons retenu quelques éléments pour guider la conception sans toutefois proposer une démarche complète.

Nous avons tenté de valider en partie ce travail en développant un système d'applications concret concernant la «Gestion de Compétitions Sportives» (cf. spécifications en annexe). Certaines parties de ce système d'applications ont été programmées et testées avec le SGBD orienté objets O2.

3 Plan de la thèse

Dans le chapitre 2, nous proposons notre perception des principales évolutions en programmation, en bases de données et dans les interfaces homme-machine. Cette étude nous a semblé nécessaire pour pouvoir mettre en évidence différentes interprétations possibles des techniques à objets. Nous présentons pour chaque domaine, les objectifs généraux, les grandes familles de logiciels de base et les principales techniques ou méthodes de conception proposées. Ce chapitre est également l'occasion de fixer le vocabulaire utilisé dans la suite du document.

Le chapitre 3 introduit les principaux concepts des techniques à objets. Chaque fois que c'est possible, nous mettons en évidence différentes interprétations de ces concepts et leurs conséquences en nous appuyant notamment sur les éléments présentés dans le chapitre 2. Nous présentons également dans ce chapitre, les principes généraux de la conception dirigée par les objets puis nous proposons un état de l'art des principales méthodes de conception dites «orientées objets» disponibles actuellement.

Le modèle d'architecture MOSAÏC est présenté dans le chapitre 4. Il identifie les principaux composants d'un système d'applications, il fixe leurs rôles et leur composition et il organise les relations entre ces composants. Ce modèle d'architecture concrétise en particulier les principes essentiels que nous proposons pour concevoir un système d'applications. Nous présentons également, de façon intuitive, des concepts originaux de la méthode.

Nous présentons dans le chapitre 5 les concepts de base de notre modèle ainsi que

les éléments essentiels des langages associés. Nous définissons trois catégories de classes : les classes applicatives, les classes interactives et les classes projets. Les classes applicatives et les classes interactives sont définies selon trois dimensions : structurelle, fonctionnelle et dynamique. Le concept de classe projet est détaillé en précisant les notions de schéma de base d'objets, de corps d'une application et de gestionnaire de dialogue d'une application.

L'outil OCAPI d'aide à la conception est présenté dans le chapitre 6. Cet outil est actuellement un éditeur graphique et textuel qui gère une partie des concepts présentés dans les chapitres précédents. Nous exposons les règles de traduction d'un schéma de conception MOSAÏC vers un programme écrit en PEPLOM, le langage du projet Aristote, et vers un programme O_2C, le langage du SGBD orienté objets O_2. Dans ce chapitre, nous présentons également quelques éléments d'une démarche de conception adaptée à MOSAÏC.

Un dernier chapitre 7 propose une évaluation de notre travail et des perspectives d'évolution de notre méthode.

Chapitre 2

Programmation, Bases de données et Systèmes interactifs

1 Introduction

Les systèmes «orientés objets» seront certainement les systèmes de la décennie 90. Les techniques à objets issues des langages de programmation orientés objets se généralisent à la plupart des domaines de l'informatique : les bases de données, les systèmes interactifs, l'intelligence artificielle, les systèmes d'exploitation et même l'architecture des processeurs [BGHS91]. Ainsi, après les langages OO, nous avons vu apparaître : les systèmes de gestion de bases de données OO, les outils OO de construction de systèmes interactifs, les systèmes d'exploitation OO, les méthodes de conception OO, les méthodes d'analyse OO, etc. Tous ces systèmes offrent généralement les mêmes notions de base : objet, classe et héritage. Cependant, sous cette apparente uniformité de vocabulaire, se cachent des interprétations et des utilisations qui diffèrent souvent en fonction de l'approche adoptée ou en fonction du domaine considéré.

Pour proposer une méthode qui intègre, de façon homogène, la conception des fonctions des programmes, la définition de la base de données et l'élaboration du gestionnaire de dialogue, nous devons mettre en évidence et prendre en compte ces différentes interprétations. Dans cette perspective, ce second chapitre est consacré au rappel du contexte et des principales évolutions technologiques et méthodologiques des domaines concernés par notre approche : la programmation, les bases de données et les systèmes interactifs. Cependant, n'étant pas spécialiste de chacun de ces domaines, notre étude ne prétend pas être exhaustive, ni suivre strictement la chronologie des évolutions. Ce chapitre présente donc notre point de vue particulier sur ces trois domaines de l'informatique et il nous permet de fixer les concepts et le vocabulaire utilisé dans la suite de ce document.

Ce bref tour d'horizon nous permettra, au chapitre suivant (chapitre 3), d'aborder plus en détail les techniques à objets et de faire un état de l'art des méthodes de conception orientées objets.

2 Programmation

Considéré au début de l'ère informatique comme une activité annexe, le développement de logiciels a montré très vite une grande complexité. La crise du logiciel, reconnue à la conférence de Garmisch-Partenkirchen en 1968, est marquée par le dépassement du coût matériel par le coût logiciel et la croissance exponentielle de ce dernier. Cette conférence a situé la naissance du domaine du génie logiciel qui recouvre l'ensemble des techniques intervenant dans l'activité de développement de logiciels : les techniques de programmation et de conception, les langages et les outils associés, les techniques de gestion de projets, d'évaluation et de planification des ressources et des moyens.

2.1 Généralités

2.1.1 Différentes approches de la notion de programme

J. Ferber [Ferb90] identifie quatre approches de la programmation :

- Dans la ***programmation impérative*** ou procédurale classique, un programme est une suite d'instructions manipulant un ensemble de données.
- La ***programmation fonctionnelle*** modélise un programme comme une fonction mathématique définie par son domaine et son co-domaine.
- En ***programmation logique***, un programme est un raisonnement et son exécution revient à prouver la déductibilité d'une proposition logique.
- Selon la ***programmation orientée objets***, un programme est un ensemble d'objets qui interagissent et communiquent par des messages.

Nous nous limitons à la programmation classique et à la programmation orientée objets plus particulièrement adaptées au développement de grands logiciels complexes.

2.1.2 Abstraction

Le Larousse définit l'abstraction comme : *"l'opération intellectuelle qui permet d'isoler une caractéristique, une propriété d'un objet et de la considérer indépendamment des autres."* Par extension on appelle abstraction d'un objet l'ensemble des propriétés qui peuvent être utilisées à la place de l'objet lui-même. On peut identifier trois techniques indépendantes et générales d'abstraction [GMB82] : l'***agrégation / décomposition*** (X est composé de Y1, Y2, Y3), la ***classification / instanciation*** (Xi est un exemplaire de X) et la ***généralisation / spécialisation*** (X est un cas particulier de Y). La notion d'abstraction nécessite de distinguer clairement une vue limitée d'un élément, appelée généralement **interface** et les détails techniques de réalisation. On retrouve ces trois techniques générales dans les mécanismes d'abstraction habituellement utilisés en programmation [Cohe84, THO85] :

- Le **type concret** décrit un type par une structure de données particulière.

- L'**abstraction procédurale** permet de définir une action par une suite d'instructions.
- L'**abstraction de données** introduit un composant défini par un ensemble de services réalisés à l'aide de données et de suites d'instructions.
- La **classification** ou abstraction de valeur est utilisée pour décrire une catégorie d'objets de structures similaires par un modèle unique.
- Un **type abstrait de données** définit un domaine de valeurs par les opérations utilisables sur les éléments de ce domaine.
- Un **composant générique** définit une catégorie de composants à l'aide d'un composant unique dont le type de certaines données est variable.
- Le **polymorphisme** correspond à des opérations ayant le même nom mais dont les types des paramètres sont différents.

2.1.3 Critères de qualité

Le développement de logiciels de qualité est un objectif unanimement reconnu pour tenter d'apporter une solution à la crise du logiciel notamment en permettant d'élaborer plus rapidement des programmes plus fiables, plus maintenables, plus réutilisables, etc. Les principaux critères de qualité applicables aux programmes sont les suivants [Meye88] :

- La ***modularité*** caractérise la décomposition d'un programme en composants logiciels élémentaires et relativement indépendants. C'est un critère indispensable pour construire des logiciels de grande taille. L'**unité** de décomposition décrit la forme des composants résultant de la décomposition. Le **critère** de décomposition définit les qualités ou les propriétés considérées pour effectuer la décomposition.
- La ***généralité*** est la plage d'utilisation des composants. Ce critère conditionne la possibilité de réutilisation des composants. Plus un composant est général et plus les situations dans lesquelles il peut être utilisé, moyennant éventuellement des adaptations, sont nombreuses.
- L'***auto-documentation*** est la possibilité d'extraire automatiquement de la documentation des composants logiciels. Cette documentation peut provenir du texte même du programme (mots clé du langage) ou des commentaires insérés par le programmeur.
- La ***complétude*** est le respect des spécifications du logiciel. La réalisation de ce critère est primordiale pour la satisfaction des utilisateurs du programme.

Nous présentons plus en détail les critères de modularité et de généralité particulièrement utiles pour le concepteur et pour le programmeur.

2.1.4 Modularité

Le concept de modularité, apparu au début des années 70 [Parn72], permet de réduire la complexité d'un programme en le décomposant. Une méthode ou un langage répond au critère de modularité si il permet le découpage du logiciel en éléments (composants logiciels) de granularité supérieure à celle des instructions du langage sous-jacent. Le critère de modularité peut être raffiné en cinq sous-critères [Meye88] :

- La ***décomposition modulaire*** permet de décomposer un composant en sous-composants en recherchant un couplage minimum et une forte cohésion dont Parnas a montré l'intérêt [Parn72].
- La ***composition modulaire*** introduit la combinaison de composants existants pour produire de nouveaux composants ; ce critère influe sur la réutilisabilité.
- La ***compréhension modulaire*** limite la taille des composants pour qu'ils soient plus aisément compréhensibles.
- La ***continuité modulaire*** assure qu'un petit changement de spécification conduit à de petits changements dans la conception.
- La ***protection modulaire*** garantit qu'une condition anormale à l'exécution reste limitée à un composant.

Ces sous-critères permettent d'énoncer cinq principes de modularité qui doivent être encouragés par les méthodes de conception et respectés par le programmeur :

- Le ***masquage d'information*** : masquage systématique de toute la complexité de réalisation d'un composant tout en fournissant une interface suffisante pour comprendre et utiliser ce composant.
- Les ***interfaces minimales*** : les interfaces des composants doivent être strictement limitées aux caractéristiques utiles. Toute caractéristique est par défaut privée mais peut être rendue explicitement publique.
- Les ***communications explicites*** : lorsque deux composants doivent communiquer, cette communication doit être mentionnée de façon explicite dans la définition des deux composants.
- La ***cohésion forte*** : les éléments regroupés dans un composant doivent former une unité logique.
- Le ***couplage faible*** ou dépendances fonctionnelles limitées : on doit minimiser le nombre de composants en communication.

2.1.5 Généralité

Le critère de généralité consiste à être plus général pour permettre plus de réutilisation. B. Meyer définit quatre principes pour mettre en œuvre la généralité [Meye88] :

- Le ***masquage d'information*** : (cf. section précédente).
- La ***généricité*** ou ***variante de type*** : un composant est générique si le typage de

certaines données n'est pas totalement déterminé. Pour utiliser un composant générique on doit le spécialiser en fixant les types indéfinis.

- Le ***polymorphisme*** ou variante de structures de données et d'algorithmes : un même nom peut être utilisé pour des opérations différentes.
- L'***héritage*** : des concepts communs à différents composants peuvent définir des composants intermédiaires. Des composants plus spécialisés sont obtenus par enrichissement de ces composants intermédiaires.

2.2 Evolution des langages de programmation

2.2.1 Premiers langages informatiques

Jusqu'à la fin des années 50, les langages de programmation se résumaient aux instructions de base de l'ordinateur utilisé. La conception des programmes se limitait alors à l'élaboration d'une longue suite d'instructions codées sous forme binaire, octale ou hexadécimale destinées à des machines en exemplaire souvent unique. Cette approche de la programmation souffrait de limites importantes [MB78] :

- Les programmes étaient liés aux machines et devaient tenir compte de l'organisation de l'unité centrale et du répertoire d'instructions du processeur.
- Cette liaison encourageait l'utilisation de caractéristiques particulières de l'ordinateur pour accélérer le programme au détriment de la portabilité et de la lisibilité.
- La représentation d'un programme comme une suite de commandes binaires était inadaptée à la compréhension de la fonction d'un programme.

Ces défauts ont conduit à l'élaboration de langages de programmation de plus haut niveau théoriquement indépendants d'une machine particulière (FORTRAN I (1954) ou Algol 60 (1958)). Ces langages sont très liés aux mathématiques et permettaient la conception d'applications scientifiques.

A la fin des années 50, la mise en évidence des principales structures algorithmiques a permis le développement d'une nouvelle génération de langages dotés de structures de contrôle plus riches : Algol 60 (1960), Cobol (1960), FORTRAN II (1958). Ces langages n'offraient pas de mécanismes adaptés à la mise en œuvre de la modularité et de la généralité.

2.2.2 Langages procéduraux

Les **langages procéduraux** sont les langages qui intègrent un mécanisme d'abstraction procédurale. L'***abstraction procédurale*** consiste à considérer une suite d'instructions comme une instruction unique. Pascal [Wirt71] est le représentant habituel de cette catégorie de langages dont nous présentons les concepts généraux.

Une **procédure** est composée d'une signature et d'un corps. La **signature** regroupe le nom de la procédure et les paramètres : arguments et résultats. Dans certains nouveaux langages, la signature comporte également deux propositions logiques spécifiant l'action de la procédure : la **précondition** (resp. **postcondition**) décrit une proposition qui doit être vérifiée avant (resp. après) l'exécution de la procédure. Le **corps** est constitué de l'algorithme de la procédure, de variables et éventuellement d'un ensemble de procédures locales.

A l'exécution, la **durée de vie** des variables d'une procédure est identique à la durée d'activation de la procédure : les variables d'une procédure sont utilisables uniquement lorsque cette procédure est activée. Par ailleurs, les déclarations **visibles** depuis l'algorithme d'une procédure sont limitées.

La structure d'un **programme** s'apparente à celle d'une procédure. Un programme est composé de constantes, de types concrets, de variables dites **globales**, de procédures et d'un algorithme (programme principal). Un **type concret** décrit une structure de données bâtie à partir de types de base et de types concrets à l'aide de constructeurs (article, ensemble, table, etc.).

Les langages procéduraux apportent une première solution pour décomposer et organiser un programme. La notion de **bibliothèque** de procédures permet de regrouper des procédures utilisées par différents programmes. Cependant ces langages comportent certaines limites :

- Seules les variables globales ont une durée de vie égale à celle du programme or l'accès à ces variables globales est libre et ne peut pas être contrôlé.
- Le type concret, seul mécanisme de définition de type, ne permet pas de protéger l'accès aux données d'un élément.
- La visibilité est implicite et les dépendances fonctionnelles, induites par les appels de procédures, ne sont pas mises en évidence. Déterminer les conséquences de la modification d'une procédure nécessite d'examiner les algorithmes du programme ce qui complique sa maintenance.
- Aucun mécanisme ne permet de mettre en œuvre la généralité.

2.2.3 Langages modulaires

Les **langages modulaires** proposent un mécanisme d'abstraction modulaire. L'***abstraction modulaire*** consiste à considérer différentes procédures et différentes déclarations (constantes, types, variables) comme un composant unique. Le «module» de Modula-2 [Wirt82] ou le «package» d'ADA [Ichb79] en sont des exemples.

Un **module** est constitué d'un corps et d'une signature. Le **corps** d'un module est composé de différentes caractéristiques. Une **caractéristique** est une constante, un type concret, une variable ou une procédure. Les variables du corps d'un module sont appelées **attributs** pour les distinguer des variables définies dans le corps des procédu-

res. La **signature** définit l'interface d'un module et regroupe généralement son nom, une liste d'importation et une liste d'exportation. La **liste d'importation** d'un module indique explicitement les différents modules utilisés par ce module. Un module est dit ***client*** des modules importés et ***fournisseur*** des modules qui l'importent. La **liste d'exportation** d'un module indique les caractéristiques visibles par ses clients. La signature peut être complétée par une proposition vérifiée par les attributs du module : l'**invariant.**

```
signature module classement_catégorie
   importe       (* Pas de module importé *)
   exporte
      types : classement
      attributs :      (*Pas d'attribut*)
      procédures :
         Initialiser (Entrée C : classement)
         Ajouter_inscription (Entrée C : classement, I : inscription)
         Classer_inscription (Entrée C : classement, I : inscription) : entier
   invariant :         (*Pas d'invariant*)
fin signature classement_catégorie

corps module classement_catégorie
   constantes : taille_max = 1000
   types : classement : nuplet ( T : table [1..taille_max] d'inscription
                                 nb_inscription : entier)
   attributs :         (*Pas d'attribut*)

   procédure Initialiser (Entrée C : classement)
   début C.nb_inscription := 0 fin

   procédure Ajouter_inscription (Entrée C : classement, I : inscription)
   début C.nb_inscription += 1 ; C.T[nb_inscription] := I fin

   procédure Classer_inscription (Entrée C : classement, I : inscription): en-
tier (*classement de l'inscription I en fonction de son résultat*)
fin corps classement_catégorie
```

Exemple 2-1 : Description d'un module.

Un module peut être fournisseur de plusieurs modules, il est alors dit **partagé**. Les importations, reflétant les relations client/fournisseur entre modules, peuvent être représentées sous la forme d'un graphe dit **graphe de dépendances** [Krak82]. Un module constitue généralement une **unité de compilation** compilable individuellement. Le graphe de dépendances permet de déterminer aisément l'ordre de compilation des modules ainsi que les conséquences de la modification d'un module. On notera que la modification d'un module impose la recompilation des clients uniquement si la signature du module est modifiée.

Un **programme** est généralement un module particulier dont la liste d'exportation est vide et qui possède une procédure spécifique activée au lancement du programme. A l'exécution, la **durée de vie** des modules est égale à la durée d'activation du programme utilisateur.

La notion de module met en œuvre le principe de masquage d'information et constitue un mécanisme essentiel pour la modularité et la réutilisabilité des programmes. Les variables globales d'un programme peuvent être distribuées dans les attributs des différents modules qui peuvent alors en contrôler l'accès. La mise en évidence des relation de dépendances entre les modules facilite la maintenance. La notion de module générique (paramétré par un type) constitue un élément de généralité proposé par certains langages comme ADA. Cependant les langages modulaires comportent des inconvénients :

- La structure d'un programme est figée dès la compilation. Toute évolution nécessite la modification et la recompilation de tout ou partie des modules d'un programme, limitant ainsi l'évolutivité et l'adaptabilité.
- La réutilisation de certaines caractéristiques d'un module pour définir un nouveau module nécessite la duplication de ces caractéristiques.

2.2.4 Langages à objets

Le langage de simulation SIMULA développé par une équipe Norvégienne [DN66, BDMN73] a introduit les concepts de classe, d'objet et d'héritage. SIMULA a inspiré de nombreux langages actuels comme SmallTalk, Eiffel, C++, etc. Sans entrer dans les détails, on peut distinguer quatre catégories de langages à objets [MNC*89] : les langages de classes, les langages d'acteurs, les langages de «frames» et les systèmes hybrides. Nous nous limitons ici aux langages de classes plus particulièrement adaptés au développement de grands programmes. Nous appellerons ces langages : langages à objets.

Les **langages à objets** sont les langages qui intègrent, en plus des abstractions procédurales et modulaires, deux mécanismes supplémentaires d'abstraction : classification/instanciation et généralisation/spécialisation. La ***classification*** consiste à décrire une classe de composants à l'aide d'un composant-type définissant les propriétés communes aux composants de la classe. La ***spécialisation*** consiste à définir un nouveau composant en modifiant ou en enrichissant un composant existant. Les langages à objets s'appuient sur trois concepts : l'objet, la classe et l'héritage.

Un **objet** comporte un **corps** composé de variables appelées **attributs** ou variables d'instance et de procédures appelées **méthodes**. Un objet est utilisable via une **signature** qui définit les caractéristiques publiques de l'objet. Une **caractéristique** ou service est un attribut ou une méthode. Chaque objet est **identifié** par le système de façon transparente pour le programmeur. Les attributs d'un objet peuvent contenir des références à d'autres objets. Une **référence** est un lien orienté entre deux objets qui permet notamment la communication entre les objets selon une forme particulière d'appel procédural : l'**envoi de message**.

Une **classe** décrit un modèle d'objet : attributs, méthodes et signature. Un objet est obtenu à partir d'une classe par une opération de «moulage» : l'opération d'**instanciation**. Les attributs et les méthodes décrits dans une classe ne sont pas directement utilisables. Pour utiliser une caractéristique définie dans une classe, le programmeur doit instancier la classe pour obtenir un objet auquel il peut alors demander la caractéristi-

que voulue. La classe est donc un ***composant formel*** d'un programme et l'objet un ***composant effectif***. Un objet peut être perçu comme un composant logiciel ou comme un élément d'un type de données.

```
classe classement
   signature
      hérite_de table
      procedure
         Initialise
         Ajouter_inscription (I : inscription)
         Classer_inscription (I : inscription): entier

   corps
      attributs
      méthode Initialiser...
      méthode Ajouter_inscription (I : inscription)...
      méthode Classer_inscription (I : inscription): entier..
end classement
```

Exemple 2-2 : Description d'une classe.

L'**héritage** est un mécanisme permettant de définir une nouvelle classe, dite **sous-classe**, en modifiant ou en enrichissant les caractéristiques d'une classe existante sans avoir à dupliquer les caractéristiques non modifiées. Sous certaines conditions les objets de cette sous-classe peuvent, à l'exécution, se **substituer** aux objets de la classe. Pour déterminer dynamiquement à l'exécution la méthode à appliquer en fonction de l'objet considéré, les langages à objets intègrent un mécanisme de **liaison dynamique** ou **résolution tardive**.

Un **programme** est souvent une classe particulière sans interface. Dans les langages procéduraux ou modulaires la structure du programme à l'exécution est directement celle du texte du programme. Dans un langage orienté objet, l'organisation du programme à l'exécution n'est pas totalement connue à la compilation puisque les modules effectifs (les objets) existent seulement à l'exécution.

Les langages à objets sont étudiés plus complètement dans le chapitre 3.

L'originalité des langages à objets est d'offrir deux mécanismes de réutilisation de composants : l'héritage entre classes et l'importation de classes par la référence [BMP*90]. Certains langages ajoutent à ces deux mécanismes la notion de classe générique (classe paramétrée par une autre classe) [Meye88]. L'héritage associé aux mécanismes d'instanciation et de liaison dynamique confère aux programmes une bonne adaptabilité et permet un développement progressif des programmes. L'héritage permet la mise en œuvre du critère de généralité.

2.3 Techniques de conception

Dans les années 60, les premiers éléments d'aide au développement de programmes étaient constitués de recueils d'algorithmes et de structures de données classiques [Knut68] que le développeur devait adapter à sa situation. Au fur et à mesure du développement de l'informatique, il est devenu nécessaire d'établir des règles de « bonne programmation » et de définir des méthodes permettant le développement de logiciels complexes.

2.3.1 Programmation structurée

En 1968 Dijkstra, [Dijk68a] jette les bases d'une réflexion méthodologique pour la conception de programmes de qualité. Cet article, qui met en évidence les effets néfastes de l'utilisation de l'instruction «GOTO», cristallise un courant de pensée existant depuis le début des années 60. Ce courant propose de bannir l'utilisation de cette instruction dont l'inutilité semble avoir été démontrée dès 1966 [BJ66] et qui peut s'énoncer de la façon suivante [LMW79] :

Structure theoreme : Any proper program is function equivalent to a structured program with basis set (sequence, if-then-else, while-do), using functions and predicates of the original program and assignements and test on one additional counter.

La **programmation structurée** [DDH72, CGL*75, LMW79] s'est développée au début des années 70. Elle s'appuie sur une approche cartésienne : on décompose progressivement le problème initial en sous-problèmes supposés plus simples. La composition des solutions partielles apporte une solution au problème initial.

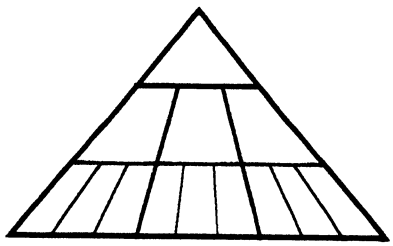


Figure 2-1 : Topologie d'un programme structuré [Rose87].

Le programme obtenu a une structure en couches et le graphe de dépendance est une arborescence. Cette approche de la programmation tente de généraliser la construction de programmes valides en encourageant le développement en parallèle d'un programme et de sa preuve de validité [Dijk68c, Hoar69]. La stratégie d'organisation en niveaux successifs a été introduite par E.W. Dijkstra [Dijk68b] pour l'approche ascendante par composition et par N. Wirth [Wirt71a] pour l'approche descendante par raffinements successifs.

Si l'apport de la programmation structurée est indéniable [CGL*75], l'application stricte de cette méthode comporte des limites importantes [Pier91, Rose87] :

- La programmation structurée nécessite une appréhension globale du problème incompatible avec le développement et la maintenance de grands programmes. Par ailleurs, la programmation structurée ne propose pas de critère précis pour décomposer un problème en sous-problèmes.
- Les données n'apparaissent pas explicitement et sont noyées dans les algorithmes. Toute évolution de la structure de ces données peut affecter une large part du programme.
- Le raffinement indépendant des sous-problèmes entraîne des répétitions de suite d'instructions pour des sous-problèmes identiques ou voisins.

Une solution à la redondance de code consiste à remonter les procédures et les données dupliquées jusqu'au niveau où elles sont visibles de toutes les procédures les utilisant. On aboutit à la topologie représentée par la figure 2-2 [Rose87].

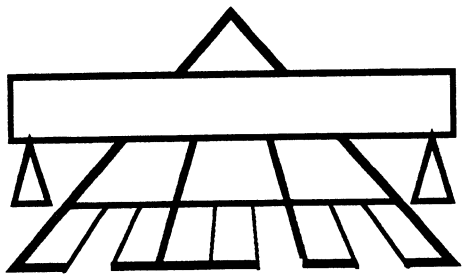


Figure 2-2 : Topologie d'un programme structuré optimisé [Rose87].

Cette optimisation est cependant en désaccord avec la programmation structurée. En effet les procédures « délocalisées » ne correspondent plus à des procédures élémentaires du niveau dans lequel elles apparaissent. En outre les procédures délocalisées sont utilisées par les niveaux supérieurs mais également par les niveaux inférieurs.

2.3.2 Conception fonctionnelle

Le paradigme de la **conception fonctionnelle** reste le même que celui de la programmation structurée [IGL82, YC79] : approche cartésienne du problème à résoudre et décomposition itérative et hiérarchique en différents niveaux. La conception fonctionnelle s'appuie généralement sur une spécification du programme réalisée à l'aide d'un diagramme de flux de données.

Un diagramme de flux de données permet de décomposer une fonction en sous fonctions et de décrire d'une part les transitions des données entre ces sous-fonctions et d'autre part l'action de chaque fonction sur les données qu'elle reçoit. Le programme est

représenté comme une fonction définie par ses entrées et ses sorties et il peut être décomposé en sous-fonctions comme toute autre fonction.

Si la conception fonctionnelle propose un critère de décomposition du programme, le programme ainsi obtenu souffre des défauts inhérents au critère de décomposition :

- L'expérience montre que dans la phase d'exploitation d'un logiciel, la plupart des modifications demandées par les utilisateurs concernent les fonctionnalités du logiciel. Dans une architecture fonctionnelle toute évolution des fonctionnalités du programme peut remettre en cause une grande part de son organisation.
- On retrouve les inconvénients de la programmation structurée comme : la sensibilité aux évolutions des structures de données et la redondance de code.

2.3.3 Décomposition selon les données

La décomposition selon les données [Jack83, Warn81] propose, pour tenter de pallier les inconvénients de l'approche fonctionnelle, une organisation du programme basée sur les structures de données. L'argument principal est que pour obtenir un programme compréhensible et maintenable, il faut lui donner une structure correspondant à l'organisation des données.

Les trois structures de base pour les structures de données ainsi que pour les algorithmes sont : la séquence, la sélection et l'itération. Le processus de conception consiste en [CGL*75] : i) définition des structures de données du problème, ii) création d'une structure de programme similaire à celle des structures de données et iii) expression des diverses tâches du programme sous forme d'opérations exécutables et affectation de chacune d'elles à un composant du programme.

La technique de conception dirigée par les structures de données s'applique essentiellement aux situations où les données ont un rôle central comme en gestion mais reste dans le courant des méthodes hiérarchisées, descendantes et structurées. Le principe de calquer l'architecture logicielle sur la structure de données peut s'avérer inacceptable dans certains types de problème. Par exemple les applications à forte composante fonctionnelle dans lesquelles les données servent essentiellement à réaliser des fonctions sont difficilement concevables selon cette approche.

2.3.4 Organisation en machines abstraites

La **décomposition en machines abstraites** est une approche structurée comme les précédentes [CGL*75, IGL84]. Le programme est décomposé en différents niveaux, chaque niveau étant constitué d'un ou plusieurs processeurs abstraits. Chaque processeur abstrait ou machine abstraite possède ses ressources propres et réalise un ensemble de fonctions. Le plus bas niveau de la hiérarchie est constitué par les instructions de base du langage ou du processeur. Les machines abstraites sont liées par deux types de

liens [Pier91] : des liens d'abstraction issus de la décomposition et des liens d'utilisation (relation client/fournisseur). Selon les contraintes imposées aux liens d'utilisation, l'organisation peut être strictement hiérarchique ou plus modulaire. Dans MACH [THO85] par exemple, une machine peut utiliser les fonctions des machines de niveau d'abstraction inférieur mais les machines des niveaux supérieurs lui sont inaccessibles. Une structure strictement hiérarchique permet d'organiser les machines en couches successives. La notion de machine abstraite est aisément programmable à l'aide de la notion de module.

Une organisation judicieuse des machines confère au programme les propriétés d'évolutivité et d'adaptabilité en limitant le nombre de machines affectées par une modification.

2.3.5 Types abstraits de données

Les types abstraits de données ne sont pas à proprement parler une technique de conception mais plutôt une technique de spécification. Par opposition aux types concrets décrivant un domaine de valeurs par une structure de données, un type abstrait décrit un domaine par les opérations qui sont applicables aux éléments du domaine. Les types abstraits sont présentés plus complètement dans l'annexe A.

Utilisée comme technique de conception, cette approche permet de protéger les données, d'en faciliter l'évolution, d'enrichir les types de données disponibles et de disposer d'un critère de décomposition du programme. Chaque type abstrait peut donner naissance à un composant logiciel. Un type abstrait peut aisément être programmé à l'aide de la notion de module.

2.3.6 Conception dirigée par les objets

La conception dirigée par les objets est issue des travaux de Parnas [Parn72]. Dans une conception dirigée par les objets, les modules logiciels sont obtenus par abstraction des objets du domaine de l'application. Les fonctions de l'application sont réparties sur ces objets. Ces principes ont été concrétisés au début des années 80 dans une méthode développée initialement pour le langage ADA [Booc82, Booc83]. Cette première méthode s'appuyait sur le mécanisme de module du langage ADA. Les méthodes actuelles la prolongent en intégrant les techniques à objets introduites par les langages de programmation à objets [Booc91, Ferb90, Meye88, RBP*91, WWW91].

La conception dirigée par les objets diffère des approches précédentes par son caractère non hiérarchique : la structure du programme reflète, dans cette approche, l'organisation du domaine d'application. Il est généralement admis que la structure ainsi obtenue est plus invariante que les organisations produites par d'autres techniques de conception. En effet, les abstractions des objets du domaine constituent souvent des invariants plus forts que les fonctions de l'application, les structures de données

manipulées ou une hiérarchie de machines abstraites.

Le chapitre 3 présente plus en détail cette technique de conception.

3 Bases de données

Avec le développement des applications de gestion, les questions de stockage des données sur les mémoires secondaires permanentes puis de leur restitution se sont posées. Avec l'augmentation de la capacité de stockage des supports magnétiques et l'utilisation croissante de l'ordinateur, les systèmes de gestion de fichiers ont progressivement laissé la place aux systèmes de gestion de bases de données.

3.1 Généralités

3.1.1 Objectifs généraux des SGBD

Une **base de données** est un ensemble structuré de données enregistrées sur des supports permanents accessibles par l'ordinateur pour satisfaire simultanément plusieurs utilisateurs [DA82]. Un **système de gestion de bases de données** (SGBD) est un logiciel permettant d'interagir avec une base de données. Les objectifs généraux fixés aux systèmes de gestion de bases de données sont les suivants [DA82, Gard83] :

- ***Description des données*** : le SGBD doit permettre de décrire la structure des données stockées dans la base ainsi que les liens existant entre ces données.
- ***Intégrité des données*** : le concepteur doit pouvoir enrichir la description structurelle de la base de données par des règles d'intégrité dont la vérification doit être assurée par le SGBD.
- ***Utilisation des données*** : les informaticiens comme les utilisateurs non informaticiens doivent pouvoir utiliser la base de données à l'aide de langages adaptés à leurs compétences respectives.
- ***Partage*** et ***concurrence*** : différents utilisateurs doivent pouvoir accéder simultanément à la base : le SGBD doit gérer ces accès concurrents et maintenir l'intégrité des données.
- ***Administration centralisée*** : la responsabilité de la définition et de l'évolution du schéma de la base de données doit être attribuée à un administrateur.

A ces objectifs généraux, on peut ajouter des préoccupations techniques (efficacité, sécurité, protection, reprise) et des services administratifs (classes d'utilisateurs, privilèges, confidentialité).

3.1.2 Indépendance données-programmes

Garantir l'évolutivité de la base de données et des programmes est un gage de longévité et de rentabilité des développements réalisés. Pour atteindre cet objectif, deux niveaux d'indépendance entre les données stockées sur les mémoires secondaires et les programmes qui les utilisent ont été définis :

- L'***indépendance physique*** : il y a indépendance physique lorsque le SGBD permet de décrire la structure de la base de données sans faire référence à la façon dont les données sont effectivement stockées sur le support permanent. Dans ce cas, l'organisation physique de la base de données (fichiers, index, etc.) peut évoluer sans affecter les programmes qui utilisent les données.
- L'***indépendance logique*** : chaque utilisateur, par sa fonction et ses besoins, doit pouvoir disposer d'une vue limitée et adaptée de la base de données. Il est également souhaitable que les programmes accèdent à la base de données via de telles vues limitées. Il y a indépendance logique si le SGBD permet de définir différentes vues et si il gère les transformations des données entre le schéma de la base et ces différentes vues. Dans ce cas, l'organisation de la base de données peut évoluer (modifications des structures des données, de leurs liens, etc.) sans affecter les programmes qui utilisent les données par l'intermédiaire de vues.

3.1.3 Conception d'une base de données : niveaux de représentation

Le groupe ANSI/X3/SPARC a proposé d'organiser le processus de conception du schéma d'une base de données en trois niveaux [Ansi75] :

- Au ***niveau conceptuel*** le concepteur élabore le **schéma conceptuel** qui spécifie le contenu de la base de données. Le formalisme utilisé, appelé **modèle conceptuel de données**, est théoriquement indépendant d'un système de gestion de bases de données particulier. Ainsi les évolutions techniques ne doivent pas remettre en cause le travail de spécification réalisé dans cette étape.
- Le ***niveau interne*** est constitué par un système de gestion de bases de données ou une famille de SGBD. Ce niveau est habituellement décomposé en niveau logique et niveau physique. Le ***niveau physique*** est constitué par le SGBD cible choisi pour gérer la base de données. Le **schéma physique** est une représentation de la base de données à l'aide des concepts du **modèle physique de données** de ce système cible. Le ***niveau logique*** est une abstraction du niveau physique. Un **modèle logique de données** est une abstraction d'une famille de modèles de données physiques. Le **schéma logique** élaboré à ce niveau est adapté à une famille de SGBD cibles. Ce niveau constitue donc une étape intermédiaire entre le schéma conceptuel abstrait et un schéma physique particulier.
- Le ***niveau externe*** permet de définir différents schémas externes adaptés à chaque utilisateur de la base de données. Un **schéma externe** est élaboré à partir du schéma conceptuel par des opérations de masquage, de restructuration, de déduction, etc. Les informations contenues dans un schéma externe peuvent toutes être obtenues à partir des informations présentes dans le schéma conceptuel.

Le modèle de données utilisé pour représenter les schémas externes est en général le modèle conceptuel de données.

3.1.4 Elaboration du schéma d'une base de données

Contrairement aux programmes pour lesquels il existe différents critères de décomposition (fonctionnel, structure de données, objet, etc.) une base de données est toujours conçue comme un modèle du domaine des applications. Le critère de décomposition d'une base de données est donc constitué par les objets du domaine. Cependant les modèles de données des SGBD influent évidemment sur le schéma de la base de données. Dans certains cas, le concepteur est obligé d'altérer la perception qu'il a de la réalité pour obtenir un schéma opérationnel.

3.2 Evolution des systèmes de gestion de données persistantes

3.2.1 Systèmes de gestion de fichiers

Les années 50 ont vu le développement des principales notions concernant les fichiers ainsi que les principales méthodes d'organisation et d'accès aux données stockées dans les fichiers. La notion de **fichier** permet d'assurer l'indépendance entre les programmes et la localisation physique des données sur les mémoires secondaires. La notion d'**article** ou d'enregistrement permet de décrire un fichier comme une suite d'éléments structurellement identiques. Les systèmes de gestion de fichiers (SGF) prennent en charge essentiellement l'identification et la localisation des fichiers, l'allocation de l'espace des mémoires secondaires, le partage des fichiers et la confidentialité des données. Les systèmes de gestion de fichiers comportent des limites importantes :

- Ils ne permettent pas l'expression ni l'exploitation des liens explicites entre fichiers ou entre articles.
- L'indépendance physique n'est pas assurée : la structure des fichiers est définie dans les programmes qui les utilisent. Les programmes sont donc tributaires des changements de l'organisation physique des données.
- Les définitions des structures de fichiers sont réparties dans les programmes nuisant à l'évolutivité globale du système.

Les faiblesses des systèmes de gestion de fichiers, la croissance rapide des capacités de stockage et l'augmentation de l'utilisation de l'ordinateur ont favorisé l'émergence du concept de base de données dans les années 60.

3.2.2 Systèmes de gestion de bases de données de première génération

Un système de gestion de bases de données se distingue d'un système de gestion de fichiers en séparant la description de la structure de la base de données, le **schéma** de la base de données, et les programmes qui utilisent la base de données. La description

du schéma est réalisée à l'aide d'un **langage de définition de données** (LDD) qui s'appuie sur le modèle de données du SGBD. Le **modèle de données** est l'ensemble des concepts offerts par le SGBD pour représenter le schéma de la base.

Le **modèle hiérarchique** du milieu des années 60 propose une représentation des données sous la forme d'arborescences. Les nœuds de ces arborescences sont des **types d'enregistrements** composés d'une liste d'attributs et les arcs sont des **liens** orientés.

Le modèle hiérarchique ne permet pas par exemple de décrire directement et simplement les associations m-n, ni les cycles. Leur représentation impose soit la duplication des données soit l'utilisation de types d'enregistrements virtuels.

Apparu à la fin des années 60, le modèle réseau CODASYL a été proposé par le Data Base Task Group [CODA69, CODA71]. Le modèle réseau SOCRATE est apparu au début des années 70 à Grenoble [Abri72]. Nous n'entrons pas dans les détails distinguant ces deux modèles et nous présentons brièvement les concepts de base.

Le **modèle réseau** repose sur deux concepts très proches de ceux du modèle hiérarchique : l'**entité** (type d'enregistrements) et le **lien**. Cependant, à la différence du modèle hiérarchique, aucune structure particulière n'est imposée au schéma de la base de données. Ce schéma est décrit par un graphe dans lequel les liens définissent des associations de nature 1-n. On notera que le modèle SOCRATE permet de représenter directement différentes formes d'associations et qu'il permet de gérer automatiquement les liens réciproques.

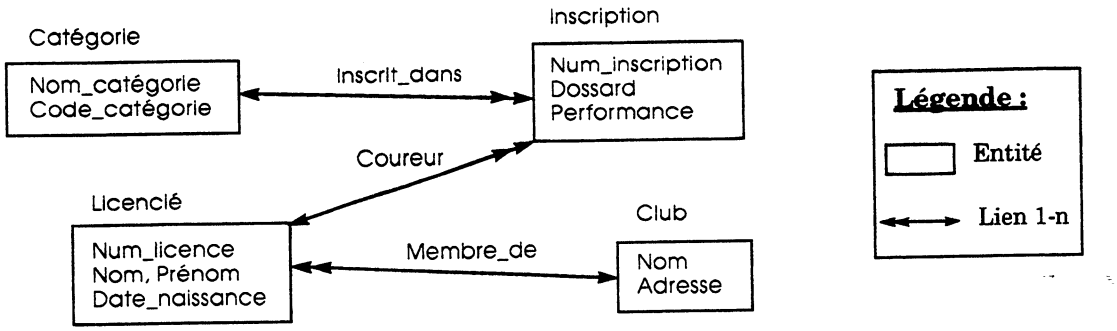


Figure 2-3 : Représentation graphique d'un schéma CODASYL.

Dans les systèmes de gestion de bases de données de la première génération (hiérarchiques et réseaux) l'accès aux données se fait en «naviguant» dans la structure de la base de données à partir de l'enregistrement courant. Cette navigation doit respecter le schéma de la base. Les langages de manipulation de données (LMD) sont donc impératifs et navigationnels.

Avec notamment la séparation données persistantes/programmes, les SGBD de la première génération apportent des avantages incontestables par rapport aux systèmes de gestion de fichiers. Ils comportent cependant des limites [DA82] :

- Pour obtenir une base de données efficace le concepteur doit connaître les solutions techniques mises en œuvre par le SGBD.
- L'indépendance logique est limitée : elle repose sur la notion de sous-schéma qui correspond plus à une opération de filtrage qu'à l'élaboration d'un modèle conceptuel partiel.
- Le langage de manipulation de données, de type impératif, ne permet pas un accès aisé des données aux utilisateurs non informaticiens.
- Les contraintes d'intégrité ne peuvent pas être décrites dans le schéma et a fortiori elles ne sont pas vérifiées automatiquement. La vérification des contraintes d'intégrité doit être programmée explicitement dans chaque programme.

3.2.3 Systèmes relationnels

La théorie des ensembles et des relations constitue la base formelle du modèle relationnel [Codd70]. Les systèmes de gestion de bases de données commerciaux bâtis sur ce modèle apparaissent au début des années 80, et sont actuellement en train d'envahir le marché des bases de données. Le modèle relationnel est simple et constitué d'un petit nombre de concepts [DA82].

Une **relation** est une forme de fichier contenant des nuplets. Un **nuplet** est une liste de valeurs appartenant à des domaines de base (entier, chaîne, etc.). La structure d'une relation est décrite par un schéma. Un **schéma de relation** est la donnée d'une liste d'**attributs** et de la liste des domaines de base de ces attributs. Une relation est donc un sous-ensemble du produit cartésien des domaines listés dans son schéma de relation. Une **clé** est un attribut ou un ensemble d'attributs permettant d'identifier chaque nuplet d'une relation. Le **schéma** de la base de données est constitué d'un ensemble de schémas de relations complété par un ensemble de contraintes d'intégrité. Les **contraintes d'intégrité** sont des propriétés que doivent vérifier les données contenues dans la base de données. La vérification et le maintien des contraintes d'intégrité doivent être assurés par le SGBD. Le mécanisme de **vue** permet de définir des relations virtuelles construites à partir de relations ou à partir d'autres vues. Les vues sont traitées, dans le langage de manipulation, comme des relations. Pour l'utilisateur, une vue ne diffère pas d'une relation. La notion de vue permet d'atteindre l'objectif d'indépendance logique : le schéma de la base de données peut être modifié sans perturber les programmes accédant aux données par des vues.

```
relation Club (Nom_club : Chaîne, Adresse : Chaîne)

relation Licencié ( Num_licence : Entier, Nom : Chaîne, Prénom : Chaîne,
                    Nom_club : Chaîne)

relation Catégorie (Code_catégorie : Entier, Nom_catégorie : Chaîne)
```

```
relation Inscription (     Num_inscription : Entier, Num_licence : Entier,
                           Code_catégorie : Entier, Dossard : Entier,
                           Performance : Réel)

vue Inscription _complète
          ( Num_inscription : Entier, Nom_catégorie : Chaîne,
            Nom : Chaîne, Prénom : Chaîne, Nom_club : Chaîne,
            Dossard : Entier, Performance : Réel)
```

Exemple 2-3 : Un schéma de base de données relationnelle.

Les **langages relationnels** s'appuient sur une ***algèbre relationnelle*** composée de cinq opérateurs de base (projection, sélection, union, différence et produit cartésien) et de différents opérateurs dérivables (intersection, division, jointure, etc.) pour offrir des langages de manipulation de données déclaratifs et de haut niveau. Cependant les langages relationnels ne sont pas complets et, pour développer des applications, ils doivent être intégrés dans des langages de programmation hôtes (les langages procéduraux par exemple).

Le maintien de la cohérence, lors de la modification de la base de données, est assuré par un mécanisme de transaction. Une **transaction** est une séquence d'appels au système de gestion de bases de données. Une transaction est une unité logique atomique qui fait évoluer la base de données d'un état cohérent vers un autre état cohérent.

Différentes anomalies liées au schéma relationnel peuvent survenir à l'exploitation de la base de données (anomalies en mise-à-jour, en insertion, en suppression etc.). Pour obtenir de bons schémas relationnels, différents critères de qualité ou **formes normales** ont été définis. Ces critères s'appuient sur la mise en évidence de dépendances entre les attributs d'une relation : dépendances fonctionnelles, multivaluées, hiérarchiques ou produits. Les formes normales ne constituent pas à proprement parler une méthode de conception mais plutôt des niveaux de qualité des schémas relationnels.

Un modèle formalisé, des langages de haut niveau, un maintien automatique de l'intégrité des données, un haut degré d'indépendance (logique et physique) et des critères de qualité des schémas sont les apports principaux et importants des systèmes relationnels. On peut cependant regretter deux inconvénients majeurs [DLR91] :

- Le modèle de données est adapté aux applications classiques de gestion mais il s'avère trop simple pour permettre une représentation aisée des données «complexes» et structurées. La contrainte d'atomicité des attributs est trop forte dans le cas d'applications telles que la CAO, les AGL, le multimédia, etc.
- L'intégration des langages relationnels dans des langages classiques imposée par l'incomplétude des premiers entraîne divers dysfonctionnements qui influent sur l'efficacité du programmeur et sur la fiabilité du programme.

3.2.4 Modèles sémantiques

Les modèles hiérarchiques, réseaux ou relationnels sont souvent trop pauvres pour permettre d'élaborer directement le schéma de la base de données. Pour faciliter les modélisations préliminaires, des modèles plus riches ont été développés : les modèles sémantiques [HK87, PM88]. Depuis les premières propositions comme le modèle Z [Abri74] ou le modèle entité/relation [Chen76], de nombreux modèles ont été décrits [Borg85, Codd79, HM83, SFL83, Ship81]. Quelques-uns ont donné naissance à des systèmes de gestion de bases de données mais le plus souvent ils sont restés des propositions théoriques utilisables en amont d'un SGBD conventionnel.

Les **modèles sémantiques** proposent généralement une large gamme de constructeurs dont les rôles se recouvrent partiellement. Nous présentons différents concepts et différents mécanismes d'abstraction généralement présents dans un modèle sémantique [DLR91].

Le monde réel est perçu comme un ensemble d'**entités** reliées par différents **liens**. Les entités sont regroupées en classes d'entités. Une **classe** décrit un modèle d'entité par un ensemble d'attributs. Les **attributs** décrivent des liens entre entités ou des propriétés. Les liens sont binaires et orientés, monovalués ou multivalués. Les entités peuvent éventuellement être **identifiées** indépendamment de la valeur de leurs attributs.

Le **groupement** est un mécanisme d'abstraction permettant de construire une nouvelle entité à partir d'un ensemble d'entités de même classe. La différence entre une classe et un groupement est qu'une classe, malgré le groupement d'objets qu'elle représente, n'est pas une entité. L'**agrégation** est un mécanisme d'abstraction permettant de construire une nouvelle entité à partir d'entités de classes différentes [SS77]. Une **association** permet de relier les entités entre elles mais ne constitue pas une nouvelle entité. Les associations sont n-aires et sans orientation particulière. Une association peut être enrichie par des attributs et par des cardinalités.

La **spécialisation** permet de définir une nouvelle classe à partir d'une classe existante. La **généralisation** permet de définir une nouvelle classe comme l'union de différentes classes. Les classes munies de la relation de spécialisation/généralisation forment une hiérarchie dite d'**héritage** similaire à la relation d'héritage des langages à objets. Les attributs **dérivés** sont des attributs calculés à partir d'autres attributs. Une classe dérivée est un sous-ensemble d'une classe obtenu par exemple par filtrage des entités de cette dernière. Les classes **dérivées** sont à rapprocher du mécanisme de vue des systèmes relationnels.

Les modèles sémantiques permettent généralement d'exprimer des **contraintes d'intégrité** pour affiner le schéma de données.

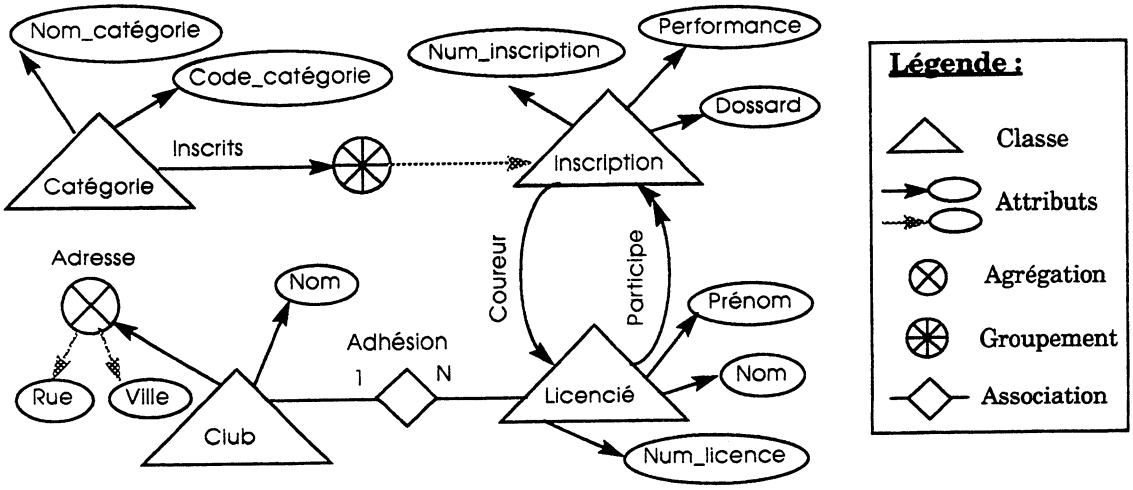


Figure 2-4 : Un modèle sémantique.

Les modèles sémantiques offrent de larges facilités pour représenter les données mais permettent rarement de représenter la dynamique du système et les traitements. Le domaine des modèles sémantiques est encore un domaine de recherche et les performances actuelles des quelques réalisations existantes ne permettent pas de concurrencer les systèmes de gestion de bases de données relationnelles [DLR91].

3.2.5 Systèmes «relationnels étendus»

Pour conserver les apports du modèle relationnel, notamment sa base formelle, mais pallier sa trop grande simplicité, différentes extensions ont été proposées [AC89, DLR91, Rich89].

Les modèles à **valeurs structurées** (NF2 : Non First Normal Form) proposent de généraliser le modèle relationnel en abandonnant la première forme normale qui impose l'atomicité des attributs. Dans ces modèles, la valeur d'un attribut n'appartient plus nécessairement à un domaine de base. La valeur d'un attribut peut être « complexe » c'est-à-dire élaborée à l'aide d'une combinaison de constructeurs (ensemble, liste, nuplet, etc.) appliqués aux domaines de base. Comme pour le modèle relationnel, des algèbres ont été définies pour les modèles à valeurs structurées [AB84]. Un problème cependant n'est pas résolu : une donnée «complexe» ne peut pas être explicitement partagée par plusieurs nuplets.

Les modèles avec **identité** d'objet proposent de mettre en évidence et de généraliser la notion d'identité d'objet. Cette notion, plus ou moins présente dans les modèles conventionnels à travers la notion de clé, est directement héritée des modèles sémantiques. L'identité d'objet permet le partage des données.

Les systèmes avec **types abstraits** prolongent les modèles à valeurs structurées en substituant le typage concret des attributs d'une relation (types de base ou types structurés) par un typage abstrait (types abstraits de données).

3.2.6 Langages de programmation de bases de données

Les langages de programmation de bases de données [DLR91, Dech93] tentent d'apporter une solution aux difficultés d'intégration entre les langages de programmation classiques et les modèles des SGBD. On peut distinguer deux approches [DLR91] :

- La première, suivie par exemple dans Pascal/R [SM80], consiste à intégrer les fonctions d'un SGBD dans un langage de programmation existant. Pour cela, le langage de programmation est étendu par de nouveaux types de données, notamment les collections, et par les opérateurs associés. La persistance est introduite par l'intermédiaire de ces nouveaux types de données.
- L'objectif de la seconde est d'ajouter la persistance aux types de données manipulés dans le langage de programmation retenu. Dans ces systèmes, dont PS-algol [ACC81] est un exemple, l'accent est mis sur l'orthogonalité entre le typage et la persistance (toute donnée peut persister quelque soit son type) ainsi que sur la modularité des programmes.

Les langages de programmation de bases de données offrent une puissance de calcul suffisante pour programmer des applications. En revanche, les services de manipulations des données persistantes offertes par les langages de requêtes ne sont pas exploitables puisque tout accès aux données persistantes doit être réalisé par programme. Par ailleurs ces systèmes ne permettent pas de tirer parti des recherches menées pour optimiser les requêtes. Or cette optimisation est vitale lorsque plusieurs utilisateurs manipulent simultanément des données volumineuses.

3.2.7 SGBD «orientés objets»

Les systèmes de gestion de bases de données orientés objets tirent parti d'une part des différents travaux récents menés dans le domaine des bases de données, modèles sémantiques et relationnels étendus, et d'autre part des techniques à objets développées pour les langages de programmation. De nombreux systèmes de ce type sont actuellement en cours d'élaboration dans les laboratoires ou déjà commercialisés. Cependant, malgré différentes propositions [ABD*89, Beer89], il n'existe pas actuellement de consensus autour d'un modèle à objets pour les SGBD. Compte tenu des variations sur l'interprétation des termes «orienté objet», nous fixons les caractéristiques minimales des SGBD que nous qualifions d'orientés objets.

Un **système de gestion de base de données orientées objets** (SGBDOO) offre les notions d'encapsulation, d'identification, de classification et d'héritage. Evidemment, il assure la persistance des objets et on parlera de **base d'objets** plutôt que de base de données. Nous présentons brièvement le SGBDOO O_2 [ODeu89] qui est représentatif

de notre idée des SGBDOO. Le système O_2 a été développé à la fin des années 80 par le GIP Altaïr et il est actuellement commercialisé par la société O_2 Technology.

Dans le modèle de données O_2, la notion d'objet est distincte de celle de valeur. Une **valeur** est une donnée simple appartenant à un type de base (entier, chaîne, etc.) ou une donnée structurée appartenant à un **type** (concret) défini par une combinaison de constructeurs (ensemble, liste ou nuplet) appliqués à des types de base. Un **objet**, comme dans les langages de programmation, est instance d'une classe. Une **classe** comporte une structure de données (type concret) et des méthodes. Les méthodes d'une classe peuvent être publiques ou privées et la structure de données peut être publique (lecture/écriture ou lecture seule) ou privée. L'usage de l'héritage est réservé aux classes et ne peut être appliqué aux types. Les objets sont identifiés par le système et peuvent être liés entre eux par des **références**.

Un **schéma** regroupe un ensemble de classes, de types, de noms d'objets ou de valeurs, de procédures et d'applications. Une **base** est une «instance persistante» d'un schéma. Le couple schéma/base est analogue au couple classe/objet et un schéma peut générer différentes bases comme une classe peut générer différents objets. Contrairement aux modèles de données habituels, une classe O_2 ne regroupe pas les objets qu'elle a générés. Les classes ne constituent donc pas les points d'entrée dans la base d'objets. Un **nom d'objet** est une variable globale du schéma accessible directement par toutes les classes du schéma. La persistance se propage à partir des noms d'objets par les références : tous les objets accessibles directement ou indirectement à partir d'un nom d'objet sont persistants. Les noms d'objets constituent donc les points d'entrée dans la base d'objets persistants. Un schéma peut **exporter** (resp. **importer**) vers (resp. depuis) d'autres schémas des classes et des types.

Une **application** est définie dans un schéma et elle est composée de variables et de transactions et peut être lancée par l'utilisateur. Les **variables** sont accessibles uniquement aux transactions et leur durée de vie est égale à celle de l'application. Une **transaction** réalise une suite d'opérations atomiques sur la base d'objets. Les transactions publiques peuvent être activées par l'utilisateur. Aucune transaction ne peut être déclenchée par des objets. O_2 offre un ensemble de classes spécialisées (boutons, choix, boîtes de dialogue, etc.) pour faciliter la programmation de l'interface entre l'application et l'utilisateur.

```
schéma Gestion_d'une_Compétition
   export  classe

   nom d'objet compétition_courante : Compétition

   application Inscription
      variables
      transaction
         inscription_individuelle,
         annulation_d'une_inscription,
         inscription_par_club...
   fin (*Inscription*)

   classe Compétition
```

```
      signature
           nom_compétition : chaîne...
        méthode ajout_catégorie (c : Catégorie)...
      corps
        attribut
           catégories : ensemble_de (Catégorie)...
        méthode ajout_catégorie (c : Catégorie)...
   fin (*Compétition*)

   classe Catégorie
     signature
          nom_catégorie : chaîne...
        méthode ...
     corps...
   fin (*Catégorie*)

fin (*Gestion_d'une_Compétition*)
```

Exemple 2-4 : Définition d'un schéma en O_2.

O_2 apporte une solution intéressante au besoin de modularisation et de protection des données persistantes que nous avons soulevé lors de l'étude des langages de programmation. Nous verrons au chapitre 4 que le choix de limiter la définition des transactions aux applications et l'impossibilité de déclencher une transaction par envoi de message sont des contraintes qui s'avèrent, à l'usage, trop limitatives.

3.3 Techniques de conception

Comme nous l'avons déjà mentionné, le schéma d'une base de données est conçu comme un modèle du domaine d'application. Les différences entre plusieurs schémas d'une base de données dépendent donc plus des concepts du modèle de données utilisé que du critère de décomposition. Dans les entreprises, l'enjeu stratégique de l'information a accéléré l'émergence et la multiplication de recherches sur les méthodes de conception de systèmes d'information. Après avoir rappelé la définition de la notion de système d'information, nous évoquons brièvement les particularités de quelques méthodes d'analyse et de conception de systèmes d'information.

3.3.1 Systèmes d'information et méthodes de conception

Actuellement, la définition de la notion de système d'information s'appuie sur une modélisation systémique des entreprises. Il existe deux approches générales pour construire des modèles : l'approche analytique et l'approche systémique.

Dans l'***approche analytique*** qui prévaut depuis plus de trois siècles, le modèle d'un phénomène est obtenu selon une approche cartésienne. Le phénomène est abordé comme un tout indépendant du contexte (système fermé et isolé). Il est décomposé autant de fois que nécessaire pour réduire les difficultés et permettre l'analyse et la

compréhension du phénomène. Le modèle est ensuite obtenu par recomposition.

Les ***méthodes d'analyse*** des années 60-70 s'appuyaient sur une approche cartésienne. Le système d'information (SI) était perçu comme un processeur de traitement de l'information caractérisé par sa fonction. La construction du SI demandait une analyse des besoins pour définir les différentes fonctions à réaliser. Les SI étaient donc conçus essentiellement dans le but d'assister les opérateurs et ils ressemblaient généralement à une mosaïque de petits systèmes sans grande cohésion. Les redondances et les incohérences dans les informations étaient nombreuses et entraînaient des difficultés d'exploitation et de communication [TRC85]. L'expérience a par ailleur montré qu'en général l'analyse des besoins ne conduisait pas à la mise en évidence de l'ensemble des informations nécessaires au fonctionnement de l'organisation [Roll86].

La systémique a pour objet la modélisation des phénomènes complexes comme les systèmes planétaires, les organismes vivants, les sociétés, etc. L'***approche systémique*** propose d'aborder a priori le phénomène à modéliser comme un système organisé en relation avec son environnement. Le modèle d'un phénomène est généralement élaboré à partir d'un système-type choisi dans une gamme de neuf systèmes génériques de complexité croissante : du plus simple au plus organisé [LeMo77, LeMo90]. Nous retiendrons uniquement le système-type du sixième niveau (cf. figure 2-5) : le système agit (système opérant), mémorise (système d'information) et décide (système de décision). Le système d'information mémorise sous forme symbolique les opérations réalisées par le système opérant et met ces informations à disposition du système de décision. En retour le système de décision décide et en informe le système opérant et le système d'information. Le système d'information constitue donc une représentation abstraite du système opérant.

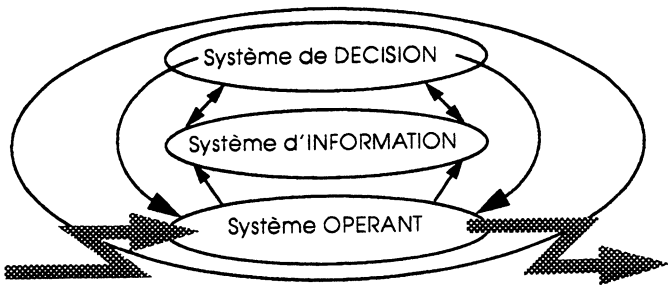


Figure 2-5 : Le sixième niveau d'organisation d'un système.

Les ***méthodes systémiques*** des années 70-80 s'appuient sur une approche systémique des organisations. L'entreprise est modélisée à partir d'un système-type du sixième niveau. Cette approche établit donc que le système d'information d'une organisation doit être conçu comme un modèle du système opérant de cette organisation. Le SI est abordé suivant trois dimensions : structurelle (structures des informations), fonctionnel-

le (opérations réalisées) et dynamique (ordonnancement des opérations).

Les méthodes systémiques permettent de remédier aux inconvénients des méthodes analytiques et conduisent à des systèmes d'informations homogènes et cohérents. Cependant l'application stricte des principes systémiques conduit à reproduire dans le système d'information les dysfonctionnements de l'entreprise et à négliger le soutien que peut apporter l'ordinateur au fonctionnement du système opérant. En effet, *"les besoins en matière d'outils ne sont pas nécessairement inclus dans l'image abstraite du système opérant"* [Roll86].

3.3.2 La méthode MERISE

La méthode MERISE [TRC85] est une méthode systémique définie à la fin des années 70. Elle est largement utilisée dans les entreprises depuis le début de la décennie 80. Plus récemment, elle a été étendue pour intégrer les facilités offertes par les techniques à objets. Nous évoquons ces extensions au chapitre 3.

Les concepteurs de MERISE ont identifié trois niveaux d'abstraction : les niveaux conceptuel, organisationnel et opérationnel.

Le **niveau conceptuel** permet de représenter le système opérant à l'aide d'un **modèle conceptuel des données** et d'un **modèle conceptuel des traitements**. Pour le premier modèle, le concepteur dispose de trois concepts : la classe d'entités, l'attribut et la relation. La représentation des traitements est basée sur les réseaux de Petri.

Au **niveau organisationnel** le concepteur élabore un **modèle organisationnel des traitements** et un **modèle logique des données**. Le premier permet notamment de répartir la charge de travail en fonction des ressources (homme ou machine). Le modèle logique des données est la traduction du modèle conceptuel des données dans l'un des modèles logiques des bases de données : hiérarchique, réseau ou relationnel.

Le **niveau opérationnel** est celui de la programmation du système d'information. On distingue, comme dans les autres étapes, deux modèles. Le **modèle opérationnel des traitements** est constitué par les programmes élaborés à partir du modèle organisationnel en appliquant les principes de la programmation structurée. Le **modèle physique des données** résulte d'une adaptation du modèle logique à un système de gestion de fichiers ou de bases de données particulier.

La méthode MERISE offre une démarche pour guider le concepteur pendant le développement du SI. Cette démarche prend en compte à la fois les aspects techniques et les aspects humains du développement du SI.

MERISE est adaptée à l'utilisation de SGBD conventionnels (hiérarchiques, réseaux ou relationnels) et préconise l'utilisation de la programmation structurée pour élaborer les programmes. Cependant le niveau conceptuel permet théoriquement d'établir les spécifications du système d'information indépendamment de la solution technique retenue aux niveaux organisationnel et opérationnel.

3.3.3 La méthode REMORA

La méthode REMORA a été développée au début des années 80 par une équipe dirigée par Colette Rolland [RFB88]. C'est une méthode systémique qui s'appuie sur trois niveaux d'abstraction. Le niveau conceptuel est centré sur l'élaboration d'un modèle de l'organisation. Le niveau logique est consacré à l'élaboration d'une solution technique et le niveau physique est celui de la programmation du système d'information. REMORA s'intéresse essentiellement à la dimension informatique du système d'information.

La spécification du système d'information établie au niveau conceptuel est décrite dans le **schéma conceptuel**. Pour élaborer ce schéma conceptuel, le concepteur dispose de trois concepts principaux : l'objet, l'événement et l'opération. Une **classe** d'objets est une relation (au sens relationnel) en troisième forme normale. Les **objets** sont donc des nuplets. Une **opération** est une séquence atomique d'actions (transaction). Une opération modifie un objet unique. Un **événement** représente un changement observé dans la valeur d'un objet unique. Un événement peut déclencher plusieurs opérations.

La conception proprement dite du système d'information permet d'élaborer le **schéma logique**. L'organisation des données est décrite à l'aide des concepts de type d'enregistrement et de lien proposé dans les modèles de données des SGBD réseaux. La représentation des traitements s'appuie sur la notion de transaction.

REMORA propose une démarche pour traduire le schéma conceptuel dans le formalisme du niveau logique. On notera la forme d'encapsulation introduite par l'association des opérations et des événements à une classe d'objets unique. Si la notion de relation, sous-jacente à celle de classe d'objets, n'explicite pas les associations entre nuplets, en revanche, la notation graphique proposée dans REMORA met en évidence ces associations entre objets.

Comme MERISE, REMORA est adaptée aux systèmes de gestion de bases de données transactionnels conventionnels (hiérarchiques, réseaux ou relationnels) mais ces deux méthodes continuent à évoluer pour s'adapter aux techniques orientées objets (O* [CR89], OOM [Roch91]).

3.3.4 La méthode IA (NIAM)

La méthode IA (Information Analysis) [Habr88] ou NIAM (Nijssen IA Method) a été mise au point par GM. Nijssen à la fin des années 70. Elle est centrée sur l'élaboration du schéma conceptuel spécifiant le contenu d'une base de données. La méthode MAIA complète IA en y intégrant les apports de MERISE pour la modélisation des traitements. Les concepts de la méthode NIAM sont présentés à l'aide du vocabulaire utilisé dans les sections précédentes ; le vocabulaire original est indiqué entre parenthèses.

NIAM permet de distinguer les types de base (LOT : Lexical Object Type) dont les éléments sont des valeurs et les classes d'objets (NOLOT : NOn Lexical Object Type) dont les éléments sont des représentations de classes d'objets du domaine d'application. Une classe peut être définie comme un sous-ensemble d'une autre classe. Entre les deux

catégories d'éléments, types et classes, le concepteur peut définir deux catégories de relations qui ont la particularité d'être binaires. La relation attribut (pont de dénomination) est définie entre une classe d'objets et un type de base. Une association (idée type) est définie entre deux classes d'objets. Des cardinalités fines peuvent être indiquées pour les attributs ou les associations.

L'apport principal de NIAM concerne l'expression de contraintes d'intégrité pour lesquelles le concepteur dispose de nombreuses contraintes prédéfinies et combinables :

- Les contraintes d'unicité (fonction monovaluée) ou de totalité (fonction totale) peuvent être définies entre une classe et un ensemble de classes et de types.
- Les contraintes d'égalité, d'exclusion ou d'inclusion peuvent être définies entre deux relations (attribut ou association) concernant une même classe.
- Les contraintes de totalité, d'exclusion ou de partition peuvent être définies entre une classe et ses «sous-classes».

4 Systèmes interactifs

Au début des années 80, la généralisation de l'outil informatique a été accélérée par le développement de la micro-informatique. Cette évolution a imposé une prise en compte plus précise des capacités, des connaissances et des souhaits des utilisateurs. A la même époque, l'apparition des systèmes de fenêtrages, la généralisation des écrans graphiques et l'augmentation de la puissance des ordinateurs ont permis le développement d'applications interactives plus conviviales que les anciennes applications.

4.1 Généralités

4.1.1 Structuration d'une application interactive

Dans une application interactive, la partie consacrée à la gestion de l'interaction entre l'utilisateur et l'application représente couramment 50% et parfois jusqu'à 80% du code total de l'application. La complexité de la gestion de l'interaction a nécessité très tôt le développement de modèles d'architecture pour guider le concepteur d'une application interactive. A ce titre, la décomposition d'une application interactive en deux composants : le corps et l'interface interactive, constitue un principe unanimement appliqué.

Le **corps** d'une application interactive ou **composant fonctionnel** regroupe les fonctions et les informations qui se rapportent au domaine d'application et qui sont in-

dépendantes des dispositifs d'interaction.

L'**interface interactive** réalise l'interface entre l'utilisateur et le corps de l'application. Elle regroupe les fonctions et les informations qui se rapportent à la mise en œuvre du dialogue et présente à l'utilisateur une **image** de l'application interactive.

Cette décomposition initiale constitue une première application du principe de modularité du génie logiciel. Elle permet d'améliorer l'évolutivité et la portabilité de l'application. En effet, la modification de l'image de l'application affecte seulement l'interface interactive mais n'influe pas sur le corps de l'application. Le portage sous un système de fenêtrage différent ou sur une autre plate-forme entraîne des modifications dans l'interface interactive mais n'affecte pas le corps de l'application.

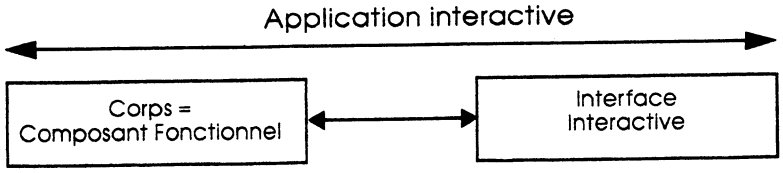


Figure 2-7 : Décomposition initiale d'une application interactive.

4.1.2 Principes ergonomiques

Nous évoquons brièvement quelques points importants pour la conception d'applications interactives [Cout90, Peto90].

Retour d'information et ***temps de réponse*** : L'utilisateur doit être tenu informé du déroulement et des résultats des actions qu'il a déclenchées. Ce retour d'information doit s'effectuer en temps réel ou avec un temps de réponse acceptable pour l'utilisateur. La **délégation sémantique** [Cout90] consiste à déplacer une partie de la connaissance portant sur le domaine d'application, du corps de l'application vers l'interface interactive. Le déplacement des contrôles de cohérence élémentaires par exemple permet de garantir la rapidité et la qualité du retour d'information. La délocalisation sémantique remet cependant partiellement en cause la séparation des rôles entre le corps et l'interface. Cette technique doit donc être utilisée avec modération car elle peut limiter l'adaptabilité et la portabilité et entraîner des redondances.

Multifils : Les programmes anciens limitent le dialogue entre l'utilisateur et le programme à une suite linéaire et figée par avance de questions posées par le programme à l'utilisateur. L'expérience montre que de tels dialogues ne sont pas adaptés au comportement opportuniste de l'utilisateur [Cout90]. Pour mener à bien une tâche informatique, l'utilisateur doit pouvoir utiliser le programme en suivant son cheminement intellectuel. Pour cela l'application interactive doit pouvoir gérer plusieurs fils de dialogue en parallèle. Elle doit laisser l'utilisateur libre de changer de fil et, dans chacun, de progresser, de revenir en arrière ou d'interrompre définitivement le dialogue.

Initiative du dialogue : On peut distinguer deux grandes classes d'utilisateurs : les utilisateurs expérimentés et les novices. La compétence des premiers doit leur permettre d'avoir l'initiative du dialogue avec l'application à l'exception éventuellement de certaines tâches délicates dont la conduite revient à l'application. En revanche, la découverte d'une application par des utilisateurs novices doit pouvoir être guidée par l'application qui dispose alors de l'initiative du dialogue. Cette solution permet notamment, d'éviter le déclenchement d'opérations inappropriées ou fatales.

Cohérence : On peut distinguer différents niveaux de cohérence [Cout90] : cohérence du déroulement d'une sous-tâche commune à différentes tâches, cohérence de la terminologie utilisée, cohérence de l'organisation spatiale de l'image présentée à l'utilisateur, cohérence des informations, etc. Si ces principes ergonomiques influent sur l'élaboration des spécifications de l'interface de l'application, la cohérence des informations influe particulièrement sur la charge de travail de l'interface interactive. L'interface doit en effet maintenir la cohérence des informations selon deux axes : la cohérence entre une information affichée à l'écran et cette même information présente dans le corps de l'application et la cohérence entre différents affichages d'une même information.

4.1.3 Niveaux d'abstraction

La nature des informations échangées entre l'interface interactive et le corps peut varier. Plus ces informations sont proches du matériel (clic souris, touche frappée, etc.) et plus leur niveau d'abstraction est dit **bas**. Plus elles sont proches des concepts manipulés par le corps de l'application (concepts du domaine) et plus leur niveau d'abstraction est dit **élevé**.

Lorsque le niveau d'abstraction est trop bas, l'interface interactive est réduite à un gestionnaire d'entrée/sortie. Le corps de l'application doit alors élaborer les concepts qu'il manipule à partir des informations de bas niveau communiquées par l'interface interactive. Le corps de l'application voit donc augmenter sa charge de travail et sa dépendance vis-à-vis des dispositifs physiques d'entrée/sortie.

Un niveau d'abstraction élevé a l'avantage de rendre le corps indépendant de l'interface interactive. La charge de travail du concepteur est alors réduite puisque le corps de l'application reçoit des informations exploitables directement. Comme l'interface interactive dispose de la connaissance nécessaire à l'élaboration d'informations proches des concepts de l'application, le retour d'information est également amélioré. Cependant un niveau d'abstraction élevé nécessite une délocalisation sémantique importante dont nous avons déjà souligné les inconvénients.

4.1.4 Localisation du contrôle

La notion de contrôle du dialogue [TB85, Cout90] désigne l'arbitrage de l'enchaînement des actions de l'utilisateur. Le contrôle du dialogue peut être localisé selon trois modes différents.

Le contrôle est **interne** si c'est le corps de l'application qui assure l'arbitrage. Dans ce cas le corps appelle les fonctions de dialogue réalisées par l'interface interactive. Cette solution, adoptée dans les applications anciennes, ne permet pas de donner aisément l'initiative du dialogue à l'utilisateur. Le dialogue, noyé dans l'application, est difficile à modifier. Dans le corps de l'application, les appels aux fonctions de l'application sont mélangés aux appels des fonctions du dialogue nuisant ainsi à la lisibilité du programme.

Le contrôle est **externe** si l'interface interactive dirige le dialogue et appelle les fonctions de l'application qui se comporte alors comme un serveur fonctionnel. Cette solution permet une mise œuvre aisée de dialogues dirigés par l'utilisateur. Le corps de l'application ne pouvant prendre le contrôle, l'interface interactive doit être capable de déterminer tous les paramètres nécessaires au déroulement complet d'une fonction de l'application. Or ce n'est pas toujours possible, par exemple lorsque le déroulement de traitements complexes peut dépendre d'un choix de l'utilisateur. Dans un tel cas de figure, la vérification sémantique de tous les paramètres avant l'appel au corps de l'application peut imposer une délocalisation sémantique très importante.

Lorsque l'arbitrage du dialogue est partagé entre le corps et l'interface interactive le contrôle est dit **mixte**. Bien utilisée, cette solution permet à l'utilisateur d'avoir l'initiative du dialogue tout en autorisant l'application à prendre temporairement le contrôle. Cette prise de contrôle doit selon le cas être limitée. Par exemple l'application peut guider ponctuellement l'utilisateur ou lui demander des données intermédiaires dans un traitement complexe. Cette solution permet en outre de prendre en compte les événements internes (horloge, courrier électronique, communication interapplications, etc.).

4.1.5 Décomposition de l'interface interactive

Comme nous l'avons présenté dans la section 4.1.1, le principe de décomposition d'une application interactive en deux composants, le corps et l'interface interactive, est unanimement appliqué. Un second principe, aujourd'hui admis, préconise d'organiser l'interface interactive en trois composants [Norm92] : l'adaptateur, le contrôleur du dialogue et l'interaction.

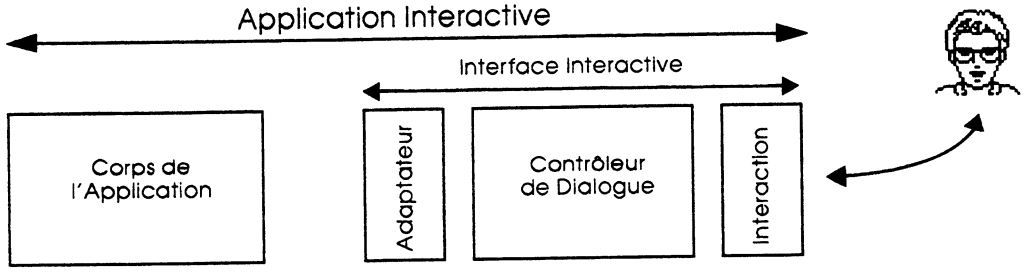


Figure 2-6 : Décomposition de l'interface interactive en trois composants.

L'**adaptateur** réalise l'interface entre le corps de l'application et le contrôleur de dialogue. Il dénote la vue qu'a le contrôleur du corps de l'application. L'adaptateur fournit des fonctions et des informations adaptées aux besoins du contrôleur de dialogue. Ces fonctions et ces informations se rapportent au domaine d'application et sont disponibles directement sur le corps ou éventuellement élaborées à partir de ces dernières.

Le composant **interaction** contient le code de mise en œuvre physique de l'interaction. Ce composant peut être un serveur graphique (ToolBox, X-Windows, NeXTStep, etc.) ou une interface bâtie au-dessus d'un serveur graphique.

Le **contrôleur du dialogue** assure la gestion du dialogue : réalisation des commandes requises par l'utilisateur, maintien de la cohérence, séquencement des opérations, etc.

Cette décomposition de l'interface a été introduite par le modèle de Seeheim [Pfaf85] cité dans [Cout90].

4.2 Outils pour la construction de systèmes interactifs

Différents outils d'aide à la construction d'interface ont été développés en s'appuyant généralement sur des langages existants. Nous évoquons trois catégories d'outils [Adre89, Cout90] : les boîtes à outils, les squelettes d'applications et les générateurs d'interfaces.

4.2.1 Boîtes à outils

Une **boîte à outils** est un ensemble de services permettant de gérer les entrées/sorties entre l'utilisateur et l'ordinateur. Les services d'une boîte à outils sont regroupés dans des bibliothèques de procédures, des modules dédiés et plus récemment dans des bibliothèques de classes spécialisées. Dans ce dernier cas, le programmeur peut réutiliser directement les classes spécialisées ou les enrichir pour les adapter à ses besoins.

La plupart des systèmes offrent des boîtes à outils et si le principe de base n'est pas nouveau, on peut remarquer que certaines réalisations sont plus que de simples bibliothèques. En effet on peut distinguer les boîtes à outils selon la localisation de la boucle principale de contrôle [Cout90]. Dans un contrôle "embarqué", toute entité du dialogue comporte un mécanisme de traitement d'événements et c'est la boîte à outils, de façon transparente pour le programmeur, qui distribue aux entités les événements en provenance de l'utilisateur. Par exemple le système X-window offre un mécanisme de callback par lequel une procédure est appelée directement par X lorsque l'événement qu'elle est chargée de traiter se produit. Dans un contrôle non embarqué la boucle d'acquisition des événements en provenance de l'utilisateur et de déclenchement des traitements correspondants est programmée explicitement dans l'application. C'est le cas par exemple pour la Toolbox du Macintosh, le programmeur doit concevoir la boucle chargée de recevoir les événements de l'utilisateur et de déclencher les actions correspondantes

dans la présentation.

Une boîte à outils offre des briques de base pour construire l'interface interactive mais laisse le programmeur totalement libre du choix de l'architecture du programme. Les services offerts par une boîte à outils sont de bas niveaux et, d'une application à l'autre, on retrouve souvent les mêmes séquences de code.

4.2.2 Squelettes d'applications

Un **squelette d'applications** est un programme générique, réutilisable et extensible qui regroupe le code correspondant à un ensemble de fonctions courantes d'une application interactive comme par exemple la boucle générale de contrôle. Un squelette d'applications factorise un ensemble de connaissances concernant notamment le fonctionnement de la boîte à outils sur laquelle il s'appuie et permet ainsi de réduire la charge de travail du programmeur. La tâche de ce dernier consiste à fixer les paramètres du squelette, à en adapter voire à en redéfinir certaines parties et à développer de nouvelles parties spécifiques à son programme.

Le rôle central joué par la réutilisation et la spécialisation dans la technique des squelettes d'applications justifie l'utilisation croissante des techniques à objets. MacApp développé pour le Macintosh en Pascal Objet est, à ce titre, un exemple typique. On notera que la spécialisation d'un squelette nécessite souvent la connaissance et le recours à la boîte à outils sur laquelle il est bâti.

4.2.3 Générateurs d'interfaces

Un **générateur d'interfaces** est un programme capable de produire, à partir de la spécification de l'interface d'une application interactive, le code chargé de mettre en œuvre cette interface. Alors qu'un squelette d'applications nécessite souvent le recours à la boîte à outils sous-jacente, un générateur d'interfaces permet, en théorie, de s'en affranchir. Un générateur d'interfaces est généralement constitué d'un éditeur et d'un compilateur chargés de produire le code de l'interface interactive, d'un noyau d'exécution et d'un mécanisme de liaison pour faire coopérer le corps de l'application, l'interface interactive et le noyau d'exécution.

La spécification de l'interface décrit l'organisation de l'écran présenté à l'utilisateur, l'enchaînement des dialogues ainsi que les liens entre les actions de l'utilisateur et les fonctions du corps de l'application. La spécification peut être textuelle ou exprimée dans un formalisme tel que les réseaux de transitions. La tendance actuelle est à la spécification interactive et visuelle des interfaces par assemblage de composants de base : menus, boutons, champs, fenêtres, etc.

4.3 Différents modèles d'architecture

Dans le domaine des interfaces homme-machine, le but d'un modèle d'architecture est de définir une structure générique à partir de laquelle le concepteur peut construire une application interactive particulière. Cette structure identifie l'agencement et la fonction des principaux composants ainsi que les transferts de données entre ces composants. Nous présentons brièvement deux familles de modèles d'architecture avant d'approfondir l'approche multiagent. Pour un exposé plus complet, on se reportera à l'ouvrage de J. Coutaz [Cout90].

4.3.1 Modèle entrée/sortie

Le **modèle entrée/sortie** [Cout90] organise l'interface interactive soit comme un empilement de machines abstraites soit comme un ensemble non hiérarchisé de machines abstraites. Le découpage en couches ou en modules reflète un découpage fonctionnel de l'interaction en différents niveaux d'abstraction. Le plus élémentaire est constitué par les actions physiques opérées sur les dispositifs physiques d'entrée/sortie et le plus élevé correspond aux concepts manipulés dans le corps de l'application.

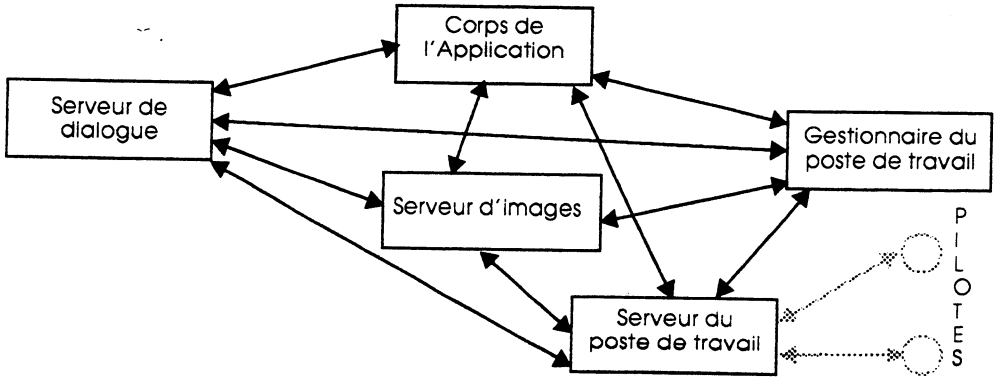


Figure 2-7 : Le modèle entrée/sortie dans une approche non hiérarchisée.

Le modèle entrée/sortie a l'avantage de proposer une architecture modulaire mais ne propose pas de méthode pour organiser le composant chargé du dialogue proprement dit.

4.3.2 Approche multiagent

Les différents modèles relevant de l'approche **multiagent** décomposent l'interface interactive en un ensemble d'agents. Chaque agent est spécialisé dans une tâche et il est capable de réagir à certaines sollicitations provenant de l'utilisateur, du corps de l'application ou des autres agents.

Cette approche permet une organisation fortement modulaire en différents agents susceptibles d'exécuter des traitements en parallèle et elle est bien adaptée à la gestion

de dialogues «multifils» pour lesquels un agent peut être affecté à chaque fil.

Le principe de décomposition de l'interface interactive en un ensemble d'agents est est à rapprocher des principes des modèles à objets et une conception en terme d'agents peut aisément être programmée à l'aide d'un langage à objets [Cout90].

4.4 Différents modèles multiagents

L'approche multiagents peut être considérée comme le troisième principe reconnu dans le domaine des interfaces homme-machine [Norm92] après la séparation initiale entre le corps d'une application et son interface interactive puis la décomposition de l'interface en trois composants décrite dans la section 4.1.5. Nous présentons brièvement trois modèles.

4.4.1 Modèle GWUIMS

Le système GWUIMS (George Washington User Interface Management System) [SHB86] est un système orienté objet qui propose d'organiser une application interactive en cinq classes d'objets.

Les **objets-A** (Abstraction) structurent le corps de l'application. Les **objets-T** ("Typing") traitent les événements d'entrée de bas niveau. Les **objets-G** (Graphique) traitent les événements de sortie de bas niveau. Les **objets-R** (Représentation) reçoivent des informations d'entrée des objets-T et contrôlent les informations de sortie des objets-G. Les fonctions réalisées par les objets-T, les objets-G et les objets-R composent les fonctions du composant interaction. Les **objets-I** (Interaction) font le relais entre les objets-A composant le corps de l'application, et les objets-R qui structurent la présentation. Les objets-I réalisent le contrôle du dialogue.

On peut remarquer que la séparation des entrées et des sorties dans deux catégories distinctes d'objets ne facilite pas le retour d'information.

4.4.2 Modèle PAC

Le **modèle PAC** [Cout90], développé à Grenoble, est un modèle indépendant d'un langage ou d'un système particulier. Le modèle PAC structure une application interactive à l'aide d'un concept unique : l'agent PAC.

Un **agent PAC** est composé de trois éléments distincts : la partie présentation, la partie abstraction et la partie contrôle. La partie **présentation** définit le comportement de l'agent en entrée et en sortie. Cette partie représente la partie perceptible (visible à l'écran) d'un agent PAC et contient par exemple les informations relatives au choix de représentation des informations ou la police de caractères retenue. La partie **abstraction** constitue une représentation abstraite des éléments de la présentation indépendante des choix de présentation. La partie **contrôle** maintient la cohérence entre la présentation et l'abstraction ainsi que les communications avec les autres agents PAC. Un agent PAC peut être programmé de différentes façons : une classe par agent, un mo-

dule par agent, trois classes par agent, etc.

Lorsqu'un agent PAC est trop compliqué, il peut être décomposé en plusieurs agents PAC fils. Cette relation de composition confère au système une structure arborescente. La communication entre agents PAC doit se faire selon les liens de composition. La racine de l'arborescence est un agent PAC dont l'abstraction représente le corps de l'application et dont la présentation contient les fonctions des gestionnaires physiques d'entrée/sortie.

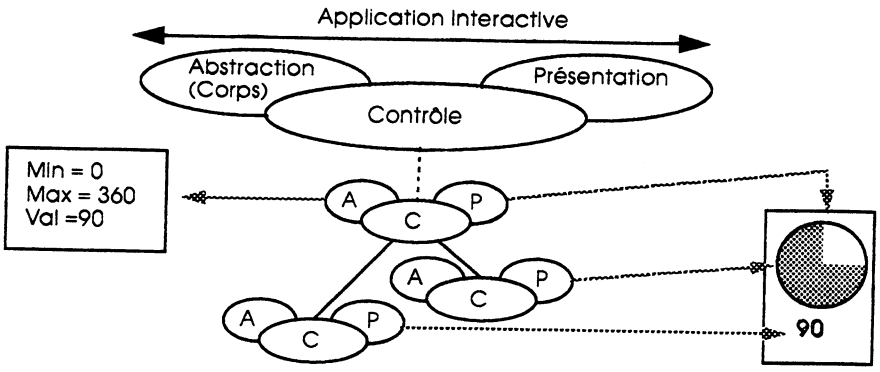


Figure 2-8 : Exemple d'architecture PAC [Cout90].

L'utilisation du modèle PAC a conduit à l'évolution représentée dans la figure 2-9 [NC91]. Le composant adaptateur, qui n'apparaît pas explicitement dans le modèle PAC, est mis en évidence. Le composant technique de présentation définit un terminal abstrait ou une boîte à outils virtuelle permettant d'assurer l'indépendance entre le contrôle du dialogue et les dispositifs physiques d'entrée/sortie.

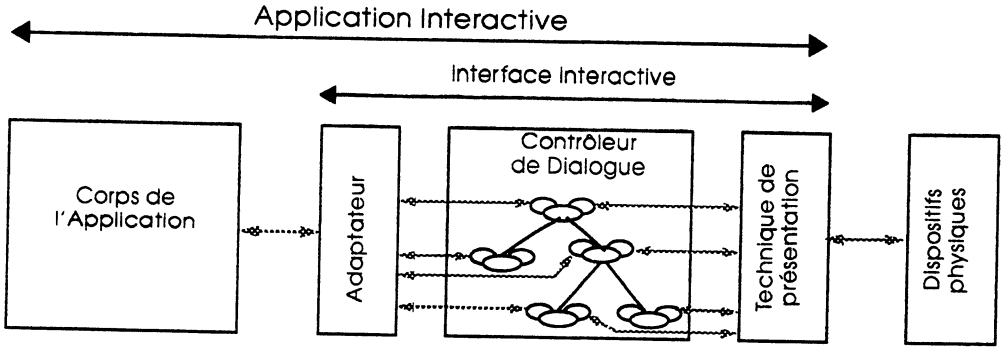


Figure 2-9 : Evolution du modèle d'architecture PAC.

La stratégie de décomposition de l'interface en agents PAC est souvent guidée par l'organisation des entités présentées sur l'écran de l'utilisateur. Comme ces entités sont souvent emboîtées hiérarchiquement, l'organisation de l'écran peut être représentée à

l'aide d'une arborescence d'agents PAC. Par ailleurs, la mise en œuvre de ce modèle d'architecture dans différentes applications a permis d'identifier des agents PAC généraux [CN89]. Ces agents PAC sont généralement issus de la contrainte imposée aux communications entre agents de devoir suivre la hiérarchie des liens de composition. Si deux agents PAC ont une relation alors un troisième agent est nécessaire pour gérer cette relation [NC92] :

- Lorsqu'une même information est présentée simultanément par plusieurs agents PAC, un autre agent PAC est nécessaire pour maintenir la cohérence entre ces différentes présentations. C'est notamment le cas des vues multiples.
- Lorsqu'une interaction est distribuée sur plusieurs agents PAC, un autre agent PAC est nécessaire pour assurer le bon déroulement de cette action.
- Si les présentations des agents PAC sont liées par des contraintes géométriques, par exemple, non recouvrement des présentations, alors un nouvel agent PAC doit être affecté à ce contrôle.
- Les erreurs doivent être centralisées et gérées par un agent PAC spécialisé, éventuellement décomposé en une hiérarchie d'agents PAC.

4.4.3 Modèle SIROCO

Le **modèle SIROCO** [Norm92] a été développé dans le cadre du projet GUIDE [KMR*88]. GUIDE (Grenoble Universities Integrated Distributed Environment) est un environnement de développement basé sur les techniques à objets qui offre un système d'exploitation réparti et un langage de programmation à objets.

Dans le modèle SIROCO, les objets permettant de structurer le corps de l'application sont dit **internes**. Ce modèle propose quatre catégories d'objets pour organiser l'interface interactive : objets montrables, objets vues, objets interactions et objets contrôles.

L'adaptateur est organisé en un ensemble d'objets **montrables**. Les informations affichées à l'écran sont celles contenues dans les objets montrables. Les objets **vues** décrivent l'image du système. Ils reçoivent les entrées de l'utilisateur et, en sortie, présentent les informations qui leurs sont communiquées par les objets contrôles. Les objets **interactions** sont les objets fournis par les boîtes à outils des systèmes graphiques. Les objets interactions sont utilisés par les objets vues pour réaliser l'interaction avec l'utilisateur. La gestion proprement dite du dialogue est réalisée par les objets **contrôles**. Ces objets font le lien entre les objets montrables et les objets vues.

Les objets montrables peuvent être partagés par plusieurs objets contrôles mais chaque objet vue est propre à un objet contrôle et chaque objet interaction est propre à un objet vue. Ces différents objets sont groupés en unités logiques qui représentent les **agents** du système. A chaque objet contrôle est associé un agent qui contient les différentes vues et les différents objets montrables associés à l'objet contrôle. L'agent contient également les différents objets interactions associés à chaque vue de l'agent.

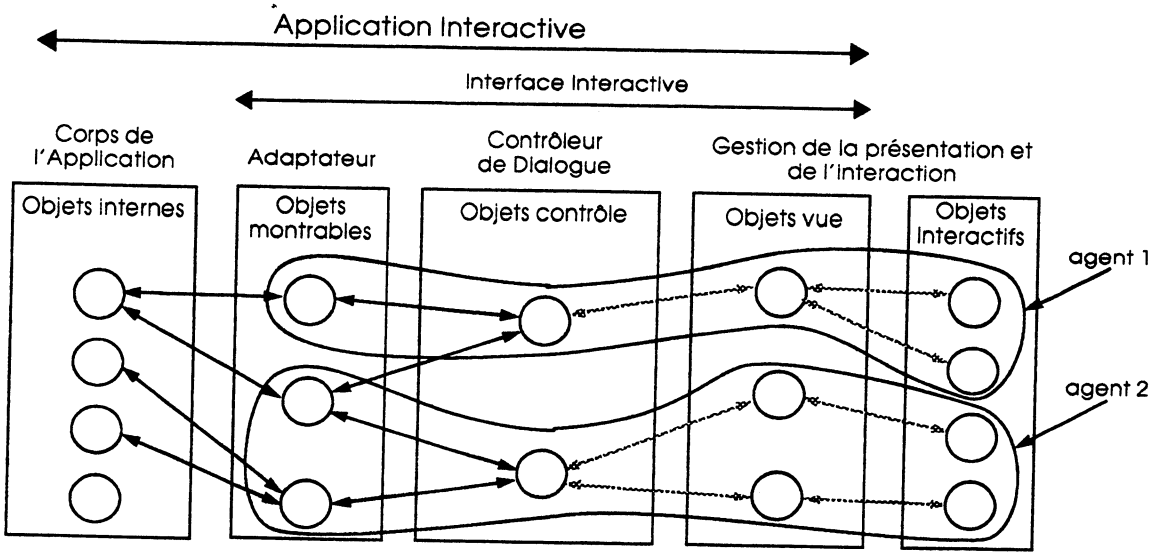


Figure 2-10 : Cinq catégories d'objets du modèle SIROCO et deux agents.

5 Bilan

Nous résumons les principaux apports des domaines de la programmation, des bases de données et des systèmes interactifs pour les étapes de spécification, de conception et de réalisation.

5.1 Etape de spécification

Les techniques de spécification n'entrent pas dans le cadre de notre étude, cependant nous pouvons les évoquer brièvement.

Dans le domaine de la programmation, différentes techniques de spécifications formelles ont été proposées comme par exemple : les approches algébriques [Gutt76, LZ75] ou les approches ensemblistes (Z [Abri78, Spiv89], VDM [Jone86]).

La spécification d'une base de données est réalisée pendant l'étape conceptuelle. Différents modèles ont été proposés pour décrire le contenu d'une base de données indépendamment d'une technique de réalisation particulière. Ces modèles se situent généralement dans le courant des modèles sémantiques comme par exemple : le modèle conceptuel de données de la méthode MERISE [TRC85], le modèle Entité/Relation [Chen76] ou le modèle sémantique SDM [HM83].

Dans le domaine des systèmes interactifs, la spécification de l'interface doit adresser à la fois la présentation et le dialogue. Si la présentation statique peut être aisément décrite en revanche la spécification du dialogue est plus délicate. Différentes

approches ont été utilisées comme les diagrammes de transition d'états, les réseaux de Petri ou les modèles à événements [Cout90].

5.2 Etape de conception

L'étape de conception permet d'élaborer une solution technique aux besoins exprimés pendant la phase de spécification.

5.2.1 Conception des programmes

L'élaboration des programmes s'appuie implicitement sur un modèle d'architecture qui distingue les programmes, des données persistantes gérées par un logiciel spécialisé tel qu'un système de gestion de fichiers ou de bases de données.

L'élaboration d'une architecture capable de réaliser les fonctions du programme, doit satisfaire différents objectifs comme : favoriser la décomposition modulaire du programme, rechercher la généralité pour chaque module, minimiser les dépendances fonctionnelles entre les modules (couplage faible), maximiser la cohésion de chaque module. Le graphe des dépendances entre les composants d'un programme exprime l'organisation fonctionnelle du programme.

Différentes techniques de décomposition ont été proposées. L'approche de conception dirigée par les objets, à l'origine de nombreuses méthodes récentes, est maintenant reconnue comme un moyen d'obtenir une organisation compréhensible et maintenable du programme. On peut remarquer que les règles méthodologiques et les nouveaux concepts utiles pour la conception ont été progressivement introduits dans les langages de programmation donnant ainsi naissance à des langages de plus en plus puissants.

5.2.2 Conception des bases de données

L'élaboration d'une base de données s'appuie implicitement sur un modèle d'architecture séparant la base de données et les programmes. La base de données est décrite et gérée à l'aide d'un SGBD et les programmes sont réalisés à l'aide d'un langage de programmation. Les SGBDOO actuels tendent cependant à remettre en cause cette séparation.

La conception d'une base de données commence par l'élaboration d'un schéma conceptuel des données. Ce schéma conceptuel est un modèle du domaine de l'application dont l'élaboration doit satisfaire l'exhaustivité vis-à-vis des besoins exprimés. Puis le modèle logique de données est dérivé du modèle conceptuel ; il doit limiter les redondances et les structurations inadéquates du point de vue du fonctionnement du type de SGBD choisi.

On peut noter que l'augmentation constante de la richesse des modèles de données offerts par les SGBD diminue la distance entre la perception du monde réel exprimée par le concepteur dans le schéma conceptuel et les représentations permises par les SGBD.

5.2.3 Conception des systèmes interactifs

La complexité des systèmes interactifs a conduit à l'élaboration de différents modèles d'architectures. Un consensus se dégage autour du principe fondammental suivant : un système interactif est décomposé en un corps (composant fonctionnel) et une interface interactive.

La conception d'une interface interactive doit permettre de mettre en œuvre les spécifications externes de l'application. L'efficacité et le confort de l'utilisateur sont conditionnés par la cohérence de l'affichage, la qualité et la rapidité du retour d'information, la facilité de mener différents fils de dialogues simultanément et la possibilité d'avoir l'initiative du dialogue avec l'ordinateur.

Les techniques de conception proposées actuellement convergent sur trois points. Une application est composée d'un corps et d'une interface interactive. Cette dernière comporte trois parties : l'adaptateur, le gestionnaire de dialogue et le composant chargé de gérer physiquement l'interaction. L'interface interactive est organisée en agents chargés de gérer une partie du dialogue ; leur composition varie selon les modèles.

5.2.4 Synthèse

L'étude des trois domaines choisis fait apparaître une architecture générale pour organiser un ensemble d'applications interactives. Nous l'avons résumée dans les figures 2-11 et 2-12.

Un système d'applications est conçu en combinant deux catégories de constituants : la **base de données** est chargée de stocker et de gérer les données persistantes et les **applications** (interactives) sont chargées de réaliser un ensemble de fonctions (cf. figure 2-11). La base de données se comporte alors comme un **serveur de données** pour l'application. Cette décomposition initiale des applications est accentuée actuellement par la généralisation du modèle client/serveur dans les systèmes de gestion de bases de données.

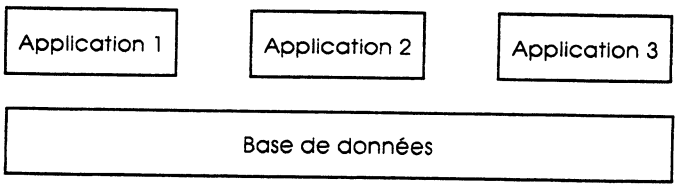


Figure 2-11 : Architecture générale d'un système applications.

Une application interactive est de la même manière décomposée en deux parties principales :

- Le **composant fonctionnel** contient les fonctions et les informations spécifiques au domaine d'application.

- L'**interface interactive** est chargée de la présentation des fonctions et des informations à l'utilisateur.

Dans cette décomposition, le corps de l'application se comporte comme un **serveur fonctionnel** pour l'interface interactive. L'interface interactive d'une application est décomposée en trois parties :

- Le **gestionnaire de dialogue** est chargé de réaliser les commandes de l'utilisateur.
- L'**adaptateur** permet d'assurer l'indépendance entre le gestionnaire de dialogue et le corps de l'application.
- Le composant d'**interaction physique** représente le logiciel de base chargé de gérer physiquement l'interaction avec l'utilisateur.

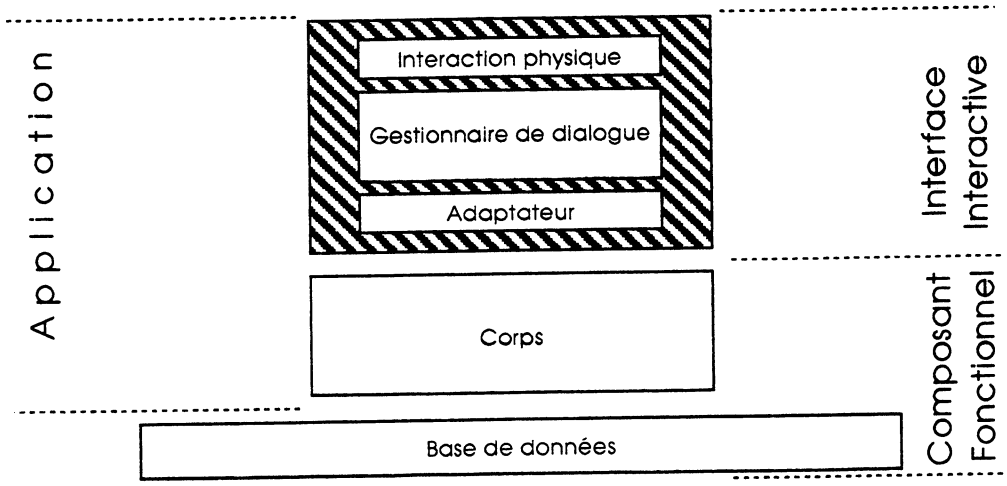


Figure 2-12 : Architecture générale d'une application interactive.

La décomposition d'une application en composants horizontaux spécialisés doit être complétée par un «découpage» de chaque composant ou de plusieurs composants simultanément en «tranches verticales» dont la composition varie selon le modèle d'architecture retenu puis selon les technologies utilisées. Dans les systèmes actuels, la base de données et le corps d'une application, sont décomposés en objets. Le macro-composant interface interactive est décomposé en agents dont la composition varie : dans le modèle PAC chaque agent est composé d'une abstraction, d'une présentation et d'un contrôle alors que le modèle SIROCO préconise un découpage vertical de toutes les couches de l'interface interactive pour obtenir des agents constitués de quatre catégories d'objets : les objets montrables, les objets contrôles, les objets vues et les objets interactifs.

5.3 Etape de réalisation

En ce qui concerne les techniques utilisées pour programmer une application lors de la phase de réalisation, on constate une nette évolution en faveur des techniques à objets.

Si le premier langage de programmation à objets, Simula, est déjà ancien, c'est surtout dans la seconde moitié des années 80 que ces langages se sont développés et qu'ils ont été de plus en plus utilisés.

Les systèmes de gestion de bases de données à objets font l'objet de nombreux travaux de recherches et de nombreuses annonces commerciales permettant de penser que les premières réalisations vont se développer en ce début de décennie 90.

Enfin dans le domaine des outils d'aide à la construction d'interfaces interactives, nous avons souligné l'utilité des techniques à objets et le nombre croissant de bibliothèques de classes spécialisées ou de générateurs d'interfaces permettant une approche par les objets.

6 Conclusion

- Comment déterminer et organiser les classes ?
- Comment structurer les données ?
- Comment concevoir les traitements ?
- Comment gérer l'interaction ?
- Quel rôle attribuer aux classes ?

Telles sont quelques unes des questions qu'un concepteur peut se poser pour réaliser un système d'applications à l'aide d'un langage de programmation de bases d'objets. Les éléments présentés dans ce chapitre n'apportent que des réponses partielles à ces questions :

- Les méthodes de conception de logiciels s'intéressent à l'élaboration d'architectures logicielles. Par définition, ces architectures sont générales et ne proposent pas de solutions particulières pour bâtir des systèmes d'applications. Ces méthodes sont axées sur la conception des programmes et n'étudient pas spécialement la gestion des données persistantes ou l'interaction. Cependant, si les méthodes de conception «non orientées objets» semblent dépassées, les méthodes de conception orientée objets constituent un point de départ intéressant pour définir une méthode adaptée à la conception de systèmes d'applications.
- Les méthodes conventionnelles d'analyse et de conception de systèmes d'information (MACSI) sont inadaptées à notre contexte. En effet, certaines approches utilisées pour spécifier le système d'information sont inadéquates pour des réalisations avec des langages de programmation de bases d'objets. Cette

inadéquation apparaît surtout pendant la phase de conception, comme par exemple la dichotomie données-traitements de MERISE. En effet la plupart des MACSI assistent le concepteur pour élaborer une solution à l'aide de modèles de données conventionnels (hiérarchique, réseau ou relationnel) et de la programmation structurée. Les MACSI n'apportent donc pas une réponse complète pour élaborer une solution technique adaptée à un langage de programmation de bases d'objets.

- Les méthodes de conception de systèmes interactifs proposent des modèles d'architecture utilisables généralement quel que soit le système cible. Cependant, si ces modèles d'architecture sont particulièrement utiles pour concevoir l'interface interactive, ils restent insuffisants pour guider le concepteur dans l'élaboration du corps de l'application ou de la base de données.

Ces trois domaines (programmation, bases de données et interfaces homme-machine) ont déjà fait l'objet de premiers travaux intégrateurs, nous en résumons trois.

Le travail d'I. Petoud [Peto90] se situe dans le cadre du développement de systèmes d'information dans les entreprises et s'appuie sur la méthode IDA [Boda86]. Ses propositions se limitent au cas des applications de gestion fortement interactives et concernent plus particulièrement la phase de saisie des informations. Son objectif est de proposer un ensemble d'outils informatiques pour aider le concepteur d'une telle application à réaliser l'interface interactive. L'idée générale est de produire automatiquement une première version de l'interface à partir des spécifications fonctionnelles obtenues à l'aide de la méthode IDA . Cette première version peut être ensuite adaptée et enrichie pour obtenir l'interface définitive. L'architecture de l'interface générée adhère aux principes du modèle de Seeheim et laisse, autant que possible et dans le respect des spécifications fonctionnelles, l'initiative du dialogue à l'utilisateur. On notera que si ce travail s'intéresse essentiellement à l'interface, il complète la méthode IDA.

Les outils FAKIR [Coll87] et FO2 [CB91] de C. Collet sont destinés au domaine de la bureautique. Ces outils concrétisent un travail de recherche mené sur la gestion interactive de formulaires. Le formulaire est considéré comme le support du dialogue entre l'utilisateur et l'ordinateur et comme un modèle d'organisation des données, des contraintes sur ces données et des opérations. Une application est alors vue comme un ensemble de formulaires qui constituent à la fois l'interface de l'application et un support pour la représentation des données et des opérations. Le gestionnaire FO2 a été conçu en s'appuyant sur le modèle d'architecture PAC et a été programmé à l'aide du SGBDOO O_2 [ODeu89].

F. Adreit [Adre89] s'est intéressée à la conception d'une interface entre l'utilisateur et une base de données et plus particulièrement à la description logique de l'interaction ainsi qu'à la réutilisation de concepts généraux dans différentes interfaces. F. Adreit propose de structurer l'interface interactive en trois composants : l'organisation qui adapte les données de la base aux besoins d'une utilisation particulière, l'interaction qui affiche les données à l'écran et gère l'interaction et enfin la correspondance qui fait le

lien entre les deux autres composants.

Ces travaux intégrateurs s'appuient sur des technologies anciennes ou s'attachent à proposer une interface pour une base de données. Dans un cas comme dans l'autre, ils n'apportent pas de propositions complètes utilisables dans notre contexte.

Pour évaluer l'intérêt que peuvent présenter les méthodes de conception orientées objets pour la réalisation de systèmes d'applications à l'aide d'un langage de programmation de bases d'objets, nous étudions plus en détails, dans le chapitre suivant, les techniques à objets, en mettant en évidence différentes interprétations de certains concepts en fonction de l'approche adoptée : programmation ou base de données.

Chapitre 3

Techniques à objets et conception dirigée par les objets

1 Introduction

L'«orientation objet» constitue une évolution majeure pour la conception et la réalisation des systèmes informatiques. Cette orientation est issue des langages de programmation et elle est reprise dans la plupart des domaines de l'informatique, notamment en bases de données et dans les systèmes interactifs. Les auteurs s'accordent pour identifier différents concepts fondamentaux dans les techniques à objets et nous retenons plus particulièrement les concepts présents dans les langages de classes [MNC*89] : le concept d'**objet**, encapsulant des données et des traitements, protégé par une **signature** et décrit par une **classe**. Nous retenons également quatre relations : la **référence** et l'envoi de **messages** entre objets, l'**instanciation** entre objets et classes et la relation d'**héritage** entre classes [Wegn87, BGHS91, KA90]. Ces concepts sont intégrés dans de nombreux logiciels de base récents mais ils sont interprétés différemment selon le point de vue adopté. Dans ce chapitre nous explicitons les concepts principaux des techniques à objets en présentant, lorsque c'est possible, plusieurs interprétations de ces concepts. Il nous semble en effet nécessaire de bien identifier ces interprétations avant de proposer des concepts plus riches que les notions habituelles d'objet, de classe et d'héritage.

Dans le domaine des méthodes de conception de logiciels, la conception dirigée par les objets combine quatre techniques d'élaboration des programmes : l'opération de réification, le principe de localité, le développement par composition et affinage et le concept de généricité [Ferb90]. Les programmes conçus selon cette technique sont reconnus comme plus stables, plus compréhensibles et plus aisés à faire évoluer. Nous présentons dans ce chapitre quelques méthodes orientées objets disponibles actuellement en étudiant notamment leur adéquation à la conception de systèmes d'applications.

L'étude critique des techniques à objets et la présentation de la conception dirigée par les objets permettent de présenter, dans les chapitres 4 et 5, la méthode MOSAÏC qui offre un ensemble de concepts supplémentaires adaptés à la description des données persistantes, à la conception des fonctions spécifiques et à l'élaboration du gestionnaire de dialogue.

2 Techniques à objets

2.1 Introduction

Nous rappelons brièvement les principaux concepts des techniques à objets tels qu'ils sont proposés dans les langages de classes [MNC*89].

Un **objet** regroupe des données, accessibles par des **attributs**, et des méthodes qui réalisent des opérations sur ces données. La **signature** d'un objet regroupe les méthodes publiques et les attributs publics de cet objet. Une **classe** décrit un modèle d'objets et on parlera des attributs, des méthodes et de la signature d'une classe. Tout objet est obtenu par **instanciation** d'une classe.

Chaque objet est repéré par un **identificateur** : l'oid (object identifier). Cet identificateur permet de toujours distinguer deux objets même s'ils sont instances d'une même classe et si leurs données sont identiques. Un objet est implicitement manipulé par l'intermédiaire de son identificateur, mais celui-ci n'est pas directement visible ni modifiable par l'utilisateur.

La notion de **référence** permet de définir un lien orienté d'un objet vers un autre objet. De façon simple, on peut dire qu'un premier objet O1 référence un second objet O2, lorsqu'un des attributs de O1 contient l'identificateur de O2. Bien que les références existent uniquement entre des objets, par abus de langage, on dit qu'une classe C réfère une classe C' lorsqu'un des attributs de C est de type C'.

L'**héritage** est un mécanisme qui permet de définir une nouvelle classe à partir d'une classe existante. Lorsque la classe C' hérite de la classe C, C' dispose par défaut des attributs et des méthodes de C.

Pendant l'exécution d'un programme développé à l'aide d'un langage à objets, les objets communiquent par des **messages**. L'envoi de message est une forme souple d'appel procédural qui provoque le déclenchement d'une méthode ou l'accès à un attribut.

2.2 Objets et classes

2.2.1 Objet = élément d'un type abstrait de données

Dans la théorie des types abstraits, un type de données est défini de façon déclarative par l'ensemble des opérations applicables aux éléments de ce type et par les propriétés de ces opérations. La notion de classe est souvent rapprochée de celle de «type abstrait de données» [GH78, Gutt77, Meye88, Moit82]. Plus exactement, une classe est considérée comme un moyen de programmer un type abstrait. Un objet est alors considéré comme un élément d'un type abstrait.

```
type abstrait T_ARC        importe Entier, Booléen
  opération
     arc : Entier × Entier × Entier × Entier → T_ARC
```

```
      origine : T_ARC → Entier × Entier
      inverse : T_ARC → T_ARC
      égal : T_ARC × T_ARC → Booléen
      plus : T_ARC × T_ARC → T_ARC
   précondition...
   axiome
      origine(arc(a,b,c,d) ) = (a,b)
      inverse(inverse(x) ) = x
      plus(a,b) = plus(b,a)
fin type abstrait T_ARC

(*le type abstrait T_ARC peut être programmé par la classe C_ARC*)

classe C_ARC
   signature     (*méthodes publiques*)
      init_arc (x1, y1, x2, y2 : Entier),
      origine : nuplet (   x1 : Entier
                           y1 : Entier),
      inverse,
      égal (autre_arc : C_ARC) : Booléen,
      plus (autre_arc : C_ARC) : C_ARC,        (*résultat := self + autre_arc*)
      ajoute (autre_arc : C_ARC)               (*self := self + autre_arc*)
   corps...
fin classe ARC
```

Exemple 3-1 : Type abstrait de données `T_ARC` et classe d'objets `C_ARC`.

Les opérations d'un type abstrait de données sont définies sur un domaine de valeurs alors qu'une méthode est définie sur un objet unique. On peut remarquer que la programmation d'un type abstrait à l'aide d'une classe conduit à transformer la signature des opérations algébriques (+, –, *) et plus généralement des opérations dont les arguments ont des rôles symétriques. La programmation d'une telle opération à l'aide d'une méthode nécessite d'attribuer la responsabilité de l'opération à l'un des arguments de cette opération. Appliquer l'opération revient alors à envoyer à cet objet un message dans lequel les autres arguments sont passés en paramètres.

Dans l'exemple 3-1, les opérations binaires `égal` et `plus` du type abstrait `T_ARC` ont été programmées, dans la classe `C_ARC`, par les méthodes `égal` et `plus` à un seul argument explicite. On notera par ailleurs que l'opération `plus` de `T_ARC` peut être programmée de deux façons différentes dans `C_ARC` (`plus` ou `ajoute`). De ce point de vue, la programmation d'un type abstrait à l'aide du concept de module proposé par les langages modulaires conduit à une représentation plus conforme à la spécification du type abstrait comme le montre l'exemple 3-2.

```
module TYPE_ABSTRAIT_ARC
   exporte
      type M_ARC,
           POINT : article (x, y : Entier) fin_article,
      procédure
         arc : (x1, y1, x2, y2 : Entier) : M_ARC,
         origine (arc : M_ARC) : POINT,
         inverse (arc : M_ARC) : M_ARC,
         égal (arc1, arc2 : M_ARC) : Booléen,
         plus (arc1, arc2 : M_ARC) : M_ARC,
   corps
```

```
      type M_ARC = article origine, destination : POINT fin_article,
      procédure arc : (x1, y1, x2, y2 : Entier) : M_ARC...
fin module TYPE_ABSTRAIT_ARC
```

Exemple 3-2 : Module TYPE_ABSTRAIT_ARC.

2.2.2 Objet = constituant d'une base d'objets persistants

Dans une approche de nature base de données, l'objet est perçu comme un groupement de données persistantes. Ces données peuvent être atomiques (valeurs simples) ou complexes (valeurs construites à l'aide de constructeurs). Les modèles de données conventionnels permettent de décrire et de manipuler des ensembles d'entités. Chaque ensemble est nommé et il est caractérisé par une «entité type». Les noms des ensembles constituent généralement des points d'entrée de la base de données et donnent accès aux ensembles d'entités. Dans certains systèmes à objets, la notion de classe est indépendante de l'ensemble des objets générés à partir de cette classe. Par conséquence, les propriétés liées à l'ensemble des objets, par exemple la cardinalité de l'ensemble ou la définition de clés, ne peuvent plus être définies au niveau de la classe. Par ailleurs, dans de tels systèmes, les noms de classes ne peuvent pas constituer les points d'entrée dans la base des objets persistants. Dans O2 par exemple, les points d'entrée sont constitués par des «objets nommés» accessibles directement par leur noms et visibles de tout objet.

```
classe Personne
   signature
      nom : Chaîne,
      prénom : Chaîne,
      âge (année_courante : Entier) : Entier,
      initialiser (nom, prénom, date_naissance : Chaîne),
      imprimer
   corps
      attributs date_naissance : Chaîne,
      méthode âge (année_courante : Entier) : Entier...
      méthode imprimer...
fin classe Personne
```

Exemple 3-3 : Classe Personne.

Lorsque la notion de classe est considérée seulement comme un nouveau constructeur de types de données, des mécanismes doivent être proposés pour structurer les programmes. Ainsi ObjectPascal [Tesl85] étend un langage procédural et permet de faire cohabiter dans un programme des classes et des procédures. Dans Modula-3 [Nels91] les classes cohabitent avec des modules comme dans les versions 1 et 2 du modèle de base du générateur Aristote [ABD*90, DD91] qui introduit des A_types (classes) et des A_opérations (procédures) regroupés dans des A_schémas qui constituent des formes de bibliothèques de A-procédures.

2.2.3 Objet = composant modulaire d'un programme

Un objet peut être perçu comme un composant logiciel qui regroupe, comme les modules des langages de programmation, des données et des traitements. Conceptuellement, l'objet peut être vu comme un module qui communique avec d'autres modules [Cohe84].

La classe est un modèle d'objets et constitue à ce titre le premier mécanisme de factorisation de code entre plusieurs objets [BBL91, Call91]. Une classe peut donc être vue comme l'ajout d'un mécanisme d'instanciation à la notion de module. Cependant la signature d'une classe est moins riche que celle d'un module. Dans les langages modulaires, l'indépendance entre le typage des données et la modularité permet à un module d'exporter plusieurs types de données. Dans les systèmes à objets, le typage est lié à la modularité : une classe représente une famille de modules ou d'objets de même type.

```
module Pile_entier
   exporte
      procédure
         initialiser,
         empiler (élément : Entier),
         dépiler : Entier,
         pile_vide : Booléen,
   corps
      variable
         pile : table [1..50] Entier,
         sommet : 0..50...
fin module Pile_entier
```

```
classe Pile_entier
   signature
      initialiser,
      empiler (élément : Entier),
      dépiler : Entier,
      pile_vide : Booléen,
   corps
      attribut
         pile : table [1..50] Entier,
         sommet : 0..50...
fin classe Pile_entier
```

Exemple 3-4 : Module et classe `Pile_entier`.

Dans un langage modulaire, un module exécute les opérations qu'il exporte. En revanche, dans les langages à objets, une classe n'exécute pas directement les méthodes qu'elle définit. Pour utiliser une méthode d'une classe, on doit au préalable créer un objet de cette classe puis lui demander d'exécuter la méthode. La classe s'apparente donc à un composant formel du programme et l'objet à un composant effectif.

Dans les langages modulaires, un programme est un ensemble de modules et le «programme principal» est généralement un module particulier. Dans une approche où l'objet est un composant modulaire, un programme peut être décrit par un ensemble de classes et le «programme principal» est alors une classe particulière. Les langages Smalltalk ou Eiffel sont représentatifs d'une telle approche : la classe est l'unique mécanisme de structuration d'un programme et à l'exécution, un programme est un ensemble d'objets qui coopèrent. Bien entendu, dans ces langages, les objets peuvent être perçus à la fois comme des éléments de types de données et comme des composants modulaires.

2.2.4 Objet = agent pour la gestion du dialogue homme-machine

Comme la notion d'objet peut représenter un module logiciel ou un élément d'un type de données, l'objet est particulièrement bien adapté à la représentation des

«agents» utilisés pour structurer les systèmes interactifs.

Actuellement, la plupart des logiciels de base spécialisés pour le développement de systèmes interactifs sont fondés sur les techniques à objets. On pourra consulter l'ouvrage de J. Coutaz [Cout90] pour une comparaison plus précise des modèles multiagents et des modèles à objets.

2.3 Valeurs, Références et Constructeurs

2.3.1 Notion de valeur

Les types ou classes de base (entier, réel, booléen, chaîne, etc.) sont nécessaires pour élaborer de nouvelles classes. Le statut des éléments de ces classes de base diffèrent selon les systèmes à objets. Certains langages leur confèrent un statut de classe à part entière et leurs instances sont des objets (Smalltalk) alors que d'autres les représentent comme des types de données dont les éléments sont des valeurs (O2, Eiffel).

Une valeur ne peut pas être partagée et elle est manipulée par des opérateurs prédéfinis du langage alors qu'un objet peut être partagé par l'intermédiaire de son identificateur et il est manipulé par ses méthodes. On peut dire en simplifiant que l'affectation d'un objet entraîne la recopie de son identificateur permettant ainsi le partage alors que l'affectation d'une valeur entraîne la recopie de la valeur elle-même.

La notion de valeur est nécessaire pour représenter les propriétés propres d'un objet. Le `nom`, l'`âge` ou le `numéro d'INSEE`, propriétés propres d'une `Personne`, sont des exemples classiques de données qui ne doivent pas être partagées. Cependant l'expérience montre que certaines propriétés ne peuvent se ramener à des types de valeurs simples. On aimerait dans certains cas pouvoir utiliser les services offerts par le concept de classe (masquage d'information, héritage, etc.) pour représenter certaines propriétés propres d'un objet. M. Atkinson et P. Buneman ont énoncé le principe d'indépendance (orthogonalité) entre typage et persistance [AB87]. De façon similaire, on peut énoncer le principe d'orthogonalité entre le typage et la propriété :

- Tout objet, quel que soit sa classe, doit pouvoir être utilisé pour décrire une propriété d'un autre objet. Dans ce cas, le premier objet ne doit pas être partagé. La notion de valeur ne doit donc pas être limitée aux types de base.
- Le partage ou la propriété caractérise un objet et non une classe. Une classe doit pouvoir générer des objets partagés et des objets propriétés propres d'autres objets.

Dans Eiffel, les classes «expanded» permettent de définir de nouveaux domaines de valeurs tout en bénéficiant des services offerts par la notion de classe (masquage d'information et encapsulation). Cependant cette solution a l'inconvénient d'imposer le même comportement à tous les objets : aucune instance de classe «expanded» ne peut être partagée.

2.3.2 Référence = déclaration d'importation

Lorsqu'un premier objet référence un second objet, le premier peut accéder et utiliser les caractéristiques publiques du second objet. La référence constitue donc un support pour une relation client/fournisseur entre deux objets. Comme le remarque J-C. Boussard, la référence est le premier mécanisme de réutilisation des langages à objets [BMP*90] : «*L'importation d'une classe s'effectue de manière explicite comme en ADA ou en Modula-2, à l'aide de la déclaration d'un ou plusieurs objets de cette classe*». La déclaration des objets importés est portée par des attributs. Comme la référence permet le partage d'objets, un objet peut être fournisseur de plusieurs objets clients.

Lorsque l'objet est considéré comme un composant modulaire, la référence est à rapprocher de la relation d'utilisation : un `module1 utilise` un `module2`. Lorsque la référence est perçue comme une relation d'importation, elle est porteuse d'une information renseignant sur l'organisation et le fonctionnement du programme. Elle peut notamment représenter un lien de décomposition hiérarchique : un objet est décomposé en «sous-objets». L'objet initial, devenu composite, importe les caractéristiques de ses composants pour réaliser ses méthodes dans une forme de sous-traitance [Giro91]. De ce point de vue, la référence matérialise donc une relation de dépendance. Comme la recherche d'un couplage faible est un objectif de qualité pour l'élaboration de programmes, le nombre des références doit être autant que possible limité. Nous verrons dans la section 2.4 une définition plus complète de la notion de dépendance et dans la section 3.4 des règles permettant de limiter le couplage entre les objets.

2.3.3 Référence = association entre deux objets

La référence établit un lien ou un chemin d'accès d'un objet vers un autre objet. Si on considère une approche par des relations binaires, comme dans la méthode NIAM [Habr88], la référence peut représenter directement un des deux rôles d'une relation binaire. Si on adopte une approche par des relations n-aires, comme dans la méthode MERISE [TRC85], la relation peut être représentée à l'aide d'une classe et la référence indique la participation des classes à la relation. Lorsque la référence représente une association entre deux objets, la référence est porteuse d'une information concernant l'organisation du domaine. Dans une approche base de données, le concepteur doit élaborer un modèle du domaine dans lequel l'exhaustivité des informations et des relations est un critère de qualité à rechercher. La référence est à rapprocher du verbe AVOIR : une `Personne a` des enfants qui sont des `Personnes`, une `Personne a` une `Voiture`, etc.

```
classe Catégorie
  attributs     nom : Chaîne
                inscrits : ensemble (référence Inscription)
fin_classe

classe Inscription
  attributs     numéro_inscription : Entier
                coureur : référence Licencié
                résultat : Réel
fin_classe
```

```
classe Licencié
  attributs     num_licence : Entier
                personne : référence Personne
                club : référence Club
fin_classe

classe Personne
  attributs     nom : Chaîne
                prénom : Chaîne
fin_classe

classe Club...
```

Exemple 3-5 : Références = relations entre objets.

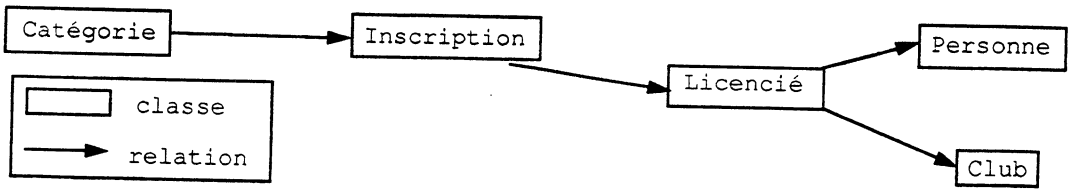


Figure 3-1 : Représentation graphique de l'exemple 3-5.

2.3.4 Notion de constructeur

Les constructeurs usuels sont l'ensemble, la liste et le tuple (ou nuplet), plus rarement la table et le choix (ou l'union). Dans les langages non «orientés objets», ces constructeurs sont utilisés pour élaborer de nouveaux types de données. Dans les langages à objets, ils permettent de préciser la structure des données désignées par les attributs d'une classe. Le statut de ces constructeurs diffère selon les systèmes considérés.

Dans un langage comme Eiffel, les constructeurs sont des classes génériques paramétrées par une ou plusieurs classes dites classes formelles. Un constructeur ne peut pas générer directement des objets mais l'instanciation de toutes ses classes formelles permet d'obtenir une classe susceptible de générer des objets. Les constructeurs peuvent être utilisés dans les attributs d'une classe ou par héritage. Les langages qui adoptent cette approche offrent souvent au programmeur la possibilité de définir de nouvelles classes génériques. Les constructeurs «de base» s'insèrent donc de façon homogène dans le modèle à objets. On peut noter que la généricité constitue, après l'importation et l'héritage, un troisième mécanisme de réutilisation.

Dans d'autres systèmes (O2 par exemple) les constructeurs sont des constructeurs de «valeurs» comme dans les langages traditionnels ou dans les SGBD à valeurs structurées. Cette approche, qui impose une distinction objet/valeur, est sans doute plus conforme à un point de vue conceptuel. En effet, les constructeurs n'ont pas de noms

particuliers, ils ne correspondent pas directement à des concepts manipulés par l'utilisateur et apparaissent seulement dans la description d'autres classes. On peut donc considérer que chaque constructeur utilisé dans une classe est local à cette classe et ne doit pas être partagé. Il s'agit donc bien dans ce cas de constructeurs de valeurs.

2.4 Graphes structurels et graphes des dépendances

La donnée d'un ensemble de classes et d'un ensemble de références entre ces classes définit un graphe que nous appelons **graphe structurel**. Les nœuds de ce graphe sont les classes et chaque référence en constitue un arc (cf. figure 3-1).

Lorsque les références sont utilisées pour représenter des associations du domaine, le graphe structurel est à rapprocher de la notion de schéma d'une base de données. Par ailleurs, chaque base d'objets est un **graphe d'objets** dont les nœuds sont les objets et les arcs les références entre les objets [Beer89].

Lorsque la référence est perçue comme une relation d'importation, on pourrait penser que le graphe structurel constitue le graphe des dépendances du programme[1] comme c'est le cas avec les langages modulaires dans lesquels les relations d'importation définissent la visibilité et limitent les relations client/fournisseur. Cependant, ce n'est pas le cas. En effet, un composant modulaire est dépendant d'un autre composant si certaines caractéristiques du second sont utilisées dans les traitements effectués par le premier. Dans les langages à objets, cette utilisation est concrétisée par l'envoi d'un message pour exécuter une méthode, consulter ou modifier un attribut. Or l'envoi d'un message peut être effectué vers un objet accessible directement ou vers un objet accessible transitivement. Au sein d'une méthode, l'accès direct concerne les objets accessibles par les attributs de l'objet ou par les arguments de la méthode. Un objet est accessible transitivement si il est renvoyé par un attribut ou une méthode d'un objet accessible directement ou indirectement. Le graphe structurel est donc insuffisant pour représenter le graphe des dépendances d'un programme.

Le **graphe des dépendances** d'un programme doit être élaboré directement à partir des messages échangés : chaque classe est un nœud du graphe et il existe un arc d'une première classe vers une seconde si au moins une méthode d'un objet de la première classe envoie un message à un objet de la seconde classe.

On constate donc une différence importante entre les langages modulaires et les langages à objets pour construire le graphe des dépendances. Dans les premiers, le graphe des dépendances est construit à partir des informations déclaratives mentionnées dans les modules (listes d'importation). Dans les langages à objets en revanche, l'élaboration du graphe des dépendances nécessite l'examen du code des méthodes de chaque classe.

---

1. Le graphe des dépendances met en évidence le couplage entre les modules logiciels d'un programme et permet d'évaluer les conséquences de la modification d'un composant.

2.5 Signature et corps d'une classe

2.5.1 Masquage d'informations et technique de programmation

Le masquage d'informations est un principe clé pour l'élaboration de programmes de qualité. Son application aux classes et aux objets conduit à distinguer la signature d'une classe et le corps de cette classe. Les variables locales à faible niveau d'invariance, ainsi que les méthodes «utilitaires» peuvent ainsi être masqués au profit d'un ensemble restreint d'attributs et de méthodes publics.

```
classe File_entier_double_entrée
  signature
     initialiser,
     ajouter_début (e : Entier),
     ajouter_fin (e : Entier),
     retirer_début : Entier,
     retirer_fin : Entier,
     file_vide : Booléen
  corps
     attribut début : référence Cellule_entier,
              fin : référence Cellule_entier
     méthode ajouter_début (e : Entier)
          (*algorithme de la méthode*)

     méthode ajouter_fin (e : Entier)
          (*algorithme de la méthode*)...
fin_classe

classe Cellule_entier
  signature    valeur : Entier,
               suivant : référence Cellule_entier
               précédent : référence Cellule_entier
fin_classe
```

Exemple 3-6 : Masquage d'informations.

Le masquage d'informations permet d'assurer que la modification des choix de programmation d'une classe ne remet pas en cause ses classes clientes.

Pour être efficace, le principe de masquage doit cacher les aspects les plus évolutifs d'un composant au profit de ses aspects les plus invariants. En programmation, le masquage privilégie un point de vue fonctionnel des classes en préconisant de cacher les données ; c'est pourquoi on parle de masquage d'«informations». Par ailleurs, on notera que lorsque les attributs sont publics, ils permettent de «naviguer» dans l'ensemble des objets ce qui peut conduire à une augmentation du couplage entre les objets.

2.5.2 Masquage d'informations et approche base de données

L'encapsulation et le masquage d'informations sont considérés comme des principes de base des techniques à objets et, à ce titre, ils sont intégrés dans les SGBDOO. Cependant la pratique montre que l'approche base de données s'accommode mal d'un masquage d'informations strict issu de la programmation. En effet, le respect du masquage d'informations conduit à cacher les attributs. Or ces attributs représentent les propriété-

tés propres des entités du domaine ou les relations entre ces entités et il est nécessaire de pouvoir les consulter. Cette nécessité conduit la plupart des SGBDOO à offrir des moyens de contourner ce masquage. Par exemple, par le biais d'un langage de requêtes pouvant consulter librement les attributs ou par un opérateur particulier (un «décapsuleur» dans O_2). A ce titre, on peut remarquer que certains systèmes nuancent le masquage en permettant un accès en lecture uniquement.

```
classe Compétition
  signature          (*caractéristiques publiques*)
    attribut
      nom : Chaîne,
      date : Date,
      organisateur : référence Club,
      arbitre_principal : référence Personne,
      catégories : ensemble (référence Catégorie),
      liste_départ : table [1..500] (référence Inscription),
    méthode
      initialiser (...),
      imprimer

  corps              (*caractéristiques privées*)
    attribut...
    méthode...
fin_classe Compétition
```

Exemple 3-7 : Une classe dans un schéma de base d'objets.

Dans un contexte de base de données, les informations ont souvent un niveau d'évolutivité voisin de celui des opérations. Le masquage des attributs au profit des méthodes n'est donc pas justifié sauf si l'objectif est de limiter la «navigation» dans l'ensemble des objets. Cependant, on peut introduire des règles méthodologiques limitant la visibilité pour restreindre le couplage entre les objets tout en autorisant la consultation des attributs. Nous présentons dans la section 3.4 un système à objets proposant de telles règles méthodologiques.

2.5.3 Attribut privé ou public et objet propre ou partagé

Dans la section 2.3 nous avons expliqué la différence entre un objet propre et un objet partagé et dans les deux sections précédentes (sections 2.5.1 et 2.5.2) nous avons discuté des attributs publics ou privés. Ces deux notions sont a priori indépendantes et devraient pouvoir être combinées librement. Notamment, un attribut donnant accès à un objet propre doit pouvoir être public et un attribut privé doit pouvoir donner accès à un objet partagé.

Cependant, en pratique, ces deux notions sont souvent liées car il est rare de disposer d'un mécanisme permettant de définir des attributs publics donnant accès à des objets propres et les objets sont par défaut partageables. Pour représenter un attribut public donnant accès à un objet privé, le programmeur doit exploiter le masquage d'informations : l'attribut est masqué et une méthode retourne au client une copie de l'objet propre. Dans les langages conventionnels, dans lesquels on manipule par défaut des va-

leurs, cette recopie est assurée automatiquement par le compilateur : lors d'une opération d'affectation, le compilateur génère le code de recopie de la valeur (contenu de la mémoire).

Cette remarque sur l'indépendance entre objet propre/partagé et attribut public/privé complète la remarque de la section 2.3.1 qui souligne la nécessité d'indépendance entre la notion de classe et la notion d'objet propre ou partagé.

2.6 Héritage : sous-ensemble, sous-typage et réutilisation

De façon générale, l'héritage permet de définir une nouvelle classe, appelée sous-classe, à partir d'une ou plusieurs classes, appelées super-classes. La sous-classe dispose par défaut de toutes les caractéristiques de ses super-classes sans qu'il soit nécessaire de les dupliquer. En outre il est possible de lui ajouter de nouvelles caractéristiques et dans certains systèmes il est possible de modifier les caractéristiques héritées. La relation d'héritage peut donner lieu à différentes interprétations [Brac83, Card84, Giro91]. Nous nous intéressons plus particulièrement à trois interprétations : ensembliste, réutilisation et sous-typage.

2.6.1 Interprétation ensembliste de l'héritage

Dans les approches bases de données, l'héritage est en général interprété comme une relation d'inclusion ensembliste. Dans cette perspective, en s'intéressant seulement aux attributs d'une classe, une classe C' est un sous-ensemble d'une classe C si :

- C' a les mêmes attributs que C et pour tout attribut «a : D» de C, C' a un attribut «a : D'» tel que soit D' $\subseteq$ D,
- C' a essentiellement de nouveaux attributs.

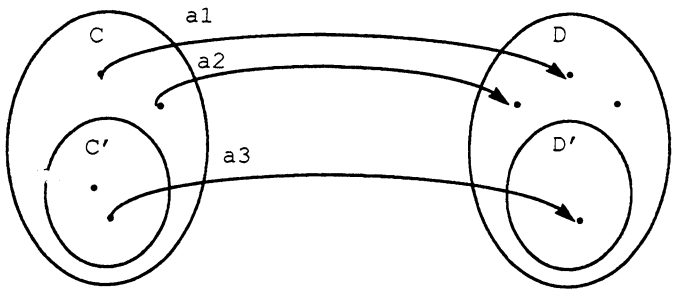


Figure 3-2 : Interprétation ensembliste de l'héritage.

Dans une telle interprétation, l'héritage peut donc permettre de spécialiser les attributs hérités et d'ajouter de nouveaux attributs. On notera que dans cette définition, nous avons pris en compte uniquement les données contenues dans un objet sans nous intéresser à ses méthodes ni aux programmes qui utilisent cet objet.

L'opérateur d'inclusion ensembliste présent dans certains modèles sémantiques comme le modèle Z de J-R. Abrial [Abri78] ou dans la méthode NIAM [Habr88] sont à rapprocher de cette interprétation de l'héritage.

2.6.2 Héritage : mécanisme de réutilisation

L'héritage peut être utilisé comme une technique de factorisation de connaissances (définition d'attributs et définition de méthodes). L'héritage permet alors de réutiliser la connaissance contenue dans un composant sans avoir à la dupliquer. Dans une telle utilisation de l'héritage, le programmeur peut d'une part ajouter des caractéristiques à la nouvelle classe et d'autre part redéfinir le plus librement possible les attributs et les méthodes héritées. L'exemple suivant permet d'illustrer une difficulté rencontrée dans une telle interprétation.

```
1     classe C
2       méthode m1 (i : Entier)...
3       méthode m2
4          début...
5             self.m1(23)
6             ...
7          fin_méthode
8     fin_classe C
9
10    classe D
11      hérite de C
12      redéfinition méthode m1 (a : Caractère)...
13    fin_classe D
```

Exemple 3-8 : Définition d'une classe `D` en réutilisant une classe `C`.

Dans l'exemple 3-8, on notera que la méthode `m2` de `C` utilise la méthode `m1` et que seule cette dernière est redéfinie dans la classe `D`. Considérons maintenant que l'on dispose à l'exécution d'un objet O instance de la classe `D`. Lors de l'appel de la méthode `m2` sur O, c'est l'algorithme décrit dans la classe `C` qui est exécuté puisque `m2` n'est pas redéfinie dans `D`. En revanche dans l'algorithme de `m2`, l'utilisation de `m1`, ligne 5, provoquera l'exécution de l'algorithme décrit dans `D` puisque `m1` est redéfinie. Si la méthode `m2` de `C` n'est pas recompilée avec la nouvelle signature de la méthode `m1` dans `D`, l'exécution de `m2` provoquera une erreur de type. On constate donc que, même dans une interprétation simple de l'héritage, le contrôle de type peut s'avérer délicat.

On peut envisager plusieurs solutions pour assurer un typage statique lorsque l'héritage est utilisé uniquement comme une technique de réutilisation :

- interdire toute redéfinition des caractéristiques héritées,
- imposer que les redéfinitions respectent les règles de sous-typage (cf. Annexe A),
- recompiler les méthodes des super-classes non redéfinies dans la nouvelle classe pour contrôler leur cohérence dans cette nouvelle classe.

2.6.3 Héritage et sous-typage

L'intérêt principal de l'héritage réside dans la possibilité d'interpréter ce mécanisme comme une relation de sous-typage. La relation de sous-typage entre deux types peut être définie de façon générale : «*le type T' est un sous-type du type T, noté T' ≤ T, si T' offre au moins le même comportement que T*» [LP91].

Dans les langages à objets, le type d'un objet correspond généralement à la signature de sa classe. Les systèmes diffèrent sur l'interprétation donnée aux termes «*au moins le même comportement*» dans la définition ci-dessus. Si tous les systèmes permettent d'enrichir un sous-type : ajout d'attributs ou de méthodes, en revanche, pour la redéfinition des caractéristiques héritées, on distingue les systèmes covariants et les systèmes contravariants.

Un système est **covariant** si les arguments et les résultats des méthodes peuvent être spécialisés dans le sous-type. Un système est **contravariant** si les résultats des méthodes peuvent être spécialisés et si les arguments peuvent être généralisés. Dans les deux cas, on peut effectuer un contrôle de type statiquement. Cependant, garantir un typage statique dans les systèmes contravariants est aisé alors que dans les systèmes covariants ce contrôle peut entraîner de nombreuses recompilations. L'expérience montre que la covariance est souvent plus utile que la contravariance. On se reportera à l'annexe A pour une étude plus approfondie du sous-typage.

La définition d'une classe C' comme sous-type d'une classe C permet aux objets de C' d'être considérés comme des objets de C. A ce titre, on peut, à l'exécution, utiliser un objet de C' partout où un objet de C est attendu et cela de façon transparente pour l'utilisateur. La relation de sous-typage garantit que cet échange n'entraîne pas d'erreur de type à l'exécution. Cette facilité est quelquefois appelée principe de substitution : un objet de C' peut être substitué à un objet de C.

Les avantages du sous-typage et de la substitution sont multiples. On retiendra plus particulièrement la facilité d'étendre un logiciel en spécialisant une partie des classes qui le composent sans affecter les autres classes ni les super-classes. Par ailleurs, comme une telle spécialisation n'affecte pas les classes qui composent les versions antérieures du logiciel, ces versions antérieures continuent à fonctionner.

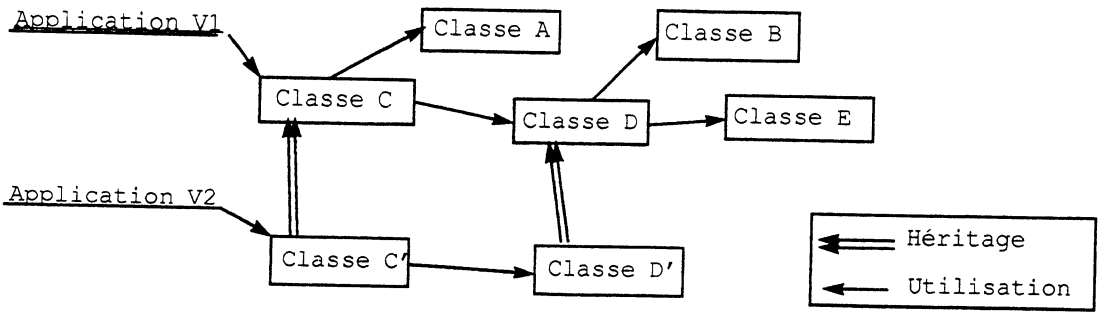


Figure 3-2 : Deux versions d'une application.

2.6.4 Résolution tardive ou liaison dynamique

La mise en œuvre pratique du sous-typage et de la substitution nécessitent des mécanismes spécifiques notamment pour localiser le code à exécuter lors de l'envoi d'un message. L'algorithme à exécuter dépend de l'objet receveur du message qui est connu seulement à l'exécution. Un mécanisme de recherche dynamique doit donc être mis en œuvre à l'exécution. Ce mécanisme est appelé **résolution tardive** ou **liaison dynamique** et différentes stratégies de recherche ont été proposées [DH89].

2.7 Eléments complémentaires

L'héritage multiple permet à une classe d'hériter de plusieurs autres classes. Le graphe d'héritage n'est plus une arborescence mais il doit toujours être sans cycle. L'utilisation de l'héritage multiple peut introduire des conflits liés à des homonymies de méthodes entre les différentes classes, en particulier lorsqu'une classe hérite plusieurs fois d'une même classe par des chemins différents. Selon les systèmes, ces conflits sont résolus automatiquement ou manuellement par renommage.

Certains modèles introduisent la notion de métaclasse pour supprimer la distinction entre classe et objet et uniformiser les concepts autour d'un concept unique : l'objet. Une métaclasse est une classe dont les instances sont elles-mêmes des classes ou des métaclasses. Les classes sont donc des objets instances de métaclasses. Le seul mécanisme de transfert de contrôle est alors l'envoi de message : une classe est créée par envoi de message à une métaclasse, un objet est créé par envoi de message à une classe et une caractéristiques d'un objet est utilisée en envoyant un message à cet objet. Cette solution présente l'avantage d'uniformiser les concepts.

3 Conception dirigée par les objets

Les techniques à objets présentées dans la section 2 regroupent un ensemble de concepts pour lesquels il n'existe pas à priori de «mode d'emploi». La conception dirigée par les objets développée ci-dessous propose différentes règles pour concevoir une application. Ces règles sont utilisables avec la plupart des langages de programmation. Cependant, les techniques à objets offrent des mécanismes particulièrement bien adaptés à cette approche. La conception dirigée par les objets est maintenant intégrée dans de nombreuses méthodes de conception d'applications.

3.1 Généralités

Le but de l'étape de conception est d'élaborer l'architecture des programmes qui réalisent les spécifications établies pendant la phase d'analyse. Pour cela, le concepteur peut s'appuyer sur différentes méthodes de conception et sur des modèles d'architecture. La conception dirigée par les objets est issue des travaux de D. L. Parnas au début

des années 70 [Parn72, Parn76]. Dans ces travaux, il affirme qu'un programme développé en utilisant l'abstraction de données est plus facile à faire évoluer qu'un programme conçu selon une approche fonctionnelle classique. La conception dirigée par les objets s'appuie également sur le constat qu'une structure calquée sur l'organisation des entités du domaine réel dans lequel l'application s'insère est plus stable qu'une structure élaborée autour de la décomposition fonctionnelle de l'application ou autour d'une hiérarchie de machines abstraites.

Comme le note J. Ferber [Ferb90], «*La conception par objets est une démarche extrêmement simple*». Le principe général de la conception dirigée par les objets est de calquer la structure logicielle d'un programme sur l'organisation du domaine dans lequel le programme est utilisé. Le programme est donc en partie un modèle du domaine et les composants logiciels correspondent à des abstractions des entités de ce domaine. Le programme est conçu comme une collection de composants en interactions. Chaque composant contient des informations et il est capable de réaliser des actions. La notion d'objet est donc particulièrement bien adaptée pour représenter de tels composants.

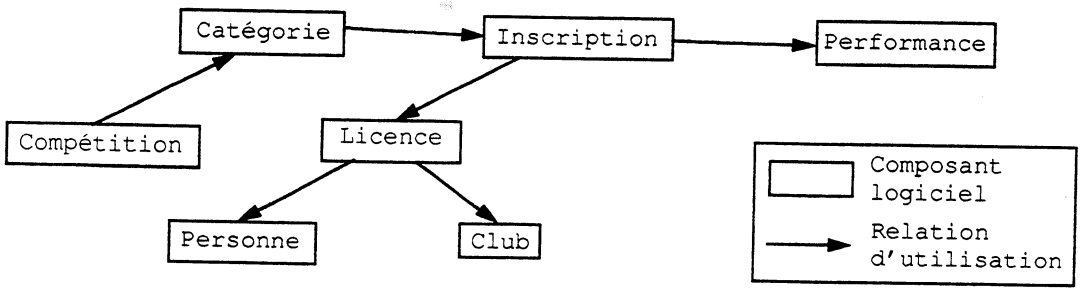


Figure 3-3 : Architecture logicielle générale d'une application.

Malgré la simplicité de la conception dirigée par les objets, l'élaboration d'une «bonne» architecture est une activité délicate et longue qui nécessite de nombreuses itérations et une solide expérience de la part du concepteur. Par ailleurs, la trop grande généralité des démarches de conception fondées sur cette approche ainsi que l'absence de métriques et de critères précis permettant de juger la qualité d'un programme et de ses composants ne facilitent pas la démarche de conception.

3.2 Principes fondamentaux

J. Ferber identifie plus particulièrement quatre principes fondamentaux sous-jacents à la conception dirigée par les objets [Ferb90] : l'opération de réification, le principe d'autonomie et de localité, le développement par composition et affinage et le concept de généricité.

La **réification** est l'opération essentielle du paradigme objet par laquelle quelque chose (chose physique, relation, événement, situation, idée, loi...) est représenté sous la forme d'un objet [Ferb90]. Cette opération repose sur le postulat que tout peut être re-

présenté par un objet. Concevoir par objet c'est donc d'abord identifier et modéliser les objets du domaine avant de s'intéresser aux fonctions de l'application. Ces fonctions viendront ultérieurement compléter le modèle du domaine. L'opération de réification est appliquée concrètement en transformant en objet toute chose à laquelle on attribue des propriétés, toute chose qui doit être manipulée ou toute chose qui agit.

Le principe d'**autonomie** et de **localité** exploite la décomposition du programme en unités indépendantes, chaque unité disposant de sa mémoire locale (attributs) et des opérations pour la manipuler (méthodes). Ce principe conduit à penser d'abord en termes d'objets puis en termes d'opérations. Il conduit également à demander à un objet d'accomplir lui-même les opérations qui le concernent ou les opérations dont il est responsable au lieu de manipuler ou de modifier cet objet à distance dans des procédures globales ou à partir des méthodes d'un autre objet. L'application de ce principe permet de limiter le couplage entre les objets tout en augmentant la cohésion de chaque objet. La réutilisabilité des composants et l'adaptabilité du système s'en trouvent améliorées.

Le développement par **composition** et par **affinage** encourage d'une part la réutilisation de composants existants et d'autre part l'élaboration de composants réutilisables. La composition est l'opération qui permet d'élaborer une nouvelle classe à partir d'objets existants ou au contraire de décomposer une classe en classes plus petites. Le développement par affinage est un procédé qui permet d'élaborer une nouvelle classe en enrichissant une classe existante. Techniquement, la composition est mise en œuvre par les attributs des classes et les références entre objets alors que l'affinage est réalisé par l'héritage.

Le concept de **généricité** s'appuie sur le développement par affinage et sur les services offerts par le sous-typage et la substitution. La généricité permet de penser en termes de famille de logiciels : à partir d'un premier programme, on peut développer progressivement une hiérarchie de programmes de plus en plus riches et spécialisés. Cette hiérarchie possède une propriété remarquable : à tous les niveaux les programmes restent opérationnels puisque les programmes plus spécialisés sont définis sans modification des classes existantes mais par définition de nouvelles classes plus spécialisées. Ce principe permet le développement progressif des programmes et encourage la réalisation de prototypes d'applications [Bézi86].

Bien que leurs buts soient différents, il existe des similitudes entre la conception dirigée par les objets et le processus d'élaboration d'un schéma de base de données. En particulier, l'opération de réification est essentielle dans ces deux processus. Cependant en base de données, les modèles de données proposent en général deux concepts ou deux familles de concepts pour élaborer les schémas de bases de données : l'objet et l'association entre objets. Dans les schémas conçus à l'aide de tels modèles, les objets et les associations sont des abstractions d'entités et d'associations du domaine. Dans la conception dirigée par les objets, l'opération de réification s'appuie sur un concept unique : l'objet, et cette opération s'applique à tout ce qui doit être manipulé ou qui possède des propriétés. Les liens entre objets n'appartiennent pas directement au domaine mais ils sont le reflet de l'organisation «fonctionnelle» du programme. Dans cette approche, les

associations du domaine sont souvent réifiées et représentées par des objets. Cette différence peut donc conduire à des schémas différents selon l'approche adoptée.

3.3 Erreurs fréquentes

Les habitudes de programmation peuvent conduire à l'élaboration de programmes contenant des structures considérées comme contradictoires avec les principes de la conception dirigée par les objets. Les erreurs les plus courantes sont les suivantes [Ferb90, Meye88] :

- Manipuler les objets à distance : au lieu de modifier directement les données d'un objet A à partir d'un autre objet B, il est préférable d'appeler depuis B une méthode de A qui effectue elle-même la modification requise.
- Demander la classe d'un objet avant d'agir : plutôt que de tester la classe d'un objet avant de lui envoyer un message, il est préférable de faire en sorte que quel que soit l'objet effectivement utilisé, il puisse traiter le message quitte éventuellement à définir une méthode vide.
- Faire des tests répétés : il est préférable de remplacer une série de tests sur le type d'un objet par une hiérarchie de classes permettant de supprimer ces tests.
- Confusion entre héritage et référence qui sont tous deux des mécanismes de réutilisation permettant de définir de nouvelles classes.

3.4 Règles méthodologiques du système Demeter

Nous avons vu dans la section 2.4. que les relations de dépendance et le couplage entre les objets dépendaient uniquement des envois de messages. Pour limiter ces dépendances, les concepteurs du système Demeter ont proposé un ensemble de règles méthodologiques connues sous le nom de Loi de Demeter [LHR88, LH89]. Le principe de la Loi de Demeter est de limiter les fournisseurs d'un objet. Cette loi a été exprimée dans deux versions (en termes d'objets ou de classes) et dans deux interprétations (forte ou faible).

Loi de Demeter (version objets) : dans une méthode m d'un objet O, on peut envoyer des messages aux objets suivants :

- i) les objets accessibles par les attributs de O,
- ii) les objets passés en argument lors de l'appel de m,
- iii) les objets créés par m directement ou indirectement, ces objets sont généralement accessibles par les variables locales de m,
- iv) les objets globaux accessibles directement par tous les objets.

Définie de la sorte, cette loi est difficilement vérifiable statiquement et un énoncé en termes de classe a été proposé. Ce second énoncé est moins restrictif que le premier mais il permet un contrôle statique de la loi.

Loi de Demeter (version classes) : dans une méthode m d'une classe C, on peut envoyer des messages aux classes suivantes :

- i) les classes accessibles par les attributs de C,
- ii) les classes des objets passés en argument à m et apparaissant dans la signature de m,
- iii) les classes des objets créés par m directement ou indirectement,
- iv) les classes des objets globaux accessibles directement par tous les objets.

Deux interprétations de ces lois peuvent être faites selon que l'on considère, dans les clauses i), les attributs hérités comme accessibles dans les sous-classes (interprétation faible) ou inaccessibles dans les sous-classes (interprétation forte).

Il a été montré que tout programme réalisé à l'aide d'un langage à objets en «style libre» pouvait être transformé en un programme respectant la Loi de Demeter [LH89]. Cependant, la transformation de certains algorithmes ne respectant pas la loi peut conduire à une augmentation importante de la complexité du programme. Cette loi qui concrétise le principe d'autonomie et de localité est donc surtout utile comme style de programmation et comme élément de référence pour le concepteur.

3.5 Démarche de conception

La méthode de conception Object Oriented Design [Booc82, Booc86] proposée par G. Booch constitue une des premières démarches dites «orientées objets». Elle a été développée comme une méthode de conception située en aval d'une analyse structurée classique et en amont d'une programmation en ADA. Dans ses premières versions, la méthode OOD n'intégrait pas les concepts essentiels des techniques à objets, absents également du langage cible, notamment la notion de classification et le mécanisme d'héritage. L'apport essentiel d'OOD réside dans un principe et dans une démarche.

Le principe de base d'OOD est d'élaborer l'application en considérant chaque objet du système comme un module du programme défini par une abstraction de données. La démarche proposée pour la conception d'une application est constituée de trois étapes :

1 - Définir le problème (étape d'analyse).

2 - Définir une stratégie informelle pour résoudre le problème posé.

3 - Formaliser la stratégie en identifiant les objets (modules) apparaissant dans la stratégie informelle, les attributs, les procédures et les interfaces de ces objets.

Les objets en question sont représentés par des modules ou «packages» d'ADA. Cette première démarche n'adhère pas explicitement aux principes de la conception dirigée par les objets. En effet, les «objets» sont identifiés dans le domaine de la solution informelle et dépendent donc plus de la stratégie adoptée par le concepteur pour élaborer cette solution informelle que de l'organisation du domaine dans laquelle s'insère l'application. On notera cependant qu'une version plus récente de la méthode OOD [Booc91] intègre les techniques à objets, nous la présentons dans la section 4.

Pour J. Ferber, la démarche générale d'une conception dirigée par les objets s'organise autour de cinq étapes parcourues de façon itérative [Ferb90] :

1 - Identifier les objets du domaine.

2 - Structurer le domaine par l'analyse des propriétés des objets et des relations entre les objets.

3 - Identifier les opérations et les interactions entre les objets.

4 - Décrire précisément les opérations et leurs algorithmes.

5 - Lancer l'exécution.

A ces cinq étapes plusieurs auteurs ajoutent une étape de généralisation des classes dont l'objectif est d'améliorer la réutilisation en produisant, à partir des classes identifiées dans le programme, des classes plus générales et donc plus réutilisables.

3.6 Méthodes orientées objets et structurées

Dans la mouvance d'OOD, de nombreuses méthodes de conception se réclamant du courant «orienté objet» et du courant «structuré» ont été développées. Les méthodes comme HOOD [Heit87], MACH2 [HL89], MAC-ADAM [Rigu87] appartiennent à cette catégorie et sont caractérisées par :

- Elles sont généralement dédiées au langage ADA ou aux langages offrant un mécanisme de module permettant le masquage d'informations et supportant les traitements temps réel ou le parallélisme (ADA, Modula-2, LTR3...).
- Elles se situent dans le courant des méthodes structurées soit parce qu'elles sont un prolongement de méthodes de conception structurée (décomposition fonctionnelle, hiérarchie de machines abstraites, etc), soit parce qu'elles offrent deux relations principales : l'inclusion d'objets (hiérarchie père/fils) et l'importation d'objets (relation client/fournisseur).
- Elles n'intègrent pas toutes les techniques à objets. Les notions de classe (classification/instanciation) et d'héritage (généralisation/spécialisation, substitution/polymorphisme) ne sont pas toujours proposées.
- Elles reprennent et étendent la démarche de conception proposée dans OOD.

Nous n'approfondissons pas plus l'examen de ces premières méthodes qui n'intègrent pas totalement les techniques à objets ni les principes de la conception dirigée par les objets car elles nous semblent inadaptées à notre contexte.

3.7 Analyse orientée objets

Dans une approche orientée objets comme dans une approche plus conventionnelle, l'étape d'analyse permet d'établir les spécifications du système informatique à dévelop-

per sans se préoccuper de la technique utilisée. Les méthodes d'analyse et de conception conventionnelles reposent souvent sur des modèles différents pour décrire les spécifications et l'architecture du programme. Cette situation impose donc un changement de formalisme et parfois un changement d'approche entre l'analyse et la conception qui conduit à une perte de sémantique et à une charge de travail supplémentaire pour le concepteur. L'intégration du modèle à objet dans toutes les phases du cycle de vie d'un programme permet de pallier ces inconvénients. Dans cette perspective, le cycle de vie peut être résumé par [CY90] :

- L'analyse permet d'identifier les objets et les classes de l'univers de l'utilisateur et d'établir les spécifications du programme en s'appuyant sur les techniques à objets.
- Pendant la conception, les objets et les classes identifiés pendant l'analyse sont utilisés pour élaborer une solution technique répondant aux spécifications. De nouvelles classes peuvent être nécessaires.
- Les classes sont enfin programmées à l'aide d'un langage à objets.

De nombreux travaux sont actuellement menés dans le domaine de l'analyse orientée objets [Brun91] et des méthodes d'analyse ont été proposées OOA [SM88], O* [CR89], OOA [CY90], MAO [PV91], OOM [Roch91], MCO [Cast91]. On peut dégager un concept supplémentaire commun à la majorité de ces travaux : la notion d'événement qui vient compléter les concepts habituels des techniques à objets.

4 Quelques méthodes de conception orientées objets

4.1 Object Oriented Design

La méthode «Object Oriented Design» proposée en 1991 [Booc91] est la dernière évolution de la méthode de conception OOD proposée par G. Booch au début des années 80. Contrairement aux versions antérieures, cette version, que nous notons OOD91, intègre tous les concepts des techniques à objets mais reste cependant très orientée vers ADA.

Les modèles d'OOD91 sont répartis en deux niveaux : le **niveau logique** où le système est représenté de façon abstraite selon un modèle orienté objet et le **niveau physique** qui permet de construire le système à l'aide de concepts plus proches des langages modulaires. Dans chacun de ces niveaux le concepteur décrit les aspects statiques et dynamiques du système.

OOD91 propose une représentation graphique pour tous les concepts des modèles de la méthode.

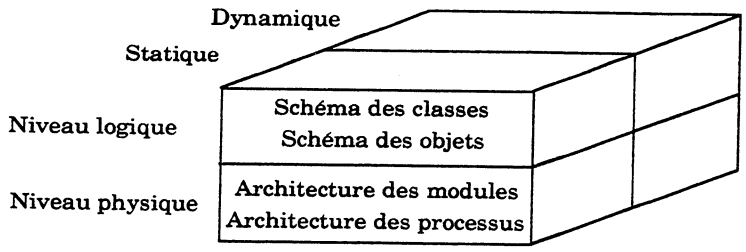


Figure 3-4 : Organisation d'OOD91.

4.1.1 Modèles

4.1.1.1 Niveau logique

Le niveau logique est organisé autour d'un modèle orienté objet et il est composé de quatre types de diagrammes : des diagrammes de classes, des diagrammes de transitions, des diagrammes d'objets et des diagrammes de séquencement.

• **Les diagrammes de classes :**

Un **diagramme de classes** est construit à l'aide de cinq éléments : les classes, les relations inter-classes, les classes utilitaires, les domaines des classes et les relations de visibilité.

Une **classe** est à la fois un modèle d'objets et le regroupement des objets générés par la classe. Les caractéristiques d'une classe sont : les classes utilisées, les attributs ou propriétés propres et les opérations. On peut y ajouter : le nombre d'objets générés, la persistance éventuelle de ces objets et leurs modes d'exécution : concurrents, séquentiels ou bloquants. Les caractéristiques peuvent être publiques, privées ou protégées. Chaque opération est définie par : une précondition, un algorithme, une postcondition, les exceptions levées, les paramètres et la complexité en temps et en espace mémoire.

Les **relations inter-classes** sont diverses : 1) utilisation entre une classe cliente et une classe fournisseur (on peut préciser la connectivité), 2) héritage entre classe et sous-classe, 3) instanciation entre une classe générique et une de ses concrétisations, 4) métaclasse pour un modèle tout objet et enfin 5) indéfinie lorsque la nature exacte d'une relation entre deux classes est provisoirement inconnue.

Une **classe utilitaire** regroupe des sous-programmes indépendants dont la réalisation peut faire intervenir des classes ou des classes utilitaires. Une classe utilitaire ne représente pas une catégorie d'objets et elle est définie par : les classes utilisées, les propriétés utilisées et les opérations réalisées.

Un **domaine de classes** est un groupement logique de classes définissant un concept plus abstrait que la classe. Un domaine de classes est défini par un diagramme composé de classes et éventuellement de domaines de classes. Les classes définies dans

un domaine peuvent être privées ou exportées par le domaine.

Une **relation de visibilité** est définie entre deux domaines : si un domaine B est visible d'un domaine A, alors les classes de A peuvent établir des relations avec les classes exportées par B. Un domaine peut être global, il est alors visible directement de tous les autres domaines. Le système est décrit par un diagramme des classes racines regroupant les principales abstractions du domaine et une hiérarchie de diagrammes reliés par des relations de visibilité.

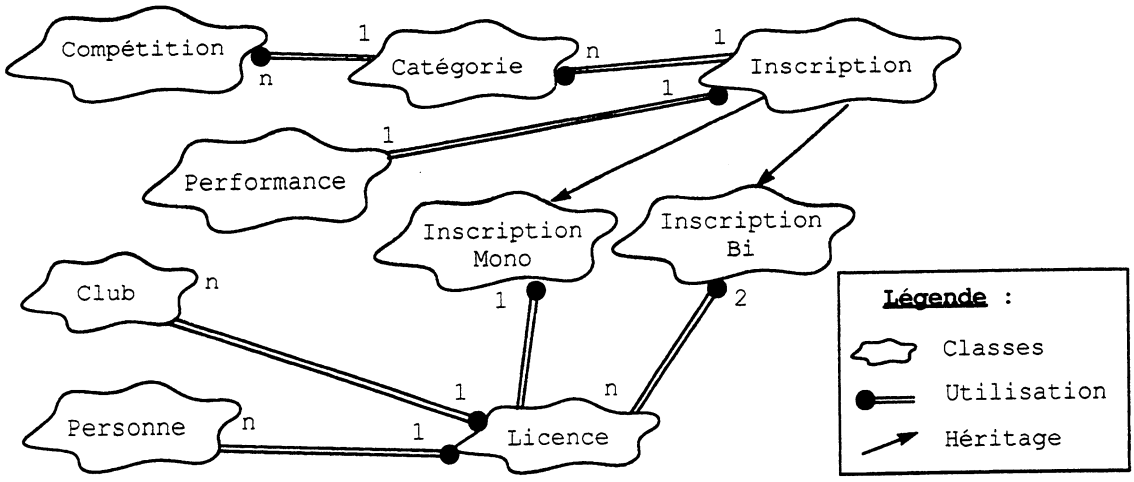


Figure 3-5 : Un diagramme de classes selon OOD91.

• **Les diagrammes de transitions :**

Un **diagramme de transitions** complète chaque classe avec les informations suivantes : les différents états des objets de la classe, les événements provoquant les transitions et les actions induites par les changements d'états.

• **Les diagrammes d'objets :**

Un **diagramme d'objets** permet de montrer l'existence des objets, leur cycle de vie et leurs inter-relations. Le rôle de chaque diagramme d'objets est d'illustrer certains mécanismes clefs du schéma logique et de montrer comment le système doit se comporter à l'exécution. Les diagrammes de classes et les diagrammes d'objets permettent de mettre en lumière des aspects différents du système : le premier montre les principales abstractions manipulées dans le système alors que le second met en évidence les mécanismes importants que réalisent ces abstractions.

Un **objet** est décrit par un nom facultatif, sa classe mère et sa durée de vie : persistant, statique (durée de vie égale à la durée du programme), dynamique (objet créé et détruit dans le programme). Une **relation** entre deux objets indique qu'ils peuvent échanger des messages. Cette relation peut être orientée ou bi-directionnelle. Un **message** est porté par une relation et définit l'opération à réaliser, son caractère périodique

ou apériodique ainsi que sa séquentialité.

La méthode OOD91 propose différents moyens pour représenter la dynamique du système (numérotation de l'ordre d'envoi des messages, diagramme de synchronisation associé à chaque opération parallèle d'un diagramme, etc.)

4.1.1.2 Niveau physique

Le niveau physique est constitué de deux types de diagrammes : les diagrammes des modules et les diagrammes des processus.

Un **diagramme des modules** représente l'architecture modulaire du système et résulte de la traduction des classes et des objets en modules. Différentes catégories de modules sont proposées : le programme principal, les sous-programmes, les sous-programmes génériques, les paquetages, les paquetages génériques, les tâches, etc. La **visibilité** (importation) qui exprime une dépendance de compilation est la seule relation pouvant exister entre deux modules. Un **sous-système** est, pour les modules, ce que le diagramme de classes est pour les classes : il représente un regroupement de modules logiquement reliés entre eux.

Les **diagrammes des processus** permettent de visualiser et de raisonner sur les problèmes d'allocation de ressources. Les diagrammes des processus sont constitués de trois éléments : les processeurs capables d'exécuter des programmes, les périphériques sans capacité d'exécution et les connexions entre les éléments précédents.

4.1.2 Démarche

La démarche préconisée par OOD91 est articulée autour de quatre étapes :

- Identification des classes et des objets : 1) mise en évidence des abstractions clefs de l'espace du problème, 2) définition des mécanismes importants produisant le comportement souhaité pour les fonctions réalisées par les objets.
- Caractérisation de la sémantique des classes et des objets : 1) identification des actions réalisées par les objets, 2) définition des interfaces et des protocoles d'échanges.
- Identification des relations entre classes et entre objets et mise en évidence de leurs interactions. Le choix des interfaces et la définition des visibilités peuvent entraîner une réorganisation du schéma.
- Réalisation des classes et des objets : 1) choix de représentation des classes et des objets, 2) allocation des classes et des objets aux modules, 3) allocation du programme aux processeurs.

4.1.3 Remarques

OOD91 est adapté à la conception de systèmes temps réels et parallèles. Le niveau logique permet de déterminer l'architecture générale du système à l'aide d'un modèle orienté objet et de préciser son fonctionnement. Le niveau logique conduit à une repré-

sentation détaillée du logiciel à l'aide de concepts proches du langage ciblé : ADA.

La persistance des objets peut être précisée mais il n'y a pas d'indépendance entre typage et persistance : toute instance d'une classe persistante est persistante. Par ailleurs, OOD91 ne s'intéresse pas du tout à la gestion du dialogue avec l'utilisateur.

4.2 Object Oriented Analysis and Design

La méthode Object Oriented Design [CY91] est proposée par P. Coad et E. Yourdon. Cette méthode complète la méthode Object Oriented Analysis [CY90]. Nous noterons OOA/OOD la combinaison des deux méthodes. La méthode OOA permet d'élaborer un modèle du domaine de l'application en cinq couches : objet, «classe&objets», structure, attribut et service. A partir d'une analyse menée avec OOA, OOA/OOD permet d'élaborer en détail une architecture logicielle organisée initialement en quatre composants : le composant «domaine du problème», le composant «interface homme-machine», le composant «gestion de tâches» et le composant «gestion des données persistantes». Nous présentons tout d'abord les concepts généraux puis nous approfondissons ces quatre composants.

4.2.1 Modèle

Les concepts proposés par OOA/OOD pour l'analyse et la conception reposent sur trois principes : la **description**où l'on distingue les objets et les attributs de ces objets, la **distinction** effectuée entre les parties d'un objet et l'objet complet et la **classification** des objets en classes d'objets similaires.

OOA/OOD offre les notions habituelles de classe, d'objet et d'héritage et propose en outre la notion de **classe&objet** qui représente le regroupement d'une classe (modèle d'objets) et des objets générés par cette classe. On notera la distinction entre la notion de classe qui représente un modèle d'objets et la notion de classe&objets qui décrit à la fois un modèle d'objets et l'ensemble des objets de la classe. Les attributs d'une classe représentent uniquement des propriétés propres. Les **connexions d'instances** représentent des relations structurelles entre des objets (liaisons bidirectionnelles) alors que les messages représentent des relations fonctionnelles entre des objets. La composition d'objets est supportée par une relation **tout/partie** entre un objet composé et ses parties composantes représentées également à l'aide d'objets.

Enfin l'ensemble des classes peut être organisé en sujets. Un **sujet** est une bibliothèque de classes. Le sujet est une construction utile pour clarifier le schéma de conception mais cette construction n'apparaît pas directement dans le texte des programmes.

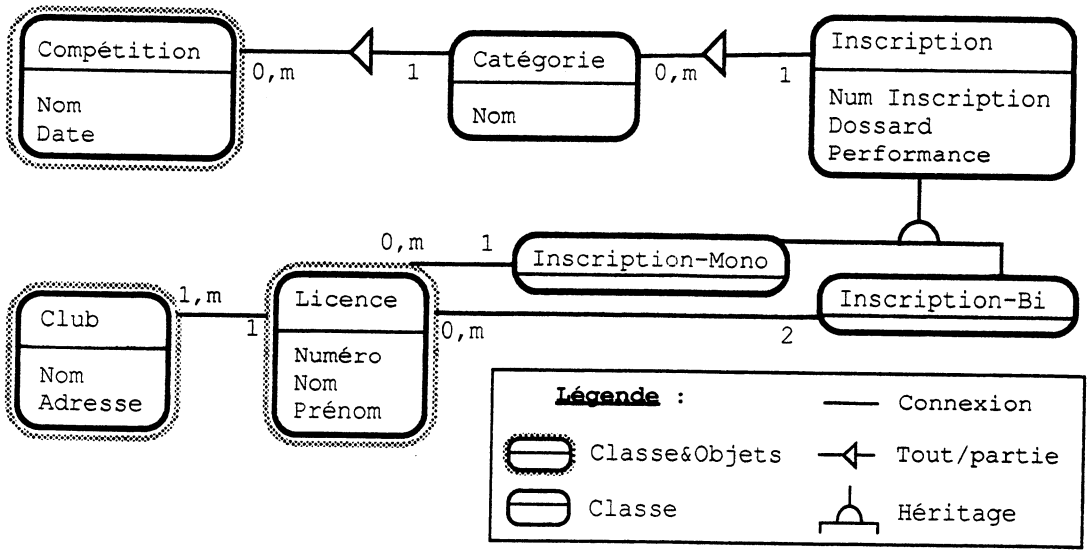


Figure 3-5 : Un schéma de classes selon OOA/OOD.

4.2.2 Démarche

Nous étudions successivement les spécificités des quatre composants constituants le modèle d'architecture préconisé par la méthode OOA/OOD. Chacun de ces composants est décrit à l'aide des concepts présentés ci-dessus.

- **Composant «domaine du problème»**

Le composant «domaine du problème» est directement issu du modèle élaboré pendant la phase d'analyse menée avec OOA. Cependant, ce modèle peut être adapté pour réutiliser des classes existantes, pour améliorer les performances du programme, pour supporter le composant base de données ou pour tenir compte des facilités offertes par le système cible.

- **Composant «interface homme-machine»**

Le composant «interface homme-machine», noté CIHM, est introduit pour gérer le dialogue entre l'utilisateur et le programme. Ce composant permet notamment de décrire comment les informations sont présentées à l'utilisateur. La stratégie proposée pour concevoir le CIHM est décomposée en différentes étapes : 1) classer les utilisateurs, 2) décrire chaque classe d'utilisateurs et le scénario des tâches qu'ils réalisent, 3) concevoir une hiérarchie de commandes et de métaphores pour dialoguer, 4) concevoir en détail l'interaction, 5) prototyper le CIHM et 6) élaborer les classes du CIHM qui dépendent du gestionnaire d'interface.

- **Composant «gestion des tâches»**

Le composant «gestion des tâches», noté CGT, permet, dans le cas de systèmes concurrents, de gérer l'exécution de différents processus. L'élaboration du CGT s'effectue en cinq étapes : 1) identifier les tâches dirigées par des événements, comme par exemple les tâches de communications et identifier les tâches dépendant d'une horloge, 2) identifier les tâches prioritaires et identifier les tâches critiques, 3) définir une tâche principale pour coordonner l'ensemble des autres tâches, 4) approfondir et «peser» l'utilité de chaque tâche et 5) définir précisément chaque tâche.

- **Conception du composant «gestion de la base de données»**

Le composant «gestion de la base de données», noté CGBD, est conçu pour permettre de stocker et de retrouver les objets persistants. Ce composant permet d'isoler l'impact du choix technologique pour le stockage des données persistantes (fichiers, SGBD, etc.) sur le reste du programme. La conception de ce composant comporte deux étapes : la définition du format des données et la définition des services offerts aux autres composants.

4.2.3 Outils

La méthode OOA/OOD propose deux outils : OOATool pour l'étape d'analyse et OODTool pour la conception. Ces outils sont disponibles dans différents environnements : Macintosh, PC, Unix, etc.

4.2.4 Remarques

Les méthodes OOA et OOD couvrent les phases d'analyse et de conception. Les concepts utilisés sont communs aux deux étapes. Ils sont simples et proches des concepts offerts par les langages à objets ce qui facilite la programmation finale de l'application. Cette méthode prend clairement en compte les trois dimensions qui nous intéressent : les fonctions spécifiques, les données persistantes et le dialogue homme-machine. Cependant, aucune précision n'est apportée quant aux communications possibles entre les quatre composants d'une application. Par ailleurs, la séparation habituelle entre la base de données et les programmes est maintenue. Cette séparation n'apporte pas de solution nouvelle et ne permet pas d'exploiter les services offerts par les langages de programmation de base d'objets et notamment par les SGBDOO.

4.3 Class Responsabilities

La méthode «Class-Responsabilities» proposée par R. Wirfs-Brock, B. Wilkerson et L. Wiener [WWW90] est adaptée au développement de programmes. Cette méthode propose une approche qualifiée de «dirigée par les responsabilités» par les auteurs car elle consiste à s'intéresser en priorité aux actions dont l'objet est responsable et aux informations qu'il propose aux autres objets.

4.3.1 Modèle

La méthode «Class-Responsabilities» s'appuie sur sept concepts.

Une **classe** est définie par un ensemble de **responsabilités** (caractéristiques). Une responsabilité est une méthode ou un attribut.

Une **collaboration** représente une requête d'un client vers un fournisseur. Une telle requête peut prendre trois formes différentes : partie-de, à-connaissance-de ou dépend-de.

Un **contrat** regroupe l'ensemble des responsabilités adaptées à un client particulier. Une classe peut avoir plusieurs contrats adaptés à différents clients mais une responsabilité appartient à un contrat unique.

Le **protocole** d'un objet (son interface) est définie par l'ensemble de ses contrats.

Il existe différentes **relations** inter-classes : sorte-de (héritage), analogue-à (superclasse commune) et partie-de.

Un **sous-système** est un ensemble de classes et éventuellement de sous-systèmes qui collaborent pour réaliser un ensemble de responsabilités offertes par le sous-système et également regroupées en contrats. Un sous-système est une unité conceptuelle dont le contenu (classes et sous-systèmes) est masqué par le protocole du sous-système.

La méthode propose deux représentations hiérarchiques globales du système : le graphe d'héritage et les diagrammes de Venn (représentation ensembliste et graphique du graphe d'héritage).

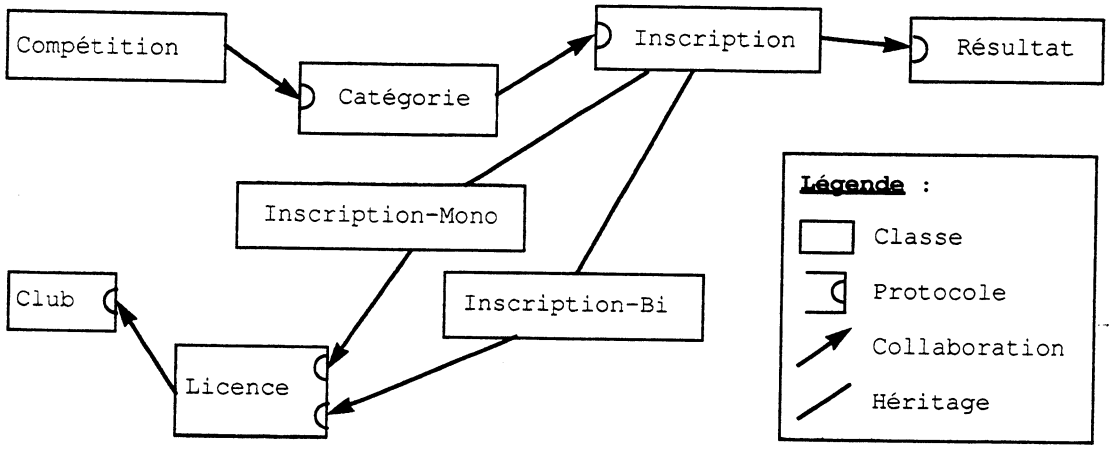


Figure 3-5 : Une conception «Class-Responsabilities».

4.3.2 Démarche

La démarche préconisée par la méthode «Class-Responsabilities» est composée de deux étapes :

- L'exploration initiale : 1) rechercher les classes, 2) définir les opérations dont chaque classe est responsable (on peut s'appuyer sur une recherche des verbes dans les spécifications), 3) déterminer comment les objets collaborent pour réaliser leurs responsabilités.
- L'exploration détaillée : 1) examiner les hiérarchies d'héritage, 2) examiner les collaborations et les contrats.

4.3.3 Remarques

La méthode «Class-Responsabilities» permet de concevoir des programmes mais n'offre pas de facilité adaptée à l'élaboration du gestionnaire du dialogue avec l'utilisateur ni à la représentation ou à la gestion des données persistantes. Les requêtes sont notamment les seules relations entre objets, il n'existe pas explicitement de relations structurelles entre objets. Par ailleurs, on peut reprocher à cette méthode de proposer un vocabulaire particulier différent du vocabulaire consensuel et un manque de formalisation qui ne facilitent pas la compréhension des concepts.

4.4 Object Modeling Technique

La méthode Object Modeling Technique [RBP*91] couvre tout le cycle de développement d'une application qu'elle organise en quatre étapes : l'analyse (spécification), la conception du système (architecture), la conception des objets (algorithmes détaillés) et la réalisation (programmation).

4.4.1 Modèles

OMT propose les mêmes concepts pour toutes les étapes du développement d'une application. Ces concepts permettent d'élaborer trois schémas : le schéma d'objets, le schéma dynamique et le schéma fonctionnel.

4.4.1.1 Shéma d'objets

Le **schéma d'objets** représente les aspects statiques du système.

Une **classe** définit un modèle d'objets et regroupe l'ensemble des objets correspondant à ce modèle. Les attributs représentent uniquement des propriétés propres (valeurs) et on peut distinguer les attributs stockés et les attributs calculés. Parmi les méthodes on distingue les méthodes de consultation et les opérations.

L'**association** entre classes est un concept à part entière. Une association est un modèle de liens entre objets. Les associations ne sont pas orientées, leur arité peut être quelconque et on peut les compléter avec des cardinalités et des attributs.

Une **clé** est un ensemble d'attributs identifiant chaque objet d'une classe ou chaque lien d'une association. Une classe ou une association peut avoir plusieurs clés. Des **contraintes** peuvent être ajoutées pour restreindre les valeurs des attributs d'une entité ou d'un lien. Ces contraintes sont exprimées par des équations ou en langue naturelle et peuvent être placées sur les classes ou sur le schéma fonctionnel (cf. section 4.4.1.3).

L'**agrégation** permet d'élaborer des objets composés à partir d'autres objets. Un agrégat peut être traité comme un objet unique sans se soucier de sa composition. L'agrégation n'est pas exclusive : un composant peut apparaître dans plusieurs agrégats. Certaines opérations peuvent se propager de l'agrégat vers ses composants.

L'**héritage** entre classes est supporté.

Un **module** est un constructeur logique pour regrouper des classes, des associations et des hiérarchies. Un module est une vue dont les limites expriment un point de vue particulier sur le système. Une classe peut apparaître dans différents modules.

La **feuille** est un moyen de découper un système de grande taille en plusieurs pages «physiques».

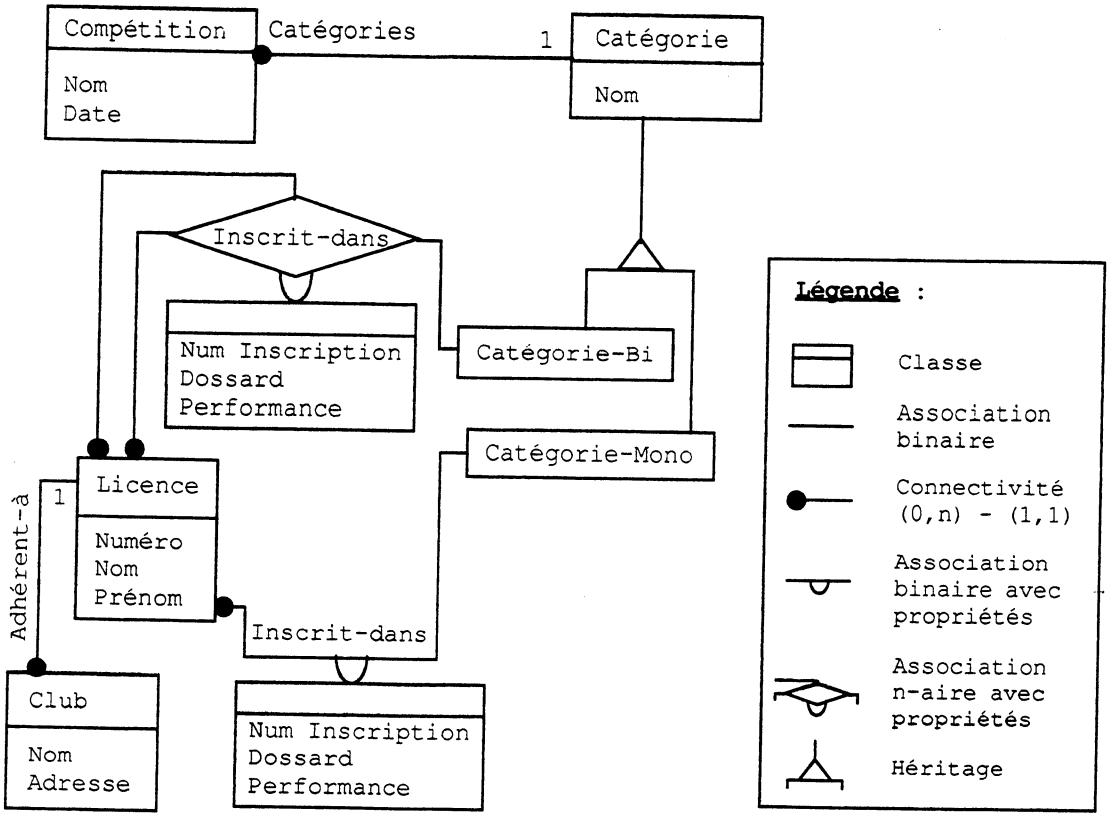


Figure 3-7 : Un schéma d'objets en OMT.

4.4.1.2 Schéma dynamique

Le **schéma dynamique** permet de décrire les séquences d'opérations qui sont réalisées en réponse aux stimulis extérieurs. Le schéma dynamique est élaboré à partir des notions d'événement et d'état.

Un **événement** est «quelque chose» qui arrive à un instant donné mais qui n'a pas de durée. Les événements sont regroupés en classes d'événements, chaque classe décrit la structure et le comportement d'une catégorie d'événements.

Un **état** est défini par un nom, la séquence d'événements le produisant, la condition le caractérisant et les événements acceptés dans cet état. L'opération déclenchée par chaque événement est précisée ainsi que l'état atteint après cette opération.

Un **diagramme d'états** est un graphe dont les nœuds sont les états, les arcs les transitions et les événements des étiquettes portées par les arcs. Un diagramme d'états peut être défini pour chaque classe. Les diagrammes d'états peuvent être structurés selon des relations d'agrégation et de spécialisation comme c'est le cas pour les objets.

Le schéma dynamique est une collection de diagrammes d'états interagissant par des événements partagés.

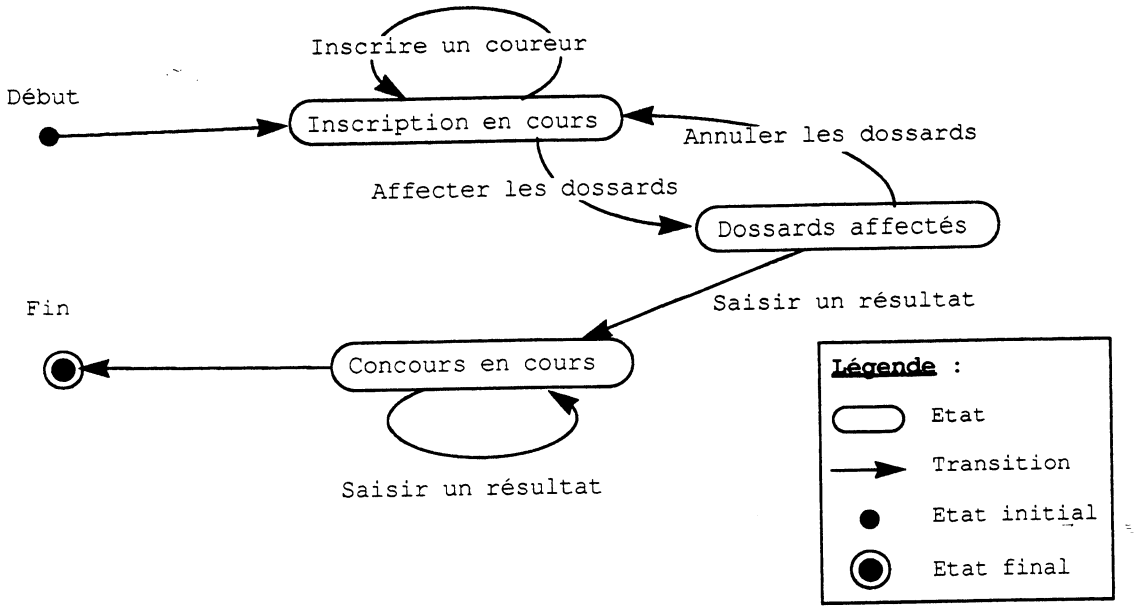


Figure 3-6 : Un schéma dynamique en OMT.

4.4.1.3 Schéma fonctionnel

Le **schéma fonctionnel** permet de décrire la relation existant entre les données d'entrée et les données de sortie à l'aide de différents diagrammes de flots de données.

Un **diagramme de flot de données** est construit à partir de processus, de flots de

données, d'acteurs et de mémoires de données. Les **processus** transforment les données. Les **flots de données** transfèrent les données. Les **acteurs** sont des objets actifs qui consomment et produisent des données. Les **mémoires de données** sont des objets passifs qui stockent les données.

4.4.2 Démarche

La démarche de conception préconisée par la méthode OMT couvre tout le cycle de développement d'un programme : analyse, conception et réalisation.

La démarche d'analyse permet d'élaborer les trois schémas : objet, dynamique et fonctionnel.

La démarche de conception est organisée en neuf étapes :

- Combinaison des trois modèles issus de l'analyse pour obtenir les opérations réalisées par les classes.
- Conception des opérations : choix des algorithmes, des structures de données et des classes internes.
- Optimisation de la conception : ajout d'associations redondantes pour améliorer certains accès, modification de l'ordre d'exécution pour améliorer l'efficacité et sauvegarde de certains attributs calculés pour éviter les calculs.
- Programmation du contrôle spécifié à l'aide d'un modèle à événements.
- Ajustement de l'héritage par ré-arrangement des classes et des opérations et par la recherche d'abstractions communes à plusieurs classes.
- Conception des associations : analyse des chemins d'accès, des connectivités, des attributs d'associations et réalisation des associations par des classes et des attributs.
- «Packaging physique» : définition des modules physiques, mise en place du masquage d'informations et contrôle de la cohésion de l'ensemble.
- Documentation de la conception.

4.4.3 Remarques

OMT est une méthode complète et riche qui applique le modèle à objet à tout le cycle de développement d'un programme. Cependant, la richesse du modèle et le degré d'abstraction des concepts (événements, diagrammes d'états, etc.) rend cette méthode plus particulièrement adaptée à la phase d'analyse. Certains points du modèle paraissent cependant trop restrictifs, notamment l'obligation qui est faite aux attributs de prendre leur valeur dans des types de base. Bien que les associations soient traitées selon une approche qui rappelle celle des modèles de données des bases de données, cette méthode est surtout adaptée à la production de programmes et la persistance, caractéristique fondamentale des bases de données, n'est pas approfondie. La description du dialogue homme-machine et sa gestion ne sont pas non plus prises en compte.

4.5 Mecano

La méthode MECANO (Méthode et Environnement de Construction d'ApplicatioNs par Objets) [Giro90, Giro91] a été développée dans le cadre d'un contrat entre le LGI de Grenoble et la société Hewlett-Packard d'Eybens. Cette méthode est centrée sur l'étape de conception et ne couvre pas l'étape d'analyse.

4.5.1 Modèle

On retrouve dans le modèle MECANO la plupart des concepts d'Eiffel complétés dans deux directions. Les relations d'utilisation et d'héritage sont affinées et de nouveaux concepts structurants de granularité supérieure à celle des classes sont proposés.

- **Concepts de base**

Une **classe** décrit un modèle d'objets. Elle est composée de six parties : l'interface (liste des attributs et méthodes exportés), la spécification (informations concernant la gestion du projet, définition de l'invariant), la généricité (nom du paramètre formel et éventuellement propriété associée), l'héritage (liste des classes parentes), le renommage (liste des méthodes renommées), les méthodes (spécifications, variables locales et corps) et les attributs (nom, type, spécification).

Une classe **virtuelle** est une classe dont certaines méthodes n'ont pas de réalisation permettant ainsi un polymorphisme plus général. Les classes génériques et les classes virtuelles sont appelées **classes abstraites** car elles ne peuvent pas générer d'objets.

- **Relations d'utilisation :**

La référence entre objets est appelée relation de **délégation** et elle est interprétée comme une importation permettant l'envoi de messages entre objets (relation client/ fournisseur). Cette relation est affinée pour tenir compte de la nature de la relation entre les deux objets. On distingue ainsi : la communication, la composition et l'utilisation secondaire.

La **communication** appelée aussi délégation collatérale est une relation proche de la communication entre processus dans l'approche parallèle. A communique avec B si : A est client de B, si A et B sont sur le même niveau conceptuel et si les instances de A et celles de B ont des cycles de vie a priori indépendants.

La **composition** est aussi appelée délégation hiérarchique. A est composé de B si : A est client de B, si A et B sont sur des niveaux conceptuels différents, si la création (resp. destruction) d'une instance de A implique la création (resp. destruction) d'une instance de B et si une instance b de B composant d'une instance a de A n'est pas accessible directement par un autre objet que a. Cette dernière règle ne s'applique qu'aux objets et une classe peut être composante de plusieurs classes distinctes.

L'**utilisation secondaire** recouvre tous les autres cas, notamment l'utilisation de classes secondaires mise en évidence pendant la phase de programmation détaillée et

qui ne sont pas significatives pour la phase de conception globale.

• **Relations d'héritage**

MECANO propose six interprétations de l'héritage. Cette taxonomie permet d'enrichir le schéma et elle est mise à profit dans les outils associés à la méthode.

- B **spécialise** A : B raffine le concept exprimé par A et exporte toutes les caractéristiques exportées par A auxquelles s'ajoutent de nouvelles caractéristiques.
- B **réalise** A : B est une classe concrète qui décrit une réalisation particulière d'une classe virtuelle A, l'interface de B est identique à celle de A.
- B **adapte** A : B renomme et/ou redéfinit certaines propriétés de A sans en ajouter de nouvelles.
- B **implante** A : les caractéristiques de la classe concrète B sont utilisées uniquement de manière interne dans A pour implanter ses méthodes et ne sont pas exportées par A.
- B **se comporte comme** A : la classe virtuelle A décrit des propriétés (ensemble de méthodes virtuelles) qui doivent être redéfinies et exportées par la classe B qui peut être virtuelle. Cette relation d'héritage est utilisée dans la généricité contrainte.
- B **fusionne** A1 et A2 : B hérite de deux classes A1 et A2 indépendantes (sans ancêtre commun) sans ajouter de nouvelles propriétés.

MECANO permet l'héritage multiple avec combinaison des différentes interprétations. Seules les relations de spécialisation, de réalisation et d'adaptation sont considérées comme des relations de sous-typage et permettent le polymorphisme. Les autres formes d'héritage sont des relations de réutilisation.

• **Concepts structurants**

MECANO propose quatre concepts de granularité supérieure à celle de la classe pour représenter des objets structurés : le domaine, le composite, l'application et le projet.

Un **domaine** est constitué à partir des trois catégories d'héritage : spécialisation, réalisation et adaptation. Il regroupe les différentes variations d'un même concept initial. Un domaine est constitué : d'une classe virtuelle représentant le concept initial du domaine, d'un ensemble de classes virtuelles spécialisant le concept initial et d'un niveau de classes concrètes réalisant des classes virtuelles ou adaptant des classes concrètes.

Le **composite** est un objet non atomique qui peut être perçu comme un objet unique à un certain niveau d'abstraction. Un composite est constitué d'un graphe de classes reliées par des liens de composition. A l'exécution, seul l'objet composite peut accéder à ses parties composantes qui sont donc masquées pour les autres objets. Le composite constitue un axe d'abstraction orthogonal aux domaines.

Une **application** est un graphe connexe de classes et de composites provenant de plusieurs domaines. Une application possède une classe racine instanciée au lancement, cette classe n'a pas de client. Les applications sont mises en place le plus tard possible dans le processus de conception pour retarder le choix de la structure de contrôle et de l'interface utilisateur du système, parties les plus exposées aux modifications et aux évolutions.

Le **projet** est l'entité racine de toute conception est regroupe les applications et les domaines du système en cours de développement.

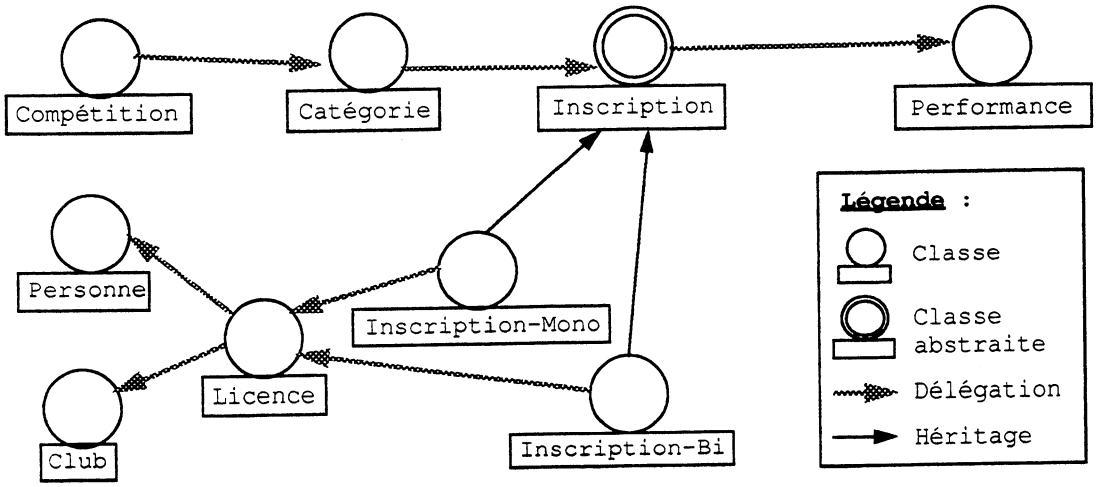


Figure 3-7 : Un schéma MECANO.

4.5.2 Démarche

La démarche préconisée par MECANO comporte cinq étapes principales :

- A - Identification des domaines : un domaine par concept identifié à l'analyse ; le domaine peut être réduit à deux classes : une classe virtuelle réalisée par une classe concrète.
- B - Création des domaines : 1) Définition du concept le plus général possible pour la racine, 2) Création des liens de spécialisation et définition des classes spécialisées, 3) Création des liens de réalisation et des classes concrètes.
- C - Retour à l'étape A avec les classes définies en B.
- D - Mise en place des applications : 1) Création de la classe racine et éventuellement de classes supplémentaires, 2) Définition des liens de communication, 3) Conception détaillée de la classe racine et des nouvelles classes.
- E - Conception détaillée des classes concrètes : 1) Ajout d'héritage de comportement ou de réalisation, 2) Définition des corps des méthodes exportées, 3) Définition des corps des méthodes internes, 4) Définition des attributs utilitaires.

4.5.3 Outils

Le poste de conception MECANO constitue un atelier de modélisation de logiciel et permet la navigation dans le schéma et la recherche sur critères, l'édition, l'affichage de différentes vues calculées ou définies par le concepteur. L'outil contrôle le respect de la syntaxe et il prend en charge le stockage des éléments de conception.

4.5.4 Remarques

La méthode MECANO est adaptée à la conception de programmes et les différentes catégories d'héritage, de relations client/fournisseur et les concepts structurants permettent d'enrichir le schéma d'un programme.

MECANO n'offre aucune facilité pour la représentation et la gestion du dialogue avec l'utilisateur. MECANO ne prend pas non plus en compte la persistance des objets.

4.6 Synthèse

Le tableau suivant résume les principales caractéristiques des méthodes présentées (cf. figure 3-8).

Comme nous l'avons constaté au cours de cette brève étude, les méthodes présentées n'apportent pas de solution adaptée au développement d'un système d'applications à l'aide d'un langage de programmation de base d'objets. En particulier, l'élaboration du gestionnaire de dialogue et la gestion des données persistantes sont rarement prises en compte. Seule OOA/OOD de P. Coad et E. Yourdon prend explicitement en compte les trois dimensions d'un système d'applications à travers un modèle d'architecture constitué de quatre composants spécialisés. Cependant nous avons noté les insuffisances de cette méthode au sujet des communications entre les différents composants. Par ailleurs, l'organisation du composant «gestion de la base de données» n'est pas approfondie et n'apporte donc pas de réponse à la structuration d'une base de données gérée à l'aide d'un SGBDOO.

	OOD 91	OOA/OOD	Class/Resp.	OMT	Mecano
Phases du cycle de vie	Conception	Analyse/ Conception	Conception	Analyse/ Conception	Conception
Type d'applications	Temps Réel	Toutes	?	Toutes	Toutes
Systèmes cibles	ADA (LOO)	LOO	LOO	LOO	LOO
Modèle d'Architecture	non	4 composants	non	non	non
Composant Fonctionnel	oui	oui	oui	oui	oui
Composant Base de données	spécification au niveau logique précisions dans le composant fonctionnel	1 composant spécialisé pas de concepts spécifiques	non	non	non
Composant Gestion du Dialogue	non	1 composant spécialisé pas de concepts spécifiques	non	non	non
Différents Schémas ou Diagrammes	Diagrammes de classes, d'objets et de transitions Diag. de modules, de processus	Diagramme de classes	Diagramme de classes	Sch. objet, Sch. dynamique, Sch. fonctionnel	Diagramme de classes
Différentes catégories de classes	classe, classe utilitaire	classe, classe&objets	classe	classe	classe, classe abstraite, composite
Différentes relations structurelles	relation	relation, tout/partie	partie-de	associations binaires ou n-aires agrégation	communication, composition, utilisation second
Différentes relations fonctionnelles	message	message	message	diagramme de flots de données	communication, composition, utilisation second
Expression de la dynamique	diagramme de transitions	diagramme de transitions	-	diagramme de transitions	assertions
Différentes relations d'héritage	héritage, métaclasse	héritage	sorte-de, analogue-à	héritage	spécialise, réalise, adapte, implante, fusionne, se comporte comme

Figure 3-8 : Tableau comparatif des méthodes de conception orientées objets.

5 Conclusion

Ce chapitre nous a permis de présenter en détail les techniques à objets et d'en proposer différentes utilisations et interprétations en fonction du point de vue adopté. Nous avons également détaillé les principes de la conception dirigée par les objets.

La présentation de quelques méthodes de conception intégrant à la fois les techniques à objets et les principes de la conception dirigée par les objets met en évidence leur inadéquation dans notre contexte.

Dans le chapitre précédent, nous avions relevé les insuffisances des méthodes d'analyse et de conception de systèmes d'information pour développer un système d'applications à l'aide d'un langage de programmation de bases d'objets.

Les chapitres qui suivent sont consacrés à la présentation de nos propositions.

Chapitre 4

MOSAÏC : Modèle d'architecture

1 Introduction

Un modèle d'architecture est un savoir-faire matérialisé par un canevas qui décrit les principaux composants d'une application et qui fixe les communications permises ou nécessaires entre ces composants. Ce canevas s'apparente à une application générique à partir de laquelle le concepteur peut dériver une application particulière. Les méthodes de conception de logiciels ou de systèmes d'information, qu'elles soient conventionnelles ou «orientées objets», sont implicitement basées sur une séparation entre les données persistantes, gérées par un SGBD ou un SGF, et les fonctions, programmées à l'aide d'un langage de programmation. Les méthodes de conception de systèmes interactifs reposent sur des modèles explicites d'architecture qui distinguent le corps d'une application (composant fonctionnel) et l'interface interactive. Ces modèles d'architectures organisent l'interface interactive en trois composants (l'adaptateur, le gestionnaire de dialogue et le composant d'interaction physique) mais ils restent généralement imprécis au sujet de la structuration du corps de l'application.

Ce chapitre est consacré à la présentation du modèle d'architecture de la méthode MOSAÏC qui regroupe nos propositions pour concevoir un système d'applications programmé à l'aide d'un Langage de Programmation de Bases d'Objets. Ce modèle d'architecture concerne seulement la phase de conception et il prend en compte trois aspects des systèmes d'applications : les données persistantes, les fonctions et la gestion du dialogue homme-machine. Ce chapitre nous permet de mettre en évidence différentes catégories de classes et d'en fixer leurs rôles. Ce dernier point est nécessaire et essentiel pour déterminer les critères de qualité à appliquer aux classes et ainsi faciliter une réponse à la question du choix des «bonnes» classes. Nous supposons que la phase de conception à laquelle nous nous intéressons a été précédée par une phase d'analyse qui a permis d'élaborer trois documents : les spécifications fonctionnelles de l'application, les spécifications des données persistantes et les spécifications externes décrivant le dialogue. Les formes de ces spécifications peuvent être variées : graphiques, textuelles, formelles, etc.

Dans le chapitre suivant (5) nous précisons les concepts et les langages proposés par MOSAÏC pour concevoir un système d'applications.

2 Décomposition initiale d'une application MOSAïC

- **Définition 1 -** Une application est décomposée en deux parties principales : le corps de l'application et l'interface interactive. Le **corps** contient toute la connaissance, informations et fonctions, relative au domaine. L'**interface interactive** est chargée de gérer le dialogue entre l'utilisateur et le corps de l'application.

Le corps d'une application contient d'une part les procédures et les données temporaires utiles pour réaliser les fonctions de l'application et programmées à l'aide d'un langage de programmation, et d'autre part les données persistantes, gérées par un SGF ou par un SGBD. Les données temporaires sont utilisées pour réaliser les fonctions et leur durée de vie est contenue dans celle de l'application. On notera que le corps de l'application n'effectue jamais d'échange direct avec l'utilisateur.

L'interface interactive est chargée de toutes les opérations d'entrée/sortie avec l'utilisateur. Les données qu'elle contient sont des données généralement temporaires utilisées pour réaliser les fonctions de gestion du dialogue.

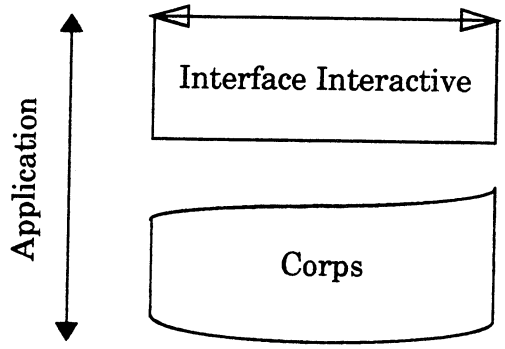


Figure 4-1 : Décomposition initiale d'une application.

- **Règle 1 -** L'interface interactive ne doit contenir aucune connaissance relative au domaine d'application, de sorte que la modification ou la suppression d'une partie de l'interface n'entraîne pas de perte de connaissances sur le domaine d'application (données ou fonctions).

Le contrôle du dialogue doit autant que possible être localisé dans l'interface interactive. Sauf cas exceptionnel, l'initiative du dialogue doit être laissée à l'utilisateur. Cependant, dans certains cas (opération délicate, demande d'informations supplémentaires à l'utilisateur, etc.), le corps peut être amené à prendre ponctuellement

et temporairement l'initiative du dialogue. Même dans ce cas, l'interrogation de l'utilisateur est réalisée via l'interface interactive. Le corps de l'application a donc un rôle de serveur fonctionnel et de serveur de données pour l'interface interactive.

3 Classes applicatives et corps d'une application

3.1 Classes applicatives

- **Définition 2 -** La classe est l'unique concept utilisé pour structurer le corps d'une application. Le corps d'une application est défini par un ensemble de classes dites **classes applicatives.** Une classe applicative décrit un modèle d'**objets applicatifs.**

Une classe peut représenter un composant modulaire d'un programme ou un composant d'un schéma de base de données (cf. chapitre 2). Conceptuellement, la notion de classe est donc suffisante pour structurer le contenu du corps d'une application.

Physiquement, un Langage de Programmation de Bases d'Objets propose un langage pour décrire les programmes et les données persistantes. Ce langage, basé sur la notion d'objet, a une capacité de structuration voisine de celle des langages de programmation à objets actuels. Le recours à un autre langage de programmation que celui proposé par le LPBO ne nous semble donc pas justifié.

Pour définir la persistance, nous adoptons le principe d'**indépendance entre la persistance et la notion de classe** [AB87].

- **Règle 2 -** Pendant l'exécution d'une application, des objets persistants et des objets temporaires instances d'une même classe applicative peuvent cohabiter.

- **Règle 3 -** Une classe décrit un modèle d'objets mais ne regroupe pas les objets générés ; il y a indépendance entre la notion de classe et la notion d'ensemble d'objets.

3.2 Architecture du corps d'une application

- **Règle 4 -** Les classes applicatives du corps d'une application sont élaborées par un processus de modélisation des entités du domaine.

Les méthodes systémiques d'analyse et de conception de systèmes d'information préconisent de construire un système d'information comme un modèle du système opérant de l'organisation considérée. La conception d'une base de données s'appuie de façon similaire sur un modèle du domaine dans lequel les entités perçues constituent le critère de décomposition. De même, les méthodes actuelles de conception de logiciels s'appuient sur une conception dirigée par les objets du domaine. Ce critère de décomposition conduit à une structure logicielle généralement plus invariante et plus aisée à maintenir que les structures obtenues à l'aide d'autres critères (décomposition fonctionnelle, décomposition selon les structures de données, etc.).

3.3 Composition des classes applicatives

- **Définition 3 -** Un objet applicatif constitue à la fois un module logiciel d'une application et une unité de mémorisation de données persistantes. Il est composé à la fois de données «potentiellement persistantes», de données temporaires et de méthodes. Ses méthodes peuvent utiliser ces deux catégories de données.

L'étape de conception doit permettre de fixer les principaux éléments de la solution technique : le choix de la représentation des données persistantes et le choix de la réalisation des fonctions (algorithmes, données temporaires et données persistantes). Le bon fonctionnement du corps d'une application nécessite la cohabitation de données persistantes, de traitements et de données temporaires utilisées pour réaliser ces traitements. On peut envisager trois organisations-types pour structurer le corps d'une application à l'aide de classes applicatives (cf. figure 4-2) :

- Une décomposition conventionnelle dans laquelle on distingue deux «sous-composants» constitués de classes applicatives. Le premier représente la base de données et ses classes applicatives décrivent seulement des données persistantes. Le second contient des fonctions spécifiques au domaine d'application et des données temporaires organisées en classes applicatives.
- Une solution distinguant également deux «sous-composants» et donc deux catégories de classes applicatives. Cependant dans cette solution, les classes applicatives décrivant les données persistantes contiennent une partie des fonctions sous forme de méthodes pour maintenir la cohérence des données persistantes.

- Une architecture dans laquelle chaque objet contient à la fois des données «potentiellement persistantes», des données temporaires et des méthodes. Dans ce cas les classes applicatives ne sont plus distinguées a priori sur la nature des données qu'elles contiennent et elles peuvent être élaborées en s'appuyant sur un processus de modélisation du domaine.

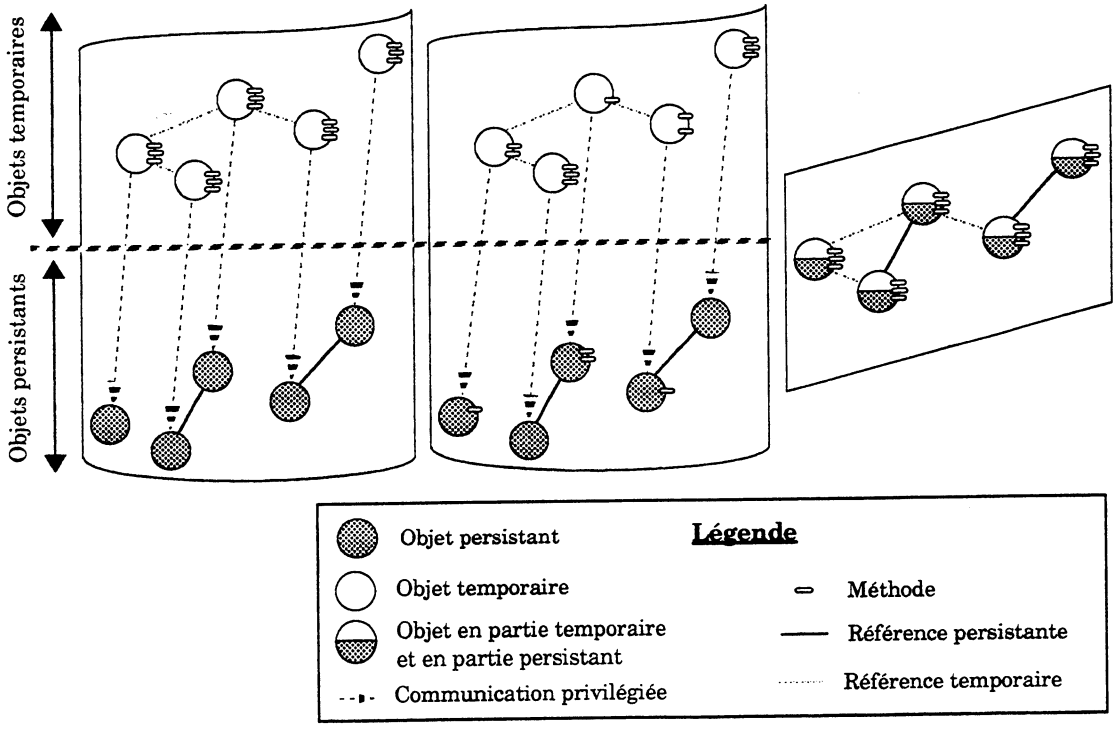


Figure 4-2 : Trois organisations du corps d'une application.

Les deux premières solutions reviennent à distinguer totalement ou partiellement deux composants : la «base de données» et le «programme». Comme ces deux composants sont constitués de classes applicatives obtenues dans les deux cas à partir d'un modèle du domaine, les classes applicatives des deux composants sont fortement corrélées deux à deux. A chaque classe d'entités du domaine correspondent deux classes applicatives : l'une décrit les données persistantes et l'autre décrit les fonctions et les données temporaires nécessaires à ces fonctions. Cette forte corrélation induit un échange dense de messages au sein de chaque couple d'objets et entraîne un affaiblissement du masquage d'informations. On constate donc que la séparation persistant/temporaire engendre un fort couplage des classes deux à deux et une baisse de la cohésion de chaque classe. Par ailleurs, l'encapsulation et le masquage d'informations ne peuvent être appliqués que partiellement.

La troisième solution conduit à des classes à forte cohésion et permet de limiter le couplage. Cette solution permet également d'exploiter plus complètement les principes d'encapsulation et de masquage d'informations. Dans une telle approche, le corps de

l'application peut être conçu directement comme un modèle du domaine dans lequel chaque objet est abordé selon deux points de vue complémentaires : un point de vue «programmation» et un point de vue «base de données».

D'un point de vue conceptuel, cette troisième solution, que nous avons retenue dans MOSAÏC, est plus satisfaisante. Mais physiquement, les LPBO sont «partagés» quant à l'indépendance entre les mécanismes d'encapsulation et de persistance. Dans le SGBDOO O2 [ODeu89], la persistance est indépendante de la notion de classe mais en revanche, la persistance d'un objet entraîne la persistance de toutes ses données. La programmation en O2 d'une application conçue selon la troisième approche nécessite donc une décomposition de chaque classe applicative en deux classes selon d'une part les aspects persistants et d'autre part les aspects temporaires pour aboutir à une solution voisine de la seconde organisation. Dans le langage O++ [BGJR92] en revanche, une classe peut définir à la fois des données temporaires et des données «potentiellement persistantes». Ce langage permet donc une programmation plus directe d'une application conçue selon le modèle d'architecture retenu.

On constate que contrairement aux architectures conventionnelles qui séparent conceptuellement et physiquement la base de données et les fonctions spécifiques de l'application, notre modèle d'architecture les réunit pour proposer une organisation élaborée autour d'un modèle du domaine d'application.

L'exemple 4-1 décrit des classes applicatives contenant des données temporaires et des données «potentiellement persistantes».

```
1     classe applicative Compétition
2       attributs
3          nom_compétition : Chaîne                  (*persistant*)
4          catégories : Liste (Catégorie)            (*persistant*)
5          liste_dossards : Liste (Inscription)...   (*temporaire*)
6     fin
7
8     classe applicative Catégorie
9       attributs
10         nom_catégorie : Chaîne                    (*persistant*)
11         inscrits : Ensemble (Inscription)         (*persistant*)
12         classement : Liste (Inscription)...       (*temporaire*)
13    fin
14
15    classe applicative Inscription
16      attributs
17         numéro_inscription : Entier               (*persistant*)
18         nom : Chaîne                              (*persistant*)
19         prénom : Chaîne                           (*persistant*)
20         club : Chaîne                             (*persistant*)
21         dossard : Entier                          (*persistant*)
22         résultat : Réel...                        (*persistant*)
23    fin
```

Exemple 4-1 : Description de trois classes applicatives.

Dans l'exemple 4-1, le classement d'une catégorie (ligne 12) peut être calculé à partir des inscrits de la catégorie (ligne 11) et du résultat de chaque inscription (ligne 22).

Cependant, pour des raisons d'efficacité, nous avons choisi de stocker temporairement ce classement pendant l'exécution de l'application (ligne 12). Par contre, pour ne pas introduire de redondance dans les données persistantes, cet attribut n'est pas persistant. La liste des dossards (ligne 5), calculée à partir des lignes 4, 11 et 21, est également stockée temporairement pendant l'exécution pour des raisons d'efficacité.

3.4 Distinction entre attributs et variables

- **Définition 4 -** Les données d'une classe applicative sont distinguées en attributs et variables. Les **attributs** résultent du processus de modélisation du domaine et sont publics et potentiellement persistants. Les **variables** sont introduites par le choix des algorithmes retenus pour programmer les méthodes. Les variables sont privées et temporaires.

Les attributs et les variables permettent de distinguer les données qui dépendent des deux rôles distincts joués par chaque classe applicative. Les attributs, mis en évidence par le processus de modélisation du domaine, ont un niveau d'invariance voisin de celui des méthodes. Par ailleurs, ils représentent des informations que l'utilisateur doit pouvoir consulter. Les attributs doivent donc être publics. La définition de variables résulte d'un choix d'algorithme pour réaliser les méthodes. Leur niveau d'invariance est identique à celui des algorithmes ou des méthodes «utilitaires» et à ce titre elles doivent être privées.

```
classe applicative Compétition
  attributs
     nom_compétition : Chaîne
     catégories : Liste (Catégorie)
  variables
     liste_dossards : Liste (Inscription)
     classement_général : Liste (Inscription)
fin

classe applicative Catégorie
  attributs
     nom_catégorie : Chaîne
     inscrits : Ensemble (Inscription)
  variables
     classement : Liste (Inscription)
fin

classe applicative Inscription
  attributs
     numéro_inscription : Entier
     nom : Chaîne
     ...
fin
```

Exemple 4-2 : Exemples d'attributs et de variables de classes applicatives.

3.5 Méthodes primaires et méthodes secondaires

- **Définition 5 -** On distingue, dans chaque classe applicative, deux catégories de méthodes : les méthodes primaires et les méthodes secondaires. Une méthode **primaire** est conçue pour réaliser un traitement complet ; elle doit «prendre» et «laisser» les objets applicatifs dans un état cohérent. Une méthode **secondaire** réalise un traitement partiel et peut «prendre» ou «laisser» les objets applicatifs dans un état temporairement incohérent.

Les méthodes primaires sont activables indépendamment d'autres méthodes alors que les méthodes secondaires font nécessairement partie d'un traitement plus complet assurant le maintien de l'intégrité des objets applicatifs. Les méthodes secondaires résultent de la décomposition de méthodes primaires ou secondaires. Une méthode primaire comme une méthode secondaire peut appeler des méthodes primaires ou secondaires. On peut noter qu'une méthode primaire n'est pas une transaction, les transactions sont abordées dans la section 4.5 de ce chapitre.

```
1     classe applicative Compétition
2      attributs
3         nom_compétition : Chaîne
4         catégories : Liste (Catégorie)
5         ...
6
7      méthode primaire attribuer_tous_les_dossards
8         variables c : Catégorie; dos_courant : entier;
9         dos_courant :=0;
10        pour tout c dans catégories faire
11           dos_courant := c.attribuer_les_dossards(dos_courant);
12        fpour
13     fin
14    fin
15
16    classe applicative Catégorie
17     attributs
18        nom_catégorie : Chaîne
19        inscrits : Ensemble (Inscription)
20        ...
21
22     méthode secondaire attribuer_les_dossards(dos_courant:Entier):Entier
23        variables i : Inscription;
24        pour tout i dans inscrits faire
25           dos_courant := dos_courant + 1;
26           i.affecter_dossard(dos_courant);
27        fpour;
28        retourner dos_courant;
29     fin
30    fin
31
32    classe applicative Inscription
33     attributs
34        numéro_inscription : Entier
35        nom : Chaîne
36        dossard : Entier
```

```
37          ...
38
39      méthode secondaire affecter_dossard(dos : Entier)
40          dossard := dos fin
41      fin
```

Exemple 4-3 : Méthodes primaires et méthodes secondaires.

Dans l'exemple 4-3, les méthodes des lignes 22 et 39 sont secondaires, elles ont été élaborées pour permettre de réaliser la méthode primaire de la ligne 7. Ces méthodes secondaires ne réalisent que des traitements localisés et partiels par rapport au traitement complet réalisé par la méthode primaire.

4 Classes interactives et interface d'une application

4.1 Classes interactives

- **Règle 5 -** La classe est l'unique concept utilisé pour structurer l'interface interactive. Le dialogue entre l'application et l'utilisateur est donc géré par un ensemble d'**objets interactifs** instances de **classes interactives.**

- **Règle 6 -** Les objets interactifs sont toujours temporaires.

Au niveau conceptuel, nous avons évoqué dans le chapitre 2 les ressemblances entre les «agents» des modèles multiagents et les objets des modèles à objets. La notion de classe est bien adaptée pour concevoir puis programmer la gestion du dialogue.

Au niveau physique, trois raisons militent en faveur d'une prise en charge de l'interface interactive par le même logiciel de base que celui utilisé pour le corps de l'application. Tout d'abord, la programmation de l'interface à l'aide d'un système distinct de celui utilisé pour le corps nécessite la connaissance de deux systèmes et de deux langages différents. Ensuite, les échanges entre le corps et l'interface interactive d'une application sont nombreux. Ils se déroulent le plus souvent de l'interface vers le corps mais également dans certains cas (demande d'informations complémentaires, etc.) du corps vers l'interface. Le bon déroulement de ces échanges dépend des services de communications bi-directionnelles entre les logiciels de base utilisés pour réaliser ces deux composants (interface et corps de l'application). Ce problème ne se pose pas si ces deux composants sont programmés à l'aide du même logiciel de base. Enfin, les LPBO offrent souvent des classes spécialisées dans la gestion de l'interaction homme-machine permettant de programmer l'interface interactive et le corps de l'application à l'aide du même langage.

4.2 Composition des classes interactives

- **Définition 6 -** Chaque classe interactive est composée d'une partie contrôle et d'une partie présentation. La **présentation** décrit un sous-ensemble des informations affichées à l'écran et un sous-ensemble des fonctions activables par l'utilisateur. Le **contrôle** assure la gestion du dialogue ainsi que le maintien de la cohérence des informations affichées.

Les attributs de la partie présentation représentent des informations affichées à l'écran. Les méthodes de la partie présentation décrivent des actions activables par l'utilisateur (item dans un menu, bouton, etc.).

Dans le modèle PAC [Cout90], la gestion du dialogue est réalisée par des agents PAC composés de trois parties : la présentation, l'abstraction et le contrôle. L'abstraction représentent une vue abstraite des concepts présentés à l'utilisateur par la présentation. Dans le cas des systèmes d'applications, deux remarques peuvent être faites :

- L'utilisation pratique du modèle PAC dans notre contexte montre qu'il y a peu de différences entre les données présentes dans l'abstraction et celles affichées par la présentation. Or dans notre contexte, ces informations peuvent être très volumineuses.
- Dans les applications de nature base de données les informations présentées à l'utilisateur sont issues directement du corps de l'application. Dans le contexte dans lequel nous nous plaçons, le formalisme dans lequel les informations sont présentées à l'utilisateur est en général très voisin de la représentation interne de ces mêmes informations dans le corps. Il y a donc à nouveau redondance entre les informations présentes dans le corps et celles présentes dans l'abstraction.

Pour limiter la redondance, nous n'avons pas retenu le composant abstraction. Les informations affichées par la présentation sont donc les informations présentes dans le corps de l'application. Cependant, pour des raisons d'efficacité, une telle redondance pourra être introduite au moment de la programmation effective de l'application. Les échanges entre le corps et l'interface interactive sont décrits plus complètement dans les sections suivantes.

Les classes interactives se distinguent du mécanisme de vue des SGBD relationnels par leur dimension dynamique et leur rôle de composant logiciel actif dans la gestion du dialogue. Le mécanisme de vue permet seulement une restructuration virtuelle et un filtrage portant exclusivement sur des données.

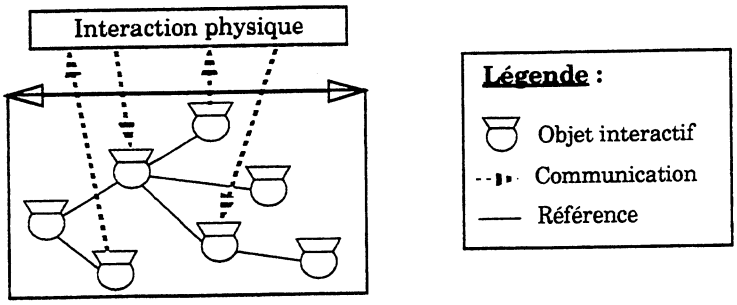


Figure 4-3 : Décomposition de l'interface interactive en objets applicatifs.

Le choix de conception que nous avons retenu dans cette section ne remet pas en cause la nécessité, pour le logiciel de base, d'offrir des ouvertures vers d'autres langages (C, C++, etc.). En pratique, par exemple dans une entreprise, de tels services sont nécessaires pour exploiter des applications développées antérieurement ou pour permettre d'utiliser des outils de développement existants (générateurs d'interfaces, etc.). Par ailleurs, le corps et l'interface peuvent être conçus selon le modèle que nous préconisons, puis programmés à l'aide de deux logiciels de base distincts si les services de communications entre ces deux systèmes le permettent. Cette solution permet de développer des interfaces conviviales pour des LPBO n'offrant pas les mécanismes et les classes nécessaires à la programmation de telles interfaces.

4.3 Processus d'élaboration des classes interactives

- **Règle 7 -** Les classes interactives sont obtenues selon un processus de conception dirigée par les objets appliqué au domaine de l'interaction entre l'utilisateur et l'application, c'est-à-dire aux spécifications externes.

L'élaboration des classes applicatives s'appuie sur un modèle du domaine réel et sur les spécifications des données et des fonctions d'une application. L'élaboration des classes interactives s'appuie sur les spécifications externes d'une application qui détaillent le contenu et la forme du dialogue entre l'utilisateur et cette application. Les classes interactives ne sont pas indépendantes des classes applicatives. Les classes interactives présentent à l'utilisateur des informations et des fonctions contenues dans les classes applicatives constituant le corps de l'application. L'expérience montre qu'il existe rarement une bijection entre les classes interactives et les classes applicatives. Une classe interactive utilise des données et des méthodes réparties souvent dans plusieurs classes applicatives différentes. Inversement, les caractéristiques d'une classe applicative peuvent être utilisées et présentées à l'utilisateur par différentes classes interactives.

```
classe interactive Menu_gestion_course
  facette présentation (*Menu*)
     méthodes
        résultat, afficher_résultat_catégorie,
        imprimer_résultat_catégorie...
  fin

                (*contrôle*)
  attributs
     résultat_courant : Saisie_résultat
     catégorie_affichée : Résultat_d'une_catégorie...
fin

classe interactive Saisie_résultat
  facette présentation (*Fenêtre*)
     attributs
        dossard :Entier          (*Dossard*)
        nom_coureur : Chaîne     (*Nom de la catégorie à afficher*)
        prénom_coureur : Chaîne
        résultat : Réel
     méthodes
        valider, annuler...
  fin

                (*contrôle*)
  attributs
     inscription : Inscription...
fin

classe interactive Résultat_d'une_catégorie
  présentation (*Fenêtre*)
     attributs
        nom_compétition, date_compétition : Chaîne,
        nom_catégorie : Chaîne
        liste_résultat : Liste (Nuplet (nom, prénom : Chaîne, résultat : Réel))
  fin

                (*contrôle*)
  méthodes
     afficher_catégorie (nom_catégorie)
     mise_à_jour
fin
```

Exemple 4-4 : Description de trois classes interactives.

La partie contrôle de la classe `Menu_gestion_course` indique par ses attributs la structuration des classes qui gèrent l'affichage pendant le déroulement de la compétition. Les informations présentées par la classe interactive `Résultat_d'une_catégorie` proviennent des classes applicatives `Compétition`, `Catégorie` et `Inscription` (cf. Exemple 4-2).

4.4 Communications entre classes applicatives et classes interactives

- **Règle 8 -** Une classe applicative peut directement référencer par ses attributs des classes applicatives et une classe applicative peut envoyer des messages directement à des classes applicatives. Une classe applicative ne peut pas référencer une classe interactive ni communiquer avec elle. Les classes applicatives peuvent interroger l'interface interactive seulement par un ensemble limité de primitives qui masquent la structure de l'interface interactive.

Dans les modèles d'architecture adaptés aux systèmes interactifs, le composant adaptateur assure l'interface entre le corps et le gestionnaire de dialogue. Le rôle de l'adaptateur est double : il assure l'indépendance entre le corps de l'application et le gestionnaire de dialogue et il assure le changement de formalisme entre ces deux composants. Dans notre contexte, on peut faire les remarques suivantes :

- Dans certaines applications interactives, les concepts manipulés dans le corps de l'application sont très éloignés des concepts présentés à l'utilisateur par l'interface. Un effort important de traduction est donc nécessaire. En revanche, dans les applications qui nous intéressent, les informations présentées à l'utilisateur sont la plupart du temps très proches des informations manipulées dans le corps de l'application.
- La structure du corps d'une application, organisée autour des abstractions des entités du domaine, est plus invariante que la structure de l'interface organisée autour d'une présentation particulière de ces entités. La modification d'une classe applicative, qui reflète une évolution de la perception du domaine ou une modification des fonctions de l'application, a généralement des répercussions sur l'interface. Par contre, la modification d'une présentation particulière peut affecter une ou plusieurs classes interactives mais n'a aucune influence sur les classes applicatives.

Pour la communication de l'interface interactive vers le corps, le composant adaptateur n'a plus sa place. En revanche, dans l'autre sens, compte tenu des niveaux d'invariance respectifs du corps de l'application et de son interface interactive, il est nécessaire de masquer la structure de l'interface interactive. Cependant nous n'introduisons pas de concept particulier pour concrétiser ce masquage qui peut être aisément réalisé à l'aide des mécanismes de base des techniques à objets : l'héritage, le sous-typage et la substitution (cf. chapitre 5 section 9.5).

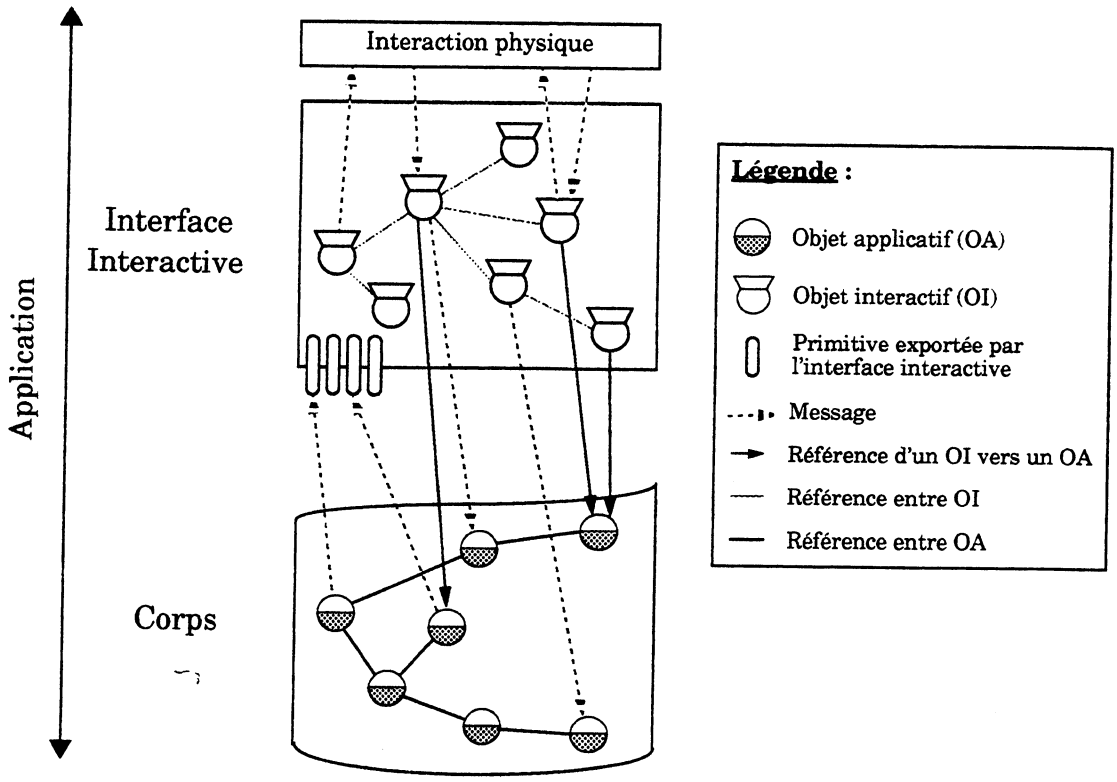


Figure 4-4 : Communications entre le corps d'une application et son interface interactive.

La localisation du contrôle du dialogue dans l'interface interactive permet de laisser autant que possible l'initiative de ce dialogue à l'utilisateur. Cependant, il est nécessaire dans certains cas que le corps puisse prendre l'initiative du dialogue. Par exemple lorsque tous les paramètres nécessaires à un traitement ne peuvent pas être déterminés avant le lancement de ce traitement et dépendent de l'exécution du traitement lui-même : le corps de l'application doit alors interroger l'utilisateur pour obtenir ces paramètres manquants.

4.5 Restrictions des communications

- **Règle 9 -** Les appels de méthodes réalisés par les classes interactives vers les classes applicatives sont limités aux méthodes primaires de ces dernières. Les méthodes secondaires des classes applicatives ne peuvent pas être utilisées par les classes interactives.

Une méthode primaire réalise un traitement complet et doit garantir le maintien de l'intégrité des objets applicatifs alors que l'exécution d'une méthode secondaire peut introduire des incohérences dans ces objets. Pour ces raisons, l'exécution d'une méthode

secondaire doit être précédée ou suivie par l'activation d'autres méthodes primaires ou secondaires pour réaliser globalement un traitement cohérent. L'enchaînement des méthodes à exécuter pour réaliser un traitement complet et cohérent est une connaissance qui se rapporte au domaine d'application. A ce titre, cet enchaînement de méthodes doit être programmé dans une méthode primaire d'une classe applicative.

Supposons que l'on autorise une méthode d'une classe interactive à appeler une méthode secondaire d'une classe interactive. Dans ce cas, l'interface interactive doit réaliser la séquence de méthodes qui préserve la cohérence des objets applicatifs. La programmation de cette séquence dans une méthode d'une classe interactive représente un transfert de connaissance du corps vers l'interface. Ce transfert est incompatible avec une bonne séparation des rôles des deux composants fondamentaux d'une application et nuit à l'évolutivité de l'application.

La notion de transaction permet de définir une séquence atomique d'instructions qui fait évoluer une base de données d'un état cohérent vers un autre état cohérent. Si la transaction est exécutée complètement et si l'état final des données est cohérent, les modifications sont validées. Dans le cas contraire (violation de contraintes d'intégrité, exécution partielle de la transaction, etc.) les modifications effectuées sont annulées et l'état initial est restauré.

Dans les LPBO, la notion de transaction est prise en compte de façon variable. Dans O2, par exemple, une transaction est un programme activable seulement par l'utilisateur. Les objets ne peuvent pas déclencher de transaction. Dans Aristote comme dans les LPBO, la prise en compte de la notion de transaction fait encore l'objet de nombreuses recherches [Mach92].

La notion de méthode primaire peut servir de support à la définition de la notion de transaction. Un mécanisme de transaction simple, sans imbrication, est suffisant si les deux conditions suivantes sont remplies :

- La notion de transaction n'est pas associée directement à celle de méthode primaire mais à l'envoi de messages de l'interface interactive vers le corps de l'application. De cette façon, une méthode primaire peut appeler une autre méthode primaire sans déclencher une nouvelle transaction.
- Les communications dans le sens corps vers interface ne provoquent pas de nouvel envoi de message de l'interface interactive vers le corps de l'application.

Une telle définition de la notion de transaction a l'avantage de ne pas influencer la structure de l'application. Cependant la définition de la notion de transaction nécessite un travail plus approfondi.

5 Organisation d'un système d'applications

Nous avons défini un système d'applications comme un ensemble d'applications opérant dans le même domaine et partageant des informations persistantes. La notion de projet permet d'organiser et de concevoir un système d'applications.

5.1 Notion de projet

- **Définition 7 -** Un système d'applications est conçu comme un projet. Un **projet** encapsule un ensemble d'objets applicatifs persistants et différentes applications.

Un projet est un concept de granularité très supérieure à celle d'un objet mais il peut être défini par analogie à un objet : un objet encapsule des données et des méthodes; un projet encapsule des objets applicatifs persistants et des applications. L'application du principe de masquage d'informations aux objets conduit à définir une signature qui limite les opérations réalisables sur un objet. L'application du principe de masquage d'informations aux projets conduit de la même façon à définir une interface limitée par laquelle les projets peuvent communiquer entre eux.

	Objet	Projet
Encapsulation	données (attributs) traitements (méthodes)	objets applicatifs (racines de persistance) applications (racines d'application)
Masquage	signature	interface
Classification	classe (modèle d'objets)	classe projet (modèle de projets)

Figure 4-5 : Analogie objet-projet.

5.2 Notion de classe projet

- **Définition 8 -** Une **classe projet** définit un modèle de projets et elle est composée de racines de persistance et de racines d'applications. Une **racine de per-**

sistance est, pour le concepteur, un identificateur qui désigne un objet applicatif persistant. Une **racine d'application** est un identificateur qui désigne une application.

Nous avons adopté le principe d'indépendance entre la persistance et la notion de classe. La persistance est définie à partir de racines de persistance et se propage à partir des racines par les attributs persistants (les attributs temporaires ne propagent pas la persistance). Une racine d'application est un identificateur typé par une classe interactive qui désigne la classe instanciée au lancement de l'application.

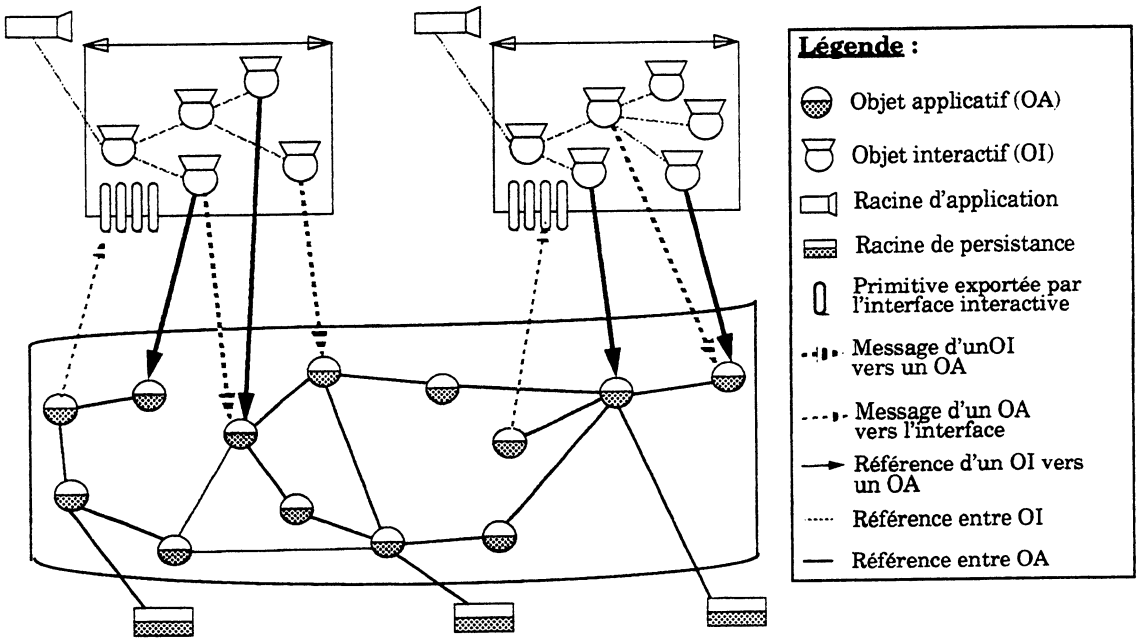


Figure 4-6 : Organisation d'un système d'applications à l'exécution.

Pour l'utilisateur, une application est un processus qu'il peut déclencher puis arrêter. Pendant son activation, l'application offre à l'utilisateur une interface conviviale gérée par des objets interactifs et elle permet d'activer des traitements réalisés par des objets applicatifs éventuellement persistants. Une application n'est donc pas définie comme un «macro-composant» bien identifié mais par une classe interactive initiale qui est instanciée lors du lancement de l'application et qui à son tour peut créer d'autres objets interactifs ou applicatifs. Cette approche de la notion d'application rejoint celle de langages tels qu'Eiffel.

Dans le modèle d'architecture proposé, les classes applicatives ont un double rôle. Chaque classe applicative est un composant modulaire d'une application ; cette classe applicative est alors utilisée essentiellement pour les méthodes qu'elle réalise. Chaque classe applicative est un constituant du schéma de la base d'objets applicatifs persistants ; cette classe applicative est alors utilisée essentiellement pour les données qu'elle

contient.

```
classe projet Gestion_Compétitions_Sportives

  racines de persistance
        compétitions : Ensemble (Compétition)
        compétition_courante : Compétition
        clubs : Ensemble (Club)
        licenciés : Ensemble (Licencié)

  racines d'application
        inscription : Init_inscription
        dossard : Init_dossard
        course : Init_course

fin_classe_projet
```

Exemple 4-5 : Description d'une classe projet.

5.3 Partage des objets et concurrence entre applications

Le partage des données persistantes est une caractéristique essentielle des systèmes d'applications. Dans notre approche, les éléments persistants sont les objets applicatifs qui assument également le rôle de composants logiciels. Partager les données persistantes demande donc de partager les objets applicatifs. Les règles et les mécanismes de partage offerts par les SGBD conventionnels, notamment les SGBD relationnels, permettent seulement de partager des données. Ils sont donc insuffisants pour notre modèle dans lequel les objets applicatifs persistants peuvent comporter à la fois des données persistantes, des données temporaires et des méthodes.

Les objets applicatifs doivent être perçus comme des ressources partagées par des processus en concurrence. Les contraintes imposées par une telle interprétation, ainsi que les mécanismes nécessaires à sa mise en œuvre doivent faire l'objet d'études plus approfondies. En particulier, les techniques développées pour la programmation concurrente doivent être prises en compte. Actuellement l'aspect multi-utilisateur n'est pas pris en compte dans MOSAÏC. Les différentes applications d'un système d'applications ne sont donc pas activées simultanément, mais se déroulent séquentiellement.

5.4 Composition des classes projets

La notion de projet telle que nous l'avons définie dans la section 5.1 peut être qualifiée de projet atomique et constitue un cas particulier d'un concept plus général utile pour structurer un système d'applications.

Un projet vérifie le principe de masquage d'informations et il est composé d'un corps et d'une interface. On distingue les projets atomiques et les projets composés dont les corps sont composés d'autres projets. L'interface d'un projet masque le corps de ce projet pour les clients du projet. Un projet composé est avant tout un concept logique

utile pour décomposer un système d'applications complexe en sous-systèmes. L'interface d'un projet est alors un moyen de définir, de façon abstraite, les communications nécessaires entre les projets.

La notion de projet composé peut être utile dans différentes situations :

- La notion de projet composé permet de limiter le nombre d'applications travaillant sur la même base d'objets applicatifs. De cette façon, la complexité des objets applicatifs, composants modulaires de ces applications, peut être maîtrisée.
- Lors du développement d'un système d'applications, il est rare de ne pas devoir tenir compte des développements antérieurs réalisés dans l'organisation, notamment dans le cas des systèmes d'information. L'intégration de ces applications existantes doit pouvoir être prise en compte dès l'étape de conception pour garantir l'homogénéité et le bon fonctionnement du système. La notion de projet permet d'intégrer ces développements antérieurs en les considérant comme encapsulés dans des projets.
- La structure même des organisations a évolué : d'une structure fortement hiérarchique, les entreprises sont passées à une organisation modulaire dans laquelle les activités, mais également les décisions et les ressources sont décentralisées [Stra88]. Cette modularisation a eu pour conséquence une dispersion géographique favorisée par le développement des communications ordinateur/ordinateur tant locales que distantes. La conception d'un système d'information doit prendre en compte cette nouvelle réalité et permettre d'identifier et de spécifier les besoins en communication entre les différents systèmes informatiques décentralisés.

On peut souligner l'analogie existante entre la situation des langages de programmation dans les années 70 et celle des systèmes de gestion de bases de données des années 90. Actuellement, de nombreux programmes peuvent accéder librement à une même base de données. A la fin des années 70, les fonctions d'un programme écrit dans un langage procédural pouvaient accéder librement aux variables globales de ce programme. La nécessité de protection des variables globales et le besoin de modularité des programmes a donné naissance à des composants modulaires encapsulant des variables et des fonctions et offrant un mécanisme de masquage d'informations. Les modules de Modula-2, les paquetages d'ADA ou les classes des langages à objets en sont des exemples. La notion de projet permet de façon similaire de protéger les données persistantes et de maîtriser la complexité toujours croissante des applications informatiques.

Dans ce cadre, le SGBDOO O_2 propose les concepts de «schéma» et de «base». Un schéma regroupe la définition de classes, d'applications et d'objets nommés. Une base est une instance de schéma et regroupe un ensemble d'objets persistants. Les classes définies dans un schéma sont par défaut masquées pour les schémas clients mais elles peuvent être explicitement exportées. Une classe projet MOSAÏC se distingue d'un schéma O_2 selon trois points :

- Une classe projet peut être composée d'autres classes projets permettant ainsi de structurer une application informatique.
- L'interface d'un projet permet de définir de façon abstraite les communications dans les deux sens entre le corps d'un projet et l'environnement de ce proje.
- Une classe projet n'encapsule pas les définitions des classes applicatives ou interactives qui peuvent être réutilisées librement dans différentes classes projets.

De nombreuses méthodes de conception dirigées par les objets offrent des concepts pour structurer l'ensemble des classes décrivant un programme. Ces concepts sont généralement à rapprocher de la notion de bibliothèque regroupant statiquement un ensemble de classes. Ils ne constituent pas, à la différence des modules ou des schémas, un niveau d'abstraction supplémentaire.

La notion de projet composé est importante mais nécessite une étude complémentaire, notamment pour déterminer la nature des liens pouvant exister entre des projets participants à un projet composé.

6 Conclusion

Dans ce chapitre nous avons introduit le modèle général d'architecture de MOSAÏC. Ce modèle constitue une forme d'application générique utilisable par le concepteur pour concevoir des systèmes d'applications. Nous pouvons résumer les particularités de ce modèle d'architecture (cf. figure 4-7) :

- Un système d'applications est conçu comme un projet qui encapsule des objets persistants et des applications.
- Une application est décomposée en classes applicatives et classes interactives qui communiquent de façon asymétrique et limitée.
- Le rôle des classes applicatives est double : ce sont à la fois des composants modulaires des applications et des constituants du schéma des informations persistantes.
- Les classes applicatives sont composées d'attributs persistants et de variables temporaires. Les méthodes sont distinguées en méthodes primaires et méthodes secondaires.
- Une classe applicative peut générer des objets persistants et des objets temporaires.
- Les classes interactives sont composées d'une partie présentation et d'une partie contrôle. Les objets interactifs sont toujours temporaires.
- Le processus d'élaboration des classes est celui d'une conception dirigée par les objets. Les classes applicatives sont des modèles des entités du domaine. Les classes interactives sont des modèles des entités du dialogue homme-machine.

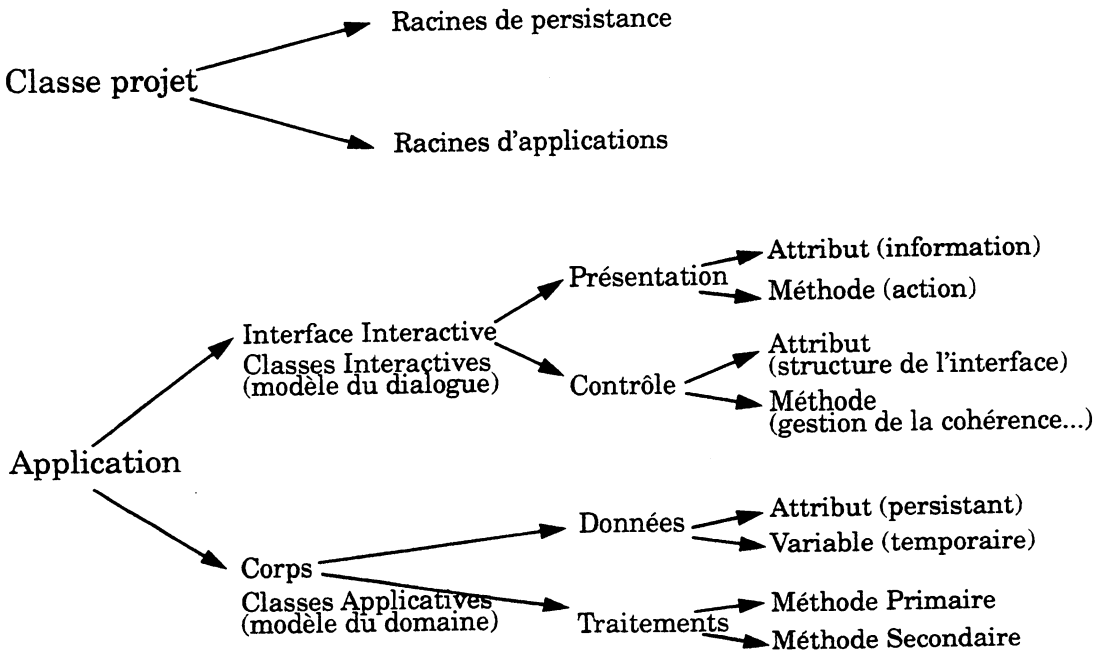


Figure 4-7 : Concepts clés de MOSAÏC.

Ces choix offrent l'avantage important de permettre d'identifier les classes applicatives par un processus de modélisation du domaine sans que le caractère persistant ou temporaire des données influe sur les classes applicatives identifiées. Par ailleurs, la décomposition d'une application en deux composants principaux (l'interface et le corps) permet de concevoir, de tester et de valider ces composants en parallèle. Enfin les choix retenus exploitent efficacement l'encapsulation et le masquage d'information pour augmenter la cohésion de chaque classe et pour limiter le couplage entre objets.

Nous présentons dans le chapitre suivant, chapitre 5, une définition plus précise de ce modèle ainsi que deux langages complémentaires (textuel et graphique) utilisables directement par le concepteur.

Chapitre 5

Classes applicatives, classes interactives et classes projets

1 Introduction

Dans le chapitre précédent nous avons introduit les concepts principaux du modèle MOSAÏC : les classes applicatives, les classes interactives et les classes projets, les attributs et les variables, les méthodes primaires et les méthodes secondaires. Pour représenter un système d'applications, nous avons présenté la notion de projet encapsulant une base d'objets applicatifs et des applications. Les objets applicatifs participent à la fois à la base d'objets persistants et aux corps des applications d'un projet. Dans une telle approche, la notion de schéma de base d'objets doit être précisée. Dans ce chapitre, nous précisons ces différents concepts, leur composition et leurs liens.

Le modèle MOSAÏC est supporté par un langage textuel et un langage graphique. Pour limiter la complexité des schémas, le langage graphique est un sous-ensemble du langage textuel dont la syntaxe est présentée dans l'annexe C. Peu d'études se sont intéressées aux critères de qualité applicables à un modèle ou à un langage pour définir un «bon» ensemble de concepts. Nous nous sommes limités à trois critères simples de qualité pour élaborer notre modèle MOSAÏC :

- Les concepts doivent tout d'abord respecter les choix du modèle d'architecture et leur nombre doit être limité.
- Les concepts et les propriétés de ces concepts doivent être aussi indépendants que possible.
- Les concepts doivent pouvoir être combinés et utilisés conjointement pour obtenir une puissance d'expression suffisante.

Pour offrir une aide au concepteur, nous nous attachons à définir un langage fortement et statiquement typé. Dans la suite, nous désignons la représentation élaborée pendant la conception sous le nom de **schéma de conception**.

2 Classes, classes applicatives et classes interactives

2.1 Définition générale d'une classe

Dans MOSAÏC, une **classe** décrit un modèle d'objets, mais elle ne permet pas d'accéder à l'ensemble des objets instanciés à partir de ce modèle. L'**objet** constitue, à l'exécution, le composant de base pour décomposer les applications et pour organiser les données persistantes. Une classe est définie par la donnée de sept éléments : <nc, H, S, F, PS, PF, PD>.

- nc est le nom de la classe.
- H représente les relations d'héritage (cf. section 7).
- S est la signature de la classe (cf. section 6).
- F décrit les différentes facettes de la classe (cf. section 2.2).
- PS est la partie structurelle ; elle est décrite par des attributs et elle permet de répondre à la question : quelles données contient chaque instance ? (cf. section 3).
- PF est la partie fonctionnelle ; elle est représentée à l'aide des notions de méthode et de message et elle permet de répondre à la question : que fait chaque instance ? (cf. section 4).
- PD est la partie dynamique ; elle est définie à l'aide des notions d'état, d'assertion et de règle d'intégrité et elle permet de répondre à la question : comment évolue chaque instance ? (cf. section 5).

L'approche de conception dirigée par les objets que nous suivons ne privilégie aucune dimension particulière à la différence des approches par les données ou par les traitements. Elle permet de définir simultanément ou séquentiellement les trois composantes d'un objet : la composante structurelle (PS), la composante fonctionnelle (PF) et la composante dynamique (PD). Cette approche de conception peut conduire à définir des classes sans composante structurelle, sans composante fonctionnelle ou sans composante dynamique.

2.2 Définition d'une classe applicative

Comme nous l'avons vu précédemment, l'**objet applicatif** constitue, à l'exécution, le composant de base à la fois pour décomposer les corps des applications et pour organiser les données persistantes. Une **classe applicative** décrit un modèle d'objet applicatif et sa représentation graphique est donnée dans la figure 5-1.

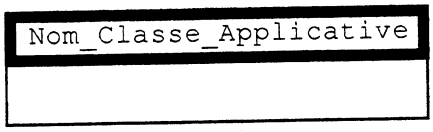


Figure 5-1 : Représentation graphique d'une classe applicative.

Nous rappelons que les classes applicatives sont obtenues par un processus de modélisation du domaine d'application.

2.3 Définition générale d'une classe facette

Une **facette** est une partie d'un objet ; elle en décrit un aspect particulier. Une facette n'est pas un objet à part entière : elle est propre à un objet et elle ne peut exister indépendamment de cet objet. Le cycle de vie d'une facette est inclus dans celui de l'objet. Une **classe facette** décrit un modèle de facette, c'est donc une partie d'une classe. La notion de classe facette permet de limiter la complexité d'une classe en fragmentant sa définition et en décrivant un aspect particulier de cette classe. A ce titre, une classe facette est rarement réutilisée par plusieurs classes. Graphiquement, une facette est représentée de la façon suivante.

Figure 5-2 : Représentation graphique d'une classe facette.

On notera qu'une facette ne peut pas disposer elle-même de facettes ; seuls les objets peuvent avoir différentes facettes. Le concept de facette est particulièrement utile pour notre approche dans laquelle un objet applicatif est à la fois un élément potentiellement persistant et un composant modulaire des différentes applications qui l'utilisent.

Les facettes d'une classe applicative peuvent être mises en évidence en étudiant le cycle de vie des données définies dans cette classe. Chaque facette regroupe les données ayant même cycle de vie et les traitements associés à ces données. La figure 5-3 et l'exemple 5-1 donnent un exemple partiel d'une classe applicative à facettes.

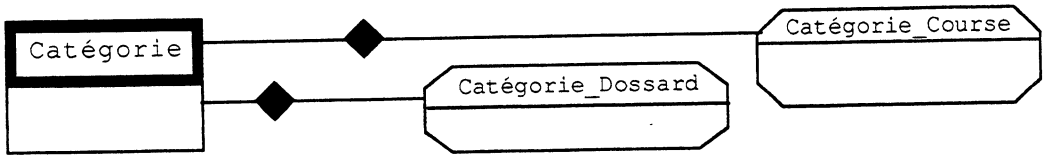


Figure 5-3 : Exemple de classe applicative constituée de deux classes facettes.

```
classe applicative Catégorie
  facette f_dossard : Catégorie_Dossard
  facette f_course : Catégorie_Course...
fin

classe facette Catégorie_Dossard
  ... (*facette créée seulement lors de l'affectation des dossards*)
fin

classe facette Catégorie_Course
  ... (*facette créée seulement lors de la gestion du concours*)
fin
```

Exemple 5-1 : Exemple de classes facettes.

Dans l'exemple 5-1, la facette `Catégorie_Dossard` regroupe tous les éléments (données et traitements) relatifs à la phase d'affectation des dossards alors que la facette `Catégorie_Course` regroupe les éléments relatifs au déroulement du concours.

2.4 Définition d'une classe interactive

L'**objet interactif** constitue, à l'exécution, la brique de base pour décomposer le gestionnaire de dialogue d'une application. Un objet interactif est toujours temporaire, sa durée de vie est incluse dans la durée d'activation de l'application qui l'a créé. Une **classe interactive** décrit un modèle d'objet interactif et elle est composée de deux parties distinctes : la partie contrôle et la partie présentation. La partie présentation est représentée par une **facette présentation** qui décrit directement une partie de l'écran visible par l'utilisateur. La partie **contrôle** est décrite dans la classe interactive elle-même. Elle réalise une partie de la gestion du dialogue et traduit les actions déclenchées par l'utilisateur via la facette présentation sous forme de messages envoyés aux classes applicatives. Elle assure également la cohérence entre les informations affichées par sa facette présentation et les informations affichées par les autres objets interactifs. La représentation graphique d'une classe interactive est la suivante :

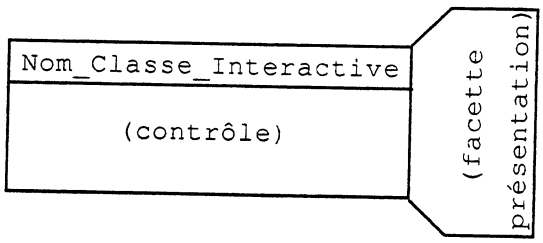


Figure 5-4 : Représentation graphique d'une classe interactive.

Une classe interactive ne peut pas posséder d'autres facettes que la facette présentation et cette dernière peut être absente lorsque la classe interactive ne décrit pas di-

rectement une partie de l'écran. C'est le cas par exemple lorsqu'une classe interactive est chargée de gérer la cohérence entre les différentes vues d'une vue multiple.

Nous rappelons que les classes interactives sont principalement obtenues par un processus de modélisation du dialogue entre l'utilisateur et l'application. Pour une stratégie plus fine, on peut consulter les travaux spécifiques menés autour du modèle PAC [CN89].

3 Aspects structurels d'une classe

3.1 Attributs

La partie structurelle d'une classe décrit une partie des données définies dans cette classe. Les autres données sont rattachées à la partie fonctionnelle. La partie structurelle d'une classe est composée d'attributs. Un **attribut** est un identificateur désignant un objet unique. Cet objet peut être «partageable», dans ce cas l'attribut est une **référence**, ou «non partageable», dans ce cas l'attribut est une **propriété**. Les attributs références permettent de représenter des relations binaires et orientées entre des objets alors que les attributs propriétés représentent les propriétés propres des objets. Un attribut référence peut être enrichi par une contrainte de dépendance existentielle. L'objet référencé est alors **dépendant** de l'objet référençant et la destruction d'un objet référençant entraîne obligatoirement la destruction de ses objets référencés dépendants. La contrainte de dépendance n'influe pas sur le partage : un objet dépendant peut être référencé par un autre objet ou dépendant d'un autre objet. Un objet non partageable est implicitement dépendant de son propriétaire.

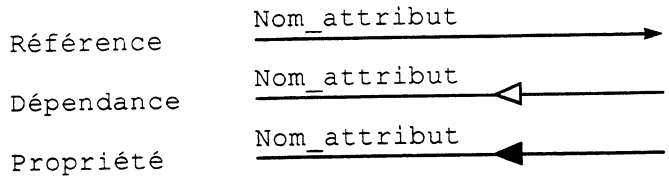


Figure 5-5 : Représentation graphique des trois catégories d'attributs.

Un attribut représente un chemin d'accès orienté d'un objet vers un second qui représente un lien total (l'attribut désigne toujours un objet) ou partiel (l'attribut désigne parfois un objet). Les cardinalités sont étudiées dans la section 5. Lorsque ce chemin est défini entre deux classes applicatives ou entre deux classes interactives, le chemin inverse peut être indiqué sous la forme d'un **attribut réciproque**. Un attribut et son attribut réciproque représentent un lien bi-directionnel entre deux objets. On notera qu'un attribut ne peut pas désigner une facette d'un objet.

```
1     classe applicative Personne
2       attribut      nom_personne : prop Chaîne
3                     prénom : prop Chaîne
4                     a_pour_licence : réf Licence inv licencié
5     fin
6
7     classe applicative Licence
8       attribut      numéro_licence : prop Entier
9                     licencié : réf Personne inv a_pour_licence
10                    club : prop Chaîne
11    fin
12
13    classe applicative Inscription
14      attribut      numéro_inscription : prop Entier
15                    dossard : prop Entier
16                    coureur : réf Licence
17                    performance : prop Réel
18    fin
```

Exemple 5-2 : Exemple de parties structurelles de classes applicatives.

Dans l'exemple 5-2, on notera les attributs réciproques définis lignes 4 et 9.

Le caractère partageable ou non partageable d'un objet dépend des attributs des objets qui le référencent et non de sa classe. Une classe peut donc générer à la fois des objets partageables et des objets non partageables. La figure 5-6 résume les situations interdites et les situations autorisées.

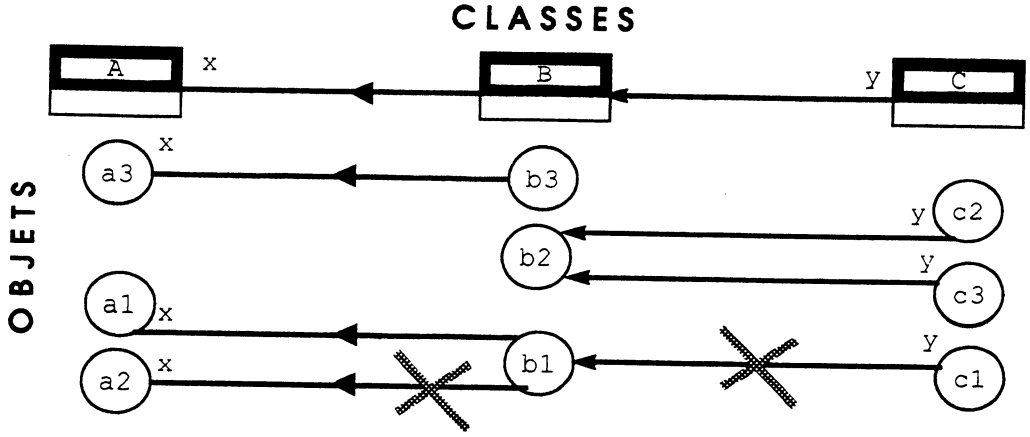


Figure 5-6 : Schéma des situations permises ou interdites.

On notera que la nature d'un attribut : référence, dépendance ou propriété est indépendante du caractère privé ou public de cet attribut. Un attribut propriété peut être public alors qu'un attribut référence, permettant l'accès à un objet partagé, peut être privé. Le masquage d'information est étudié plus en détail dans la section 6.

3.2 Classes prédéfinies

3.2.1 Classes de base

Les **classes de base** (entier, réel, booléen, caractère, chaîne, etc.) constituent les composants de base prédéfinis à partir desquelles le concepteur construit de nouvelles classes. Les instances des classes de base sont des objets. Cependant, pour améliorer la lisibilité du schéma de conception, nous imposons que tout objet partageable soit instance d'une classe construite et nommée par le concepteur. Une telle classe représente une classe d'entités du domaine d'application ou une classe d'objets de l'interface interactive. Pour cette raison les instances des classes de bases ne sont pas partageables. L'expérience montre d'ailleurs que les objets partagés sont, dans la plupart des cas, des instances de classes clairement identifiées dans le domaine de l'utilisateur.

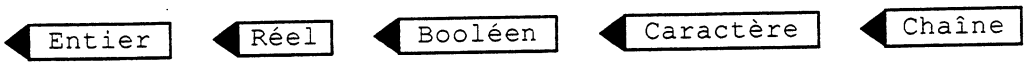
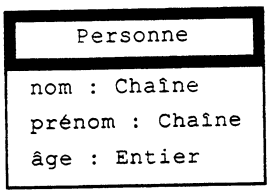


Figure 5-7 : Représentation graphique des classes de base.

Pour améliorer la lisibilité des schémas graphiques, les attributs propriétés (de nature classes de base) peuvent être représentés à l'extérieur ou à l'intérieur des classes comme le montre la figure 5-8. Par ailleurs, le mot clé **prop** qui introduit une propriété dans le langage textuel, peut être omis dans les attributs désignant des classes de base.




```
classe applicative Personne
   attributs
      nom : Chaîne
      prénom : Chaîne
      âge : Entier
fin
```

Figure 5-8 : Exemple graphique et textuel d'attributs propriétés.

Ce choix de représentation pour les classes de base permet une traduction aussi aisée vers les LPBO qui représentent les classes de base comme des domaines de valeurs (O_2 [ODeu89]), que vers ceux qui considèrent les classes de base comme des classes (Gemstone [BOS91]). Nous verrons dans la section 7 traitant de l'héritage, les facilités offertes par cette représentation pour définir des domaines en intension ou en extension.

3.2.2 Classes spécialisées dans l'interaction homme-machine

Si il existe depuis longtemps une certaine uniformité dans les classes de base offertes par les logiciels de base il n'en est pas de même pour les logiciels spécialisés dans la gestion de l'interaction entre l'utilisateur et l'ordinateur (boîtes à outils, classes spéciali-

sées, etc.). Certains outils offrent des concepts plus abstraits comme par exemple Motif développé en amont du système X-windows. Mais à notre connaissance aucun modèle n'est suffisamment général pour s'adapter à une large gamme de logiciels de base.

Aussi, dans les sections suivantes, nous proposons une représentation abstraite des principaux éléments du dialogue, en nous limitant aux entrées/sorties textuelles et en offrant un moyen de représenter les informations présentées à l'utilisateur et les actions qu'il peut déclencher.

Dans le cadre d'un développement, si le logiciel de base a été choisi, il peut être intéressant d'utiliser dès la conception, les principales classes offertes par ce logiciel de base. Toutefois une telle démarche, si elle facilite la phase de programmation, limite la généralité de l'étape de conception.

3.2.3 Classes constructeurs

Dans MOSAÏC, les **constructeurs** sont des classes génériques prédéfinies du modèle qui ne peuvent pas générer directement des objets. L'instanciation des paramètres d'un constructeur permet d'obtenir une classe que nous appelons commodément **construction** qui, elle, peut générer des objets. Cependant, nous avons imposé (cf. section 3.2.1) que les objets partagés soient des instances de classes construites et nommées par le concepteur ; les instances des constructions ne sont donc pas partageables. Comme pour les classes de base, le mot clé **prop** peut être omis dans les attributs désignant des constructeurs.

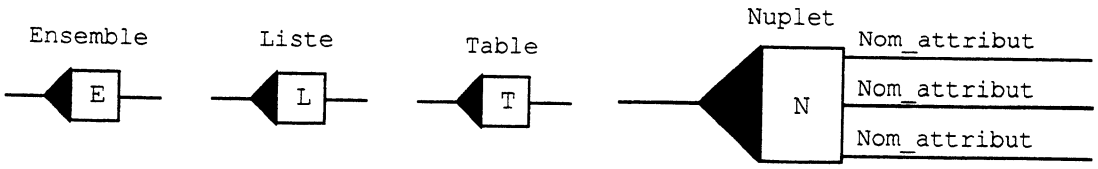


Figure 5-9 : Symboles graphiques des constructeurs.

On notera que seuls les attributs «simples» (sans constructeur) peuvent avoir un attribut réciproque. En effet, la combinaison de constructeurs rend difficile la prise en compte d'un attribut réciproque.

Le choix de représenter les constructeurs comme des classes permet une traduction aussi aisée vers les LPBO qui les représentent comme des constructeurs de valeurs (O_2 [ODeu89]) que vers ceux qui considèrent les constructeurs comme des classes génériques (Eiffel [Meye88]). La version actuelle de MOSAÏC ne permet pas de définir de nouvelles classes génériques : l'utilisation de la généricité est limitée aux quatre constructeurs de base.

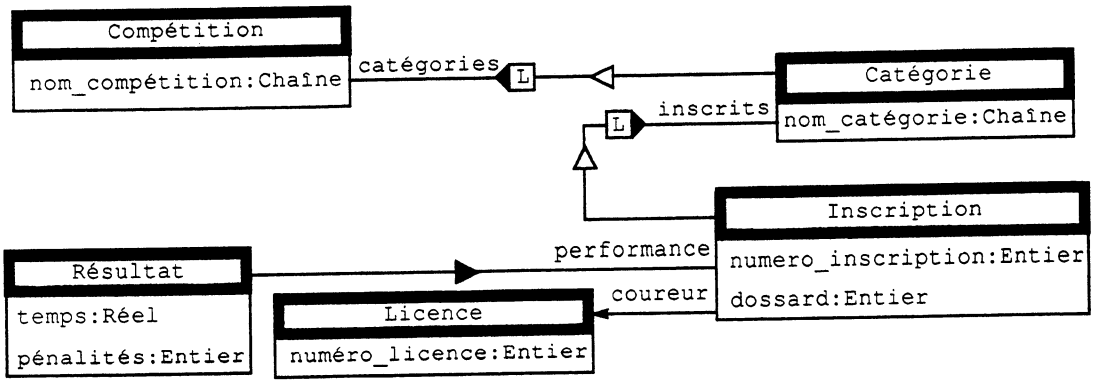


Figure 5-10 : Schéma de classes applicatives.

3.3 Partie structurelle des classes applicatives

Dans les classes applicatives, les attributs représentent des relations binaires entre des entités du domaine de l'application ou des propriétés propres de ces entités ; ils sont donc persistants. Les objets applicatifs ne peuvent pas désigner des objets interactifs par leurs attributs.

```
classe applicative Compétition
  attribut      nom_compétition : Chaîne
                catégories : Liste (dép Catégorie)
fin

classe applicative Catégorie
  attribut      nom_catégorie : Chaîne
                inscrits : Liste (dép Inscription)
fin

classe applicative Inscription
  attribut      numéro_inscription : Entier
                dossard : Entier
                coureur : réf Licence
                performance : prop Résultat
fin

classe applicative Résultat
  attribut      temps : Réel
                pénalités : Entier
                total : Réel
fin

classe applicative Licence
  attribut      numéro_licence : Entier
                licencié : réf Personne
                club :Chaîne
fin

classe applicative Personne
```

```
attribut      nom_personne : Chaîne
              prénom : Chaîne
              date_naissance : Date
fin
```

Exemple 5-3 : Représentation textuelle équivalente à la figure 5-10.

3.4 Parties structurelles des classes interactives

3.4.1 Partie structurelle d'une classe interactive

La partie structurelle d'une classe interactive permet de représenter d'une part l'organisation de l'interface interactive, notamment les relations avec les autres classes interactives, et d'autre part les relations statiques établies entre l'interface interactive et le corps de l'application. La plupart des relations structurelles entre les classes interactives sont issues de la modélisation du dialogue et elles sont souvent enrichies de dépendances existentielles qui organisent l'interface interactive comme une arborescence de classes interactives. Ces relations sont souvent bidirectionnelles pour permettre de gérer la cohérence de l'ensemble des éléments présentés à l'utilisateur. En revanche, les liens avec les classes applicatives sont toujours des références orientées vers les classes applicatives. Toutes ces relations sont temporaires puisque leur durée de vie est limitée à la durée d'activation de l'application dont les objets interactifs constituent l'interface interactive.

```
1    classe interactive Menu_Général
2      facette présentation :...
3      attribut      (*contrôle*)
4        menu_inscription : dép Menu_Inscription
5        menu_dossard : dép Menu_Dossard
6        menu_course : dép Menu_Course...
7    fin
8
9    classe interactive Menu_Course
10     facette présentation :...
11     attribut      (*contrôle*)
12       saisie_résultat : dép Saisie_Résultat
13       affichage : dép Affichage...
14   fin
15
16   classe interactive Saisie_Résultat
17     facette présentation :...
18     attribut      (*contrôle*)
19       info_générales : dép Info_Générales
20       info_course : dép Info_Course
21       inscription_courante : réf Inscription...
22   fin
23
24   classe interactive Info_Générales
25     facette présentation :...
26     attribut      (*contrôle*)
27       licence : réf Licence...
28   fin
29
```

```
30    classe interactive Info_course
31      facette présentation :...
32      attribut      (*contrôle*)
33        inscription_courante : réf Inscription...
34    fin
```

Exemple 5-4 : Quelques classes interactives.

Dans l'exemple 5-4, on peut remarquer les liens entre classes interactives (lignes 4, 5, 6, 12, 19 et 20) et les liens avec les classes applicatives (lignes 21, 27 et 33).

3.4.2 Partie structurelle de la facette présentation

Chaque attribut de la facette présentation d'une classe interactive représente une information «présentée» à l'utilisateur ou «saisie» par celui-ci. Les attributs de la présentation désignent des informations directement affichables comme des éléments des classes de base ou des constructions élaborées directement ou par combinaison à partir des classes de base. Les attributs de la présentation ne peuvent en aucun cas désigner des classes interactives ou des classes applicatives. Un **commentaire** constitué d'une chaîne de caractères peut être associé à chaque attribut dans le langage textuel. Ce commentaire permet de donner un titre à l'attribut. A la différence d'un attribut, l'information contenue dans un commentaire n'est pas modifiable dynamiquement.

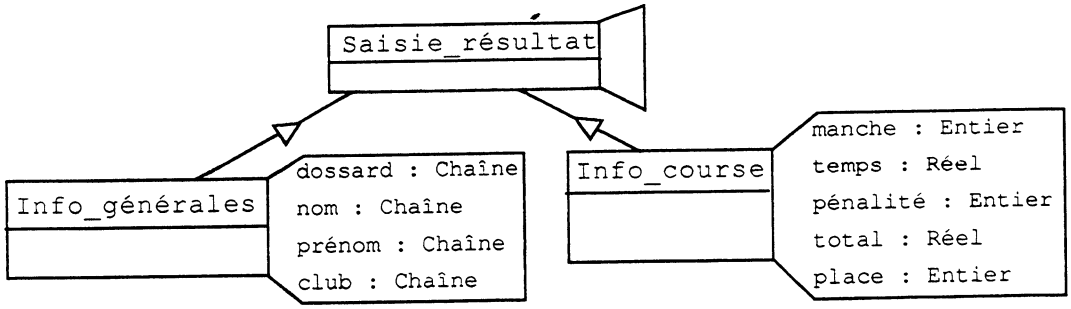


Figure 5-11 : Exemple de parties structurelles de facettes présentation.

L'exemple 5-5 affine le schéma de la figure 5-11.

```
classe interactive Saisie_Résultat
  attribut      (*contrôle*)
    info_générales : dép Info_Générales
    info_course : dép Info_Course...
fin

classe interactive Info_générales
  facette présentation :
       attribut
          dossard : Entier      commentaire "Numéro de dossard ?"
          nom : Chaîne          commentaire "Nom :"
          prénom : Chaîne
          club : Chaîne         commentaire "Club :"
```

```
  fin...
fin

classe interactive Info_course
  facette présentation :
        attribut
        manche : Entier          commentaire "Numéro de la manche (1 ou 2) :"
        temps : Réel             commentaire "Temps ?"
        pénalités : Entier       commentaire "Points ?"
        total : Réel             commentaire "Total :"
        place : Entier           commentaire "Classement :"
  fin...
fin
```

Exemple 5-5 : Forme textuelle enrichie de la figure 5-11.

La figure 5-12 donne un exemple de fenêtres correspondant aux classes interactives de l'exemple 5-5.

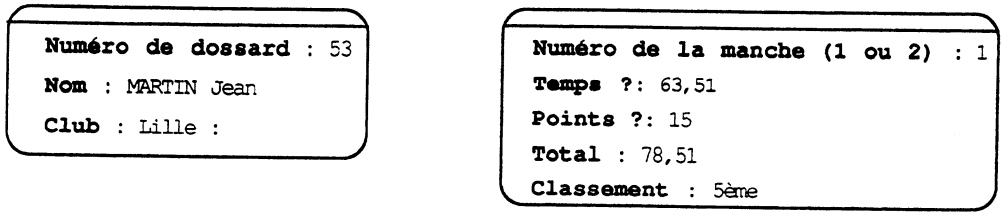


Figure 5-12 : Exemple de fenêtres correspondant à l'exemple 5-5.

Les informations complémentaires comme la localisation exacte des informations, les attributs typographiques, le caractère modifiable ou non modifiable des informations ou la forme des fenêtres ne sont pas étudiées pendant la conception mais pendant la définition des spécifications externes. Ces aspects sont affinés pendant la phase de réalisation pour prendre en compte les spécificités des systèmes cibles.

4 Aspects fonctionnels d'une classe

4.1 Généralités

La partie fonctionnelle d'une classe décrit les opérations réalisables par les objets de cette classe ; elle est décrite par un triplet <V, M, D> :

- V décrit les variables de la classe utilisables par les différentes méthodes de la classe (cf. section 4.2).
- M est l'ensemble des méthodes de la classe (cf. section 4.3).
- D représente les dépendances fonctionnelles de la classe (cf. section 4.4).

4.2 Variables

Les algorithmes retenus pour les méthodes d'une classe peuvent nécessiter la définition de données communes à plusieurs méthodes. Ces données sont décrites par les **variables** de la classe et on dispose, comme pour les attributs, de trois catégories de variables : les références, les dépendances et les propriétés. A la différence des attributs définis par modélisation du domaine d'application indépendamment des algorithmes des méthodes, les variables dépendent seulement du choix des algorithmes retenus pour les méthodes. Comme nous l'avons noté dans le chapitre 4, les variables des classes sont plus évolutives que les attributs et elles sont privées alors que les attributs sont généralement publics. Nous ne proposons pas de représentation graphique pour les variables et comme pour les attributs, le mot clé **prop** peut être omis avec les classes de base et les constructeurs.

```
1     classe applicative Compétition
2       attribut      nom_compétition : Chaîne
3                     date : Nuplet (          jour : Chaîne
4                                              année : Entier )
5                     catégories : Liste (dép Catégorie)
6       variable
7         dossards : Table (1.. 1000, réf Inscription)
8         classement_général : Liste (réf Inscription)
9       méthode inscription_de_dossard (i : Entier) : Inscription
10    fin
11
12    classe applicative Catégorie
13      attribut      nom_catégorie : Chaîne
14                    inscrits : Liste (réf Inscription)
15      variable
16        classement : Table (1..1000, réf Inscription)
17    fin
18
19    classe applicative Inscription
20      attribut      numéro_inscription : Entier
21                    dossard : Entier
22                    coureur : réf Licence
23                    performance : prop Résultat
24    fin
25
26    classe applicative Résultat
27      attribut      temps : Réel
28                    pénalités : Entier
29                    total : Réel
30    fin
```

Exemple 5-6 : Définitions de variables et d'attributs.

Dans l'exemple 5-6, la variable `dossards` de la ligne 7 introduit une redondance puisqu'une `Inscription` peut être trouvée connaissant son `dossard` en parcourant les `inscrits` (ligne 14) des `catégories` (ligne 5) de la `Compétition`. Mais cette variable permet une réalisation plus efficace du classement. De même les variables donnant les classements (lignes 8 et 16) résultent de choix techniques.

4.3 Méthodes

4.3.1 Définition d'une méthode

Une **méthode** est un traitement réalisé par un objet ; elle est définie par la donnée du quintuplet <nm, Sm, M, V, A> :

- nm est le nom de la méthode,
- Sm est la signature de la méthode, elle est composée des paramètres et d'assertions étudiées dans la section 5,
- M est l'ensemble des messages émis lors de l'activation de la méthode (cf. section 4.3.2),
- V regroupe les variables locales de la méthode et A en est l'algorithme (cf. section 4.3.3).

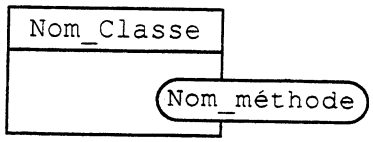


Figure 5-13 : Représentation graphique d'une méthode.

4.3.2 Messages et graphe fonctionnel d'une méthode

Les messages envoyés pendant l'exécution d'une méthode sont mis en évidence par un bloc syntaxique introduit par le mot clé **messages**. Dans une classe C, on retient seulement les messages envoyés à des instances de classes différentes de C. Les appels aux méthodes de C ou les accès aux attributs ou aux variables de C ne sont pas mentionnés dans cette rubrique. La notation utilisée pour représenter l'envoi d'un message est le point : «•». Par ailleurs, on ne mentionne pas précisément l'objet destinataire mais seulement sa classe et le nom de la caractéristique utilisée comme le montre l'exemple 5-7.

```
classe applicative Compétition
  attribut      nom_compétition : Chaîne
                catégories : Liste (dép Catégorie )

  méthode affecter_les_dossards
     message Catégorie • C_affecte_dossards
fin

classe applicative Catégorie
  attribut      nom_catégorie : Chaîne
                inscrits : Liste (dép Inscription)

  méthode C_affecte_dossards (dossard_courant : Entier) : Entier
     message Inscription • I_affecte_dossard
fin
```

```
classe applicative Inscription
  attribut      numéro_inscription : Entier
                dossard : Entier
                ...

  méthode I_affecte_dossard (dossard_courant : Entier)
fin
```

Exemple 5-7 : Messages émis lors de l'opération d'attribution des dossards.

Le **graphe fonctionnel** d'une méthode s'appuie sur les messages émis par cette méthode pour mettre en évidence graphiquement les conséquences de l'activation d'une méthode. Les nœuds du graphe fonctionnel sont les classes et leurs méthodes et à chaque envoi de message correspond un arc.

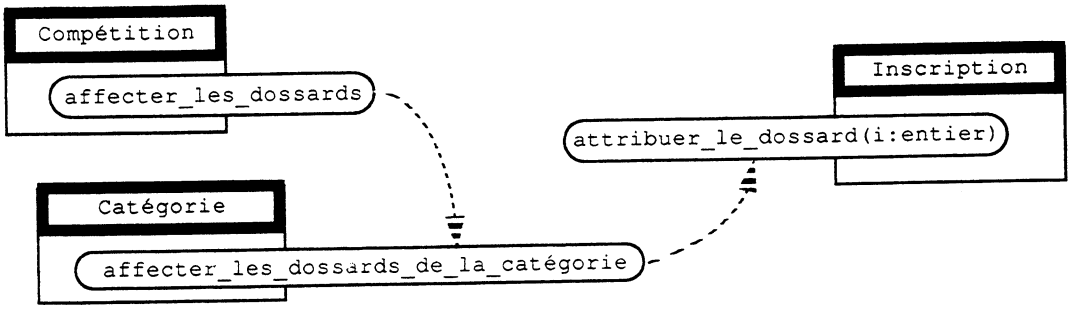


Figure 5-14 : Exemple de graphe fonctionnel correspondant à l'exemple 5-7.

On notera que le graphe fonctionnel d'une méthode peut être élaboré avant une définition complète de l'algorithme du corps de cette méthode. Il constitue alors un outil pour faciliter l'élaboration de l'algorithme.

4.3.3 Variables et algorithme d'une méthode

Le **corps** d'une méthode est composé de la déclaration de variables locales à cette méthode et de l'algorithme de la méthode. Le concepteur peut utiliser une notation algorithmique classique ou le langage du logiciel de base cible si celui-ci est fixé. Dans la suite nous utilisons une notation algorithmique classique comme le montre l'exemple 5-8.

```
classe applicative Compétition
  attribut      nom_compétition : Chaîne
                catégories : Liste (dép Catégorie )

  méthode affecter_les_dossards
    message Catégorie • C_affecte_dossards
    variable    dos_courant : Entier ;
                categ_courante : réf Catégorie ;
    début
```

```
        dos_courant := 0 ;
        pour_tout categ_courante dans catégories faire
           dos_courant := categ_courante • C_affecte_dossards (dos_courant)
        f_pour
   fin
fin

classe applicative Catégorie
  attribut      nom_catégorie : Chaîne
                inscrits : Ensemble (dép Inscription)

  méthode C_affecte_dossards (dossard_courant : Entier) : Entier
     message Inscription • I_affecte_dossard
     variable inscrit_courant : réf Inscription ;
     début
        pour_tout inscrit_courant dans inscrits faire
           dossard_courant := dossard_courant + 1 ;
           inscrit_courant • I_affecte_dossard (dossard_courant)
        f_pour
  fin
fin

classe applicative Inscription
  attribut      numéro_inscription : Entier
                dossard : Entier...

  méthode I_affecte_dossard (dossard_courant : Entier)
     début dossard := dossard_courant
  fin
fin
```

Exemple 5-8 : Quelques algorithmes.

4.3.4 Règle méthodologique

Pour limiter le couplage des objets et encourager le respect des principes d'autonomie et de localité (cf. chapitre 3), nous adaptons les règles proposées par le système Demeter présentées dans la section 3.4 du chapitre 3. Une méthode m d'une classe C peut envoyer des messages aux classes suivantes :

- les classes désignées par les attributs et les variables de la classe C, les classes des arguments de m et les classes des objets créés par m,
- les classes accessibles indirectement par l'intermédiaire de constructeurs à partir des attributs de C, des variables de la classe C, des variables locales de m et des arguments de m lorsque ceux-ci désignent des constructeurs.

Ces règles limitent le couplage en interdisant la «navigation» dans les classes. Ainsi, dans l'exemple 5-8, la méthode `affecter_les_dossards` de la classe `Compétition` peut envoyer des messages aux objets de la classe `Catégorie` mais elle ne peut pas envoyer de messages à des objets de la classe `Inscription`.

4.4 Dépendances fonctionnelles d'une classe

Les **relations de dépendance fonctionnelle** sont définies entre des classes : une classe A est **dépendante fonctionnellement** d'une classe B si A et B sont distinctes et si une méthode au moins de la classe A émet un message en direction d'un objet de la classe B. Un «message» peut être l'activation d'une méthode ou la consultation d'un attribut.

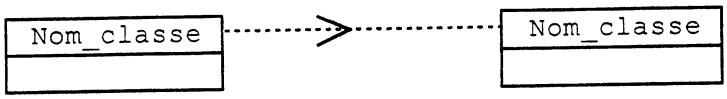


Figure 5-15 : Représentation graphique d'une dépendance fonctionnelle entre deux classes.

Les relations de dépendances fonctionnelles sont importantes car elles permettent de connaître les classes susceptibles d'être affectées par la modification d'une classe.

```
classe applicative Compétition
  attribut...
  utilise Catégorie                                  (*dépendance fonctionnelle*)
  méthode affecter_les_dossards
     message Catégorie • C_affecte_dossards
fin

classe applicative Catégorie
  attribut...
  utilise Inscription                                (*dépendance fonctionnelle*)
  méthode C_affecte_dossards (dossard_courant : Entier) : Entier
     message Inscription • I_affecte_dossard
fin

classe applicative Inscription
  attribut...
  méthode I_affecte_dossard (dossard_courant : Entier)
fin
```

Exemple 5-9 : Dépendances fonctionnelles de l'exemple 5-8.

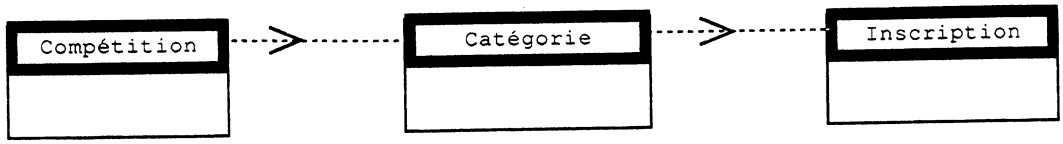


Figure 5-16 : Graphe des dépendances fonctionnelles de l'exemple 5-8.

On peut établir un parallèle entre les relations structurelles, porteuses d'informations concernant l'organisation du domaine ou celle du gestionnaire de dialogue, et les dépendances fonctionnelles porteuses d'informations concernant l'organisation fonction-

nelle de l'application. Ces deux catégories de relations caractérisent deux aspects différents d'une application et elles sont à distinguer clairement. L'existence d'une relation structurelle entre deux classes n'implique pas nécessairement l'existence d'une dépendance fonctionnelle entre ces deux classes. Par contre une dépendance fonctionnelle entre deux classes impose que la classe fournisseur soit visible par la classe cliente. Cette visibilité peut être le fait des attributs ou des variables de classes de la classe cliente ou bien le fait des arguments ou des variables locales d'une méthode de la classe cliente.

4.5 Aspects fonctionnels des classes applicatives

4.5.1 Méthodes primaires et méthodes secondaires

Parmi les méthodes d'une classe applicative, on distingue les méthodes primaires et les méthodes secondaires. Les méthodes primaires doivent être conçues de façon à ne pas introduire d'incohérence dans les objets applicatifs alors que les méthodes secondaires peuvent introduire des incohérences temporaires. La notion de cohérence est définie dans la section 5.

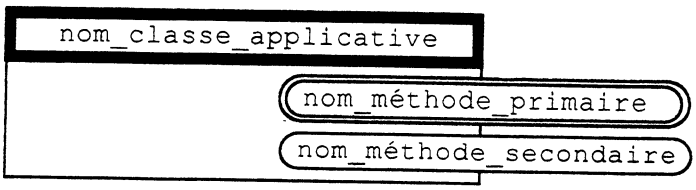


Figure 5-17 : Représentation graphique des méthodes primaires ou secondaires.

Une règle de gestion d'une compétition dit que : «tous les dossards sont attribués ou aucun dossard n'est attribué». Les méthodes `C_affecte_dossards` et `I_affecte_dossard` des classes `Catégorie` et `Inscription` sont secondaires puisqu'elles attribuent seulement une partie des dossards (ceux d'une catégorie pour la première et un seul dossard pour la seconde). Elles doivent donc être activées dans une méthode primaire `affecter_les_dossards` de la classe `Compétition` qui réalise un traitement complet.

```
classe applicative Compétition
  attribut...
  méthode primaire affecter_les_dossards
     message Catégorie • C_affecte_dossards
fin

classe applicative Catégorie
  attribut...
  méthode secondaire C_affecte_dossards (dossard_courant : Entier) : Entier
     message Inscription • I_affecte_dossard
fin
```

```
classe applicative Inscription
  attribut...
  méthode secondaire I_affecte_dossard (dossard_courant : Entier)
fin
```

Exemple 5-10 : Quelques méthodes primaires et quelques méthodes secondaires.

4.5.2 Persistance des méthodes et opération de promotion

Les attributs d'une classe applicative sont persistants alors que les variables sont temporaires (cf. chapitre 4). Cette distinction doit également être introduite dans les méthodes :

- Les méthodes des classes applicatives sont par défaut dites **persistantes**.
- Une méthode est dite **temporaire** si elle accède à une variable de la classe ou si elle utilise une méthode temporaire.

Cette propriété n'a pas de représentation graphique mais elle est mentionnée textuellement. On notera qu'en dehors d'une activité logicielle, seules les caractéristiques persistantes d'un objet applicatif persistant sont mémorisées et accessibles. Dans le cadre d'une activité logicielle, le premier accès aux caractéristiques temporaires d'un objet applicatif doit donc être précédé par l'exécution d'une opération particulière que nous appelons **promotion** chargée notamment d'affecter l'espace mémoire nécessaire aux données temporaires. Cette opération de promotion complète l'opération d'instanciation qui permet de créer un nouvel objet. Une mise en œuvre de l'opération de promotion est proposée dans le chapitre 6 qui traite de la traduction d'un schéma de conception MOSAÏC dans différents LPBO cibles.

4.6 Dimension fonctionnelle des classes interactives

Les actions offertes à l'utilisateur par un objet interactif sont définies dans la partie fonctionnelle de la facette présentation de cet objet. Ces actions, déclenchables seulement par l'utilisateur, sont représentées par des méthodes.

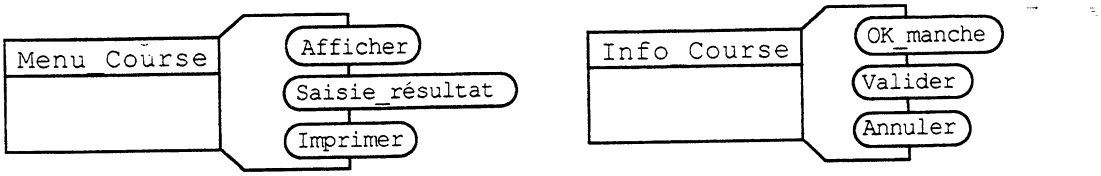


Figure 5-18 : Représentation graphique d'une méthode dans la facette présentation.

La figure 5-19 présente des fenêtres correspondant à la figure 5-18. La présentation graphique exacte de ces actions (liste de commandes dans un menu déroulant, de boutons actifs, etc.) dépend des services offerts par le logiciel de base cible.

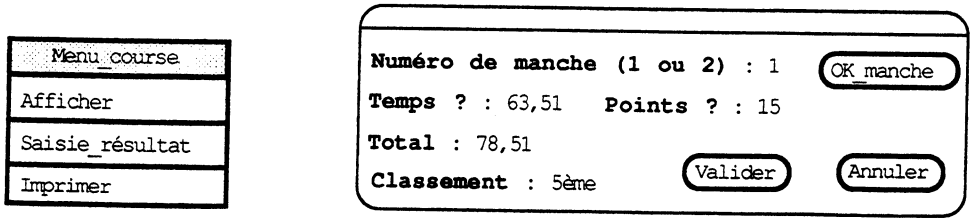


Figure 5-19 : Fenêtres correspondant à la synthèse de la figure 5-18 et de l'exemple 5-5.

Les méthodes de la facette présentation d'un objet interactif sont chargées de réaliser les premiers tests de cohérence (format, etc.) et de maintenir la cohérence entre les différentes informations affichées par cette facette. Pour assurer l'indépendance entre le gestionnaire de dialogue et un logiciel de base particulier, les méthodes de la facette présentation ne peuvent pas envoyer de messages aux autres objets interactifs ou aux objets applicatifs. Le service attendu par l'utilisateur est réalisé par une méthode de l'objet interactif lui-même (dans la partie contrôle). Le corps des méthodes de la facette présentation est donc généralement vide pendant la conception et c'est au moment de la programmation effective du système que ces méthodes sont enrichies pour prendre en compte les services offerts par le logiciel de base cible. Si le logiciel de base cible est connu lors de la conception, les classes de base pour la gestion du dialogue, par exemple `Menu_Déroulant`, `Bouton`, `Fenêtre` ou `Dialogue` peuvent être utilisées pour enrichir les présentations des classes interactives.

5 Aspects dynamiques d'une classe[1]

La partie dynamique qui permet de décrire différents états cohérents et différentes règles d'intégrité a actuellement un rôle limité de documentation du schéma de conception.

5.1 Les états

Un état permet de décrire une valuation remarquable des attributs et des variables d'une classe. Un **état** est composé d'un nom et d'une proposition logique. La proposition logique d'un état défini dans une classe C peut porter sur :

- les états, les attributs et les variables de C,
- les caractéristiques publiques (états, attributs et variables) des classes accessibles à partir des attributs et des variables de C.

Un état peut être testé, il se comporte alors comme une fonction booléenne dont le

1. Cette section s'inspire largement des services offerts par le langage Eiffel [Meye88].

résultat est `vrai` ou `faux` selon que la proposition associée est vérifiée ou non (selon que l'état est atteint ou non). Un état prédéfini, nommé **initial**, permet de représenter l'état d'un objet de la classe lors de sa création.

Nous ne proposons pas de représentation graphique pour les états. Des exemples d'états sont donnés dans la section suivante (exemple 5-11).

5.2 Règles d'intégrité

Une **règle d'intégrité** est une proposition logique définie dans une classe. Une classe peut contenir différentes règles d'intégrité et l'ensemble de ces règles d'intégrité définit l'**invariant** de la classe. Les règles de validité et d'évaluation des règles d'intégrité sont abordées dans la section 5.4.

```
classe applicative Inscription
  attribut      numéro_inscription : Entier
                dossard : Entier...
fin

classe applicative Catégorie
  attribut      nom_catégorie : Chaîne
                inscrits : Ensemble (dép Inscription)
  état
     initial = (card(inscrits) = 0) (*pas d'inscrit à sa création*)

     minimal = (nom_catégorie ≠ "") et (card(inscrits) < 100)
(*une catégorie a un nom et moins de 100 inscriptions*)

     dossards_affectables = (card(inscrits) ≥ 3)
(* trois inscrits sont nécessaires pour attribuer les dossards d'une catégorie*)

     dossards_non_affectés = (∀ i ∈ inscrits, i.dossard = 0)
(*aucun inscrit de la catégorie n'a un dossard*)

     dossards_affectés = (∀ i ∈ inscrits, i.dossard ≠ 0)
(*tous les inscrits de la catégorie ont un dossard*)

  invariant
     minimal
     ¬ dossards_affectables et_alors dossards_non_affectés
     dossards_affectés et_alors dossards_affectables
     initial ou dossards_non_affectés ou dossards_affectés
fin

classe applicative Compétition
  attribut      nom_compétition : Chaîne
                catégories : Liste (dép Catégorie )
  état
     initial = (card(catégories) = 0) (*pas de catégorie à sa création*)

     minimal = (nom_compétition ≠ "") et (card(catégories) ≤ 40)
(*une compétition a un nom et moins de 50 catégories*)

     dossards_affectables = (∃ c ∈ catégories / c.dossards_affectables)
(*on peut attribuer les dossards si on peut les attribuer dans*)
(*au moins une catégorie *)
```

```
    dossards_non_affectés = (∀ c ∈ catégories, c.dossards_non_affectés)
(*aucun dossard n'est attribué*)

    dossards_affectés = (∀ c ∈ catégories, c.dossards_affectés)
(*tous les dossards sont attribués*)

  invariant     minimal
                ¬ dossards_affectables et_alors dossards_non_affectés
                dossards_affectés et_alors dossards_affectables
                initial ou dossards_non_affectés ou dossards_affectés
fin
```

Exemple 5-11 : Quelques états et un invariants.

5.3 Précondition et postcondition

Le traitement réalisé par une méthode peut être exprimé de façon déclarative par une **précondition** et une **postcondition**. Les propositions logiques des préconditions et des postconditions sont construites de la même façon que les propositions logiques des états et des règles d'intégrité. L'exemple 5-12 illustre l'utilisation des assertions.

```
classe applicative Catégorie
  attribut      nom : Chaîne
                inscrits : Liste (dép Inscription)
  variable
     class_alphab : Table [1..100] (réf Inscription)

  état     initial = (card(inscrits) = 0)
           minimal = (nom_catégorie ≠ "") et (card(inscrits) ≤ 100)
           dossards_non_affectables = (card(inscrits) < 3)
           dossards_affectables = (card(inscrits) ≥ 3)
           dossards_non_affectés = (∀ i ∈ inscrits, i.dossard = 0)
           dossards_affectés = (∀ i ∈ inscrits, i.dossard ≠ 0)

  méthode inscrire (i : Inscription)
       précondition dossards_non_affectés et (card(inscrits) ≤ 99)
                    et (j ∈ inscrits ⇒ i.coureur ≠ j.coureur)
       postcondition inscrits = ancien (inscrits) ∪ {i}

  méthode supprimer (i : Inscription)
       précondition dossards_non_affectés et (i ∈ inscrits)
       postcondition inscrits = ancien (inscrits) \ {i}

  méthode classement_alphabétique
       précondition VRAI
       postcondition (i, j ∈ [1..card(inscrits)]) ⇒ (i ≤ j ⇔
              (class_alphab[i].coureur.nom ≤ class_alphab[j].coureur.nom))

  méthode C_affecte_dossards
       précondition dossards_affectables
       postcondition dossards_affectés
fin
```

Exemple 5-12 : Exemple de préconditions et de postconditions.

Dans l'exemple 5-12, la précondition de la méthode `inscrire` indique qu'elle n'est pas utilisable si les dossards ont été affectés, il s'agit là d'une règle de gestion du règlement des compétitions. La méthode `classement_alphabétique` est activable quelque soit l'état de l'objet. La précondition et la postcondition de la méthode `C_affecte_dossards` indiquent l'effet de cette méthode.

Le mot clé **ancien** (identificateur) désigne la valeur d'un identificateur avant l'activation de la méthode. Dans Eiffel le mot clé `old` remplit le même rôle.

5.4 Partie dynamique des classes applicatives

Comme nous l'avons défini dans le chapitre 4, les méthodes primaires doivent préserver l'intégrité des objets applicatifs alors que les méthodes secondaires peuvent introduire des incohérences temporaires. L'invariant d'une classe applicative définit donc un ensemble de règles d'intégrité qui doivent être vérifiées avant et après chaque exécution d'une méthode primaire.

Nous ne nous intéressons pas dans ce travail à la validation effective des règles d'intégrité ou à l'élaboration de preuves formelles à partir des assertions et des règles d'intégrité définies dans les classes. Des travaux sont menés autour de ce thème dans le cadre du projet Aristote [Mart91].

5.5 Partie dynamique des classes interactives

Les états et les assertions des classes interactives permettent de préciser de façon déclarative, le déroulement du dialogue entre l'utilisateur et l'application.

6 Masquage et héritage

6.1 Signature d'une classe et type des objets

La **signature** d'une classe regroupe les éléments publics utilisables par les autres classes : des attributs, des variables, des méthodes et des états. Elle est utilisée pour limiter les communications entre deux classes : seuls les éléments contenus dans la signature d'une classe sont utilisables par les classes clientes. A ce titre la signature détermine le **type** des objets de la classe. Nous ne proposons pas de représentation graphique pour la notion de signature mais elle est représentée textuellement par une liste d'exportation comme le montre l'exemple 5-13.

```
classe applicative Catégorie
  exporte    fonction nom, inscrits
             méthode inscrire, supprimer, classement
             état initial, dossards_affectables, dossards_non_affectables
```

```
attribut      nom : Chaîne
              inscrits : Liste (dép Inscription)
variable
   class_alphab : Table [1..100] (réf Inscription)

état     initial = card(inscrits) = 0
         minimal = nom_catégorie ≠ "" et card(inscrits) ≤ 100
         dossards_affectables = (card(inscrits) ≥ 3)
         dossards_non_affectables = card(inscrits) < 3

méthode inscrire (i : Inscription)
méthode supprimer (i : Inscription)
méthode classement_alphabétique
méthode classement : Liste (Inscription)
fin
```

Exemple 5-13 : Signature d'une classe applicative.

Lorsqu'un attribut ou une variable d'une classe est public, il est accessible seulement en lecture. Il se comporte donc, pour les clients de cette classe, comme une fonction. Pour cette raison, une donnée publique (attribut ou variable) est introduite dans la signature par le mot clé `fonctions`.

En général les attributs sont publics alors que les variables sont privées. En effet les variables sont introduites lors du choix des algorithmes et leur niveau d'évolutivité est identique à celui des algorithmes. En revanche, les attributs proviennent de la modélisation des entités du domaine et leur niveau d'évolutivité est voisin de celui des signatures des méthodes. Par ailleurs, les attributs publics ne sont pas eux-mêmes responsables de l'accroissement du couplage, seule l'utilisation des attributs par les méthodes accentue le couplage. Or nous avons proposé une règle méthodologique (cf. section 4.3.4) qui impose des restrictions sur les messages pour limiter le couplage.

Les signatures des classes applicatives contiennent généralement les attributs et les méthodes primaires alors que les méthodes secondaires peuvent être privées ou publiques.

Les signatures des classes interactives contiennent généralement les méthodes de mise-à-jour alors que les méthodes décrivant des actions déclenchables par l'utilisateur sont privées.

6.2 Deux relations d'héritage

L'héritage est un mécanisme essentiel des techniques à objet et nous avons vu dans le chapitre 3 qu'il pouvait donner lieu à différentes interprétations (ensembliste, sous-typage, réutilisation, logique, etc.) [Card84]. Nous retenons deux interprétations particulières de l'héritage entre classes, la relation **est-un** et la relation **adapte**, et nous les détaillons dans les sections suivantes.

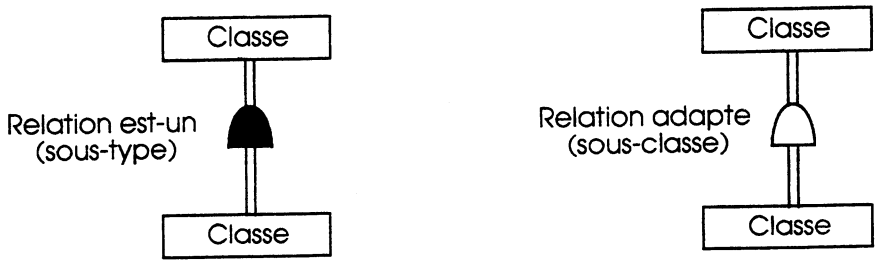


Figure 5-20 : Représentation graphique des deux relations d'héritage.

Lorsqu'une classe C' «est-une» classe C on dit que C' est un **sous-type** de C. Lorsqu'une classe C' «adapte» une classe C on dit que C' est une **adaptation** ou une **sous-classe** de C. L'héritage multiple est autorisé ainsi que la combinaison de sous-typage et d'adaptation.

6.3 Relation de sous-typage

La relation «est-un» définit une relation de sous-typage entre les classes : les instances de C' sont considérées comme étant du même type que les instances de C. Un objet d'une classe C peut donc être remplacé par un objet d'une classe C' sous-type de C. La relation «est-un» impose différentes contraintes sur les constituants des classes.

6.3.1 Contraintes sur la signature

Lorsqu'une classe C' est sous-type d'une classe C, relation notée $C' \leq C$, alors les signatures et les facettes sont soumises aux contraintes suivantes :

- Pour toute méthode de la signature de C de la forme $m(a) \rightarrow r$, la signature C' offre une méthode de même nom de la forme $m(a') \rightarrow r'$ telle que : d'une part la classe de l'argument a soit un sous-type de la classe de l'argument a' et d'autre part la classe du résultat r' soit un sous-type de la classe du résultat r.
- Pour toute fonction la signature de C de la forme $f \rightarrow r$, la signature C' offre une fonction de même nom de la forme $f \rightarrow r'$ telle que la classe du résultat r' soit un sous-type de la classe du résultat r.
- Tout état présent dans la signature de C est également présent dans la signature de C'.
- Pour toute classe facette J de C de la forme j : J, C' offre une facette de même nom de la forme j : J' telle que la classe facette J' soit un sous-type de la classe facette J.
- La signature de C' peut contenir de nouvelles méthodes, de nouvelles fonctions ou de nouveaux états. C' peut contenir de nouvelles facettes.

```
classe applicative Inscription_générale

  exporte    fonction numéro_inscription, dossard, coureur, performance
             méthode affecter_dossard, affecter_résultat

  attribut
     numéro_inscription : Entier
     dossard : Entier
     performance : Réel
  méthode affecter_dossard (d : Entier)
  méthode affecter_résultat (r : Réel)
fin


classe applicative Inscription_Mono
  est-un Inscription_générale        (*sous-type de Inscription_générale*)

  exporte    fonction coureur
             méthode créer

  attribut
     coureur : réf Licence           (*Inscription_Mono = 1 Licencié*)
     performance : Réel
  méthode créer (l : Licence)
fin


classe applicative Inscription_Bi
  est-un Inscription_générale        (*sous-type de Inscription_générale*)

  exporte    fonction coureurs
             méthode créer

  attribut
     coureurs : Ensemble (réf Licence)
  invariant card (coureurs = 2)      (*Inscription_Bi = 2 Licenciés*)
  méthode créer (l : Ensemble (réf Licence))
fin
```

Exemple 5-14 : Exemple de sous-typage.

6.3.2 Contraintes sur les caractéristiques privées

La relation de sous-typage impose des contraintes sur les caractéristiques privées : attributs, variables et méthodes. Si C' est un sous-type de C alors :

- Les attributs et les variables privés de C peuvent être redéfinis dans C'. Dans ce cas, toutes les méthodes de C ou d'une classe parente de C utilisant des attributs ou des variables redéfinis dans C' doivent également être redéfinies dans C' pour prendre en compte ces modifications.
- Les méthodes privées de C peuvent être redéfinies dans C' soit en respectant les règles de sous-typage présentées dans la section 6.3.1 soit sans tenir compte de ces règles. Dans ce dernier cas, toutes les méthodes de C ou d'une classe parente de C utilisant des méthodes privées redéfinies dans C' doivent également être redéfinies pour prendre en compte les modifications des méthodes privées.
- De nouveaux attributs, variables ou méthodes privés peuvent être ajoutés à C'.

6.3.3 Contraintes sur les constructeurs

La relation de sous-typage est définie entre les constructeurs de la façon suivante :

- Nuplet ($a_1 : C'_1$, $a_2 : C'_2$) ≤ Nuplet ($a_1 : C_1$, $a_2 : C_2$) ⇔ ∀ i, $C'_i \leq C_i$.
- Nuplet ($a_1 : C_1$, $a_2 : C_2$) ≤ Nuplet ($a_1 : C_1$, $a_2 : C_2$,..., $a_n : C_n$).
- Liste (C') ≤ Liste (C) ⇔ C' ≤ C.
- Ensemble (C') ≤ Ensemble (C) ⇔ C' ≤ C.
- Table (C') ≤ Table (C) ⇔ C' ≤ C.

On notera que la définition de telles relations de sous-typage est possible car nous avons interdit le partage des objets générés par les constructions. Les constructeurs se comportent comme des constructeurs de valeurs et, dans ce cas, les règles présentées ci-dessus garantissent une relation de sous-typage [DLR91] (cf. annexe A).

La version actuelle de MOSAÏC ne permet pas de définir de nouveaux constructeurs par héritage et une classe ne peut pas hériter d'un constructeur.

6.3.4 Contraintes sur les classes de base

La notion d'invariant permet de limiter l'état des objets d'une classe de base ; on peut ainsi définir des domaines restreints.

```
classe applicative Compétition
  attribut      nom : Chaîne,
                discipline : Discipline
                sexe : Sexe
fin

classe Discipline   est-un Chaîne
  invariant self = 'Slalom' ou self = 'Descente' ou self = 'Vitesse'
fin

classe Sexe     est-un Caractère
  invariant self = 'M' ou self = 'F'
fin

classe applicative Compétition_bis
  attribut      nom : Chaîne,
                discipline : Chaîne {'Slalom', 'Descente', 'Vitesse'}
                sexe : Caractère {'M', 'F'}
fin
```

Exemple 5-15 : Quelques sous-types sur des classes de base

Dans l'exemple 5-15, l'attribut `Discipline` de la classe `Compétition` est limité aux chaîne de caractère : `Slalom`, `Descente` ou `Vitesse`. Le langage textuel offre une facilité syntaxique pour décrire les domaines en extension comme le montre la classe `Compétition_bis` dans l'exemple 5-15.

6.3.5 Contraintes sur la partie dynamique

La partie dynamique dont le rôle est seulement documentaire n'est pas affectée par ces contraintes.

6.4 Relation d'adaptation

La relation d'adaptation est un mécanisme de réutilisation qui n'introduit pas de relation de sous-typage entre une classe et ses sous-classes. Un objet d'une classe C ne peut donc pas être remplacé par un objet d'une classe C' adaptation de la classe C. Lorsqu'une classe C' adapte une classe C les contraintes suivantes doivent être vérifiées :

- Les méthodes, les attributs, les variables, les états et les facettes de la classe C peuvent être redéfinis librement dans C'.
- Des méthodes, des attributs, des variables, des états ou des facettes peuvent être ajoutés à C'.
- La signature de C' est définie indépendamment de celle de C.
- Les méthodes de C ou d'une classe parente de C qui utilisent des attributs, des variables ou des méthodes redéfinis dans C' doivent être redéfinies dans C' pour prendre en compte les nouvelles définitions dans C'. Toutefois, les méthodes redéfinies dans C' en respectant les règles de sous-typage (cf. section précédente 6.3.1) n'imposent pas de redéfinir les méthodes qui les utilisent.

```
classe applicative Inscription
  exporte    fonction numéro_inscription, dossard, coureur, performance
             méthode créer, affecter_dossard, affecter_résultat

  attribut      numéro_inscription : Entier
                dossard : Entier
                coureur : réf Licence
                performance : Réel
  méthode créer (l : Licence)
  méthode affecter_dossard (d : Entier)
  méthode affecter_résultat (r : Réel)
fin


classe applicative Inscription_Bi
  adapte Inscription
  exporte    fonction numéro_inscription, dossard, coureur, performance
             méthode créer, affecter_dossard, affecter_résultat

  attribut      coureur : Ensemble (réf Licence)
(*l'attribut coureur est redéfini sans respecter les règles de sous-typage*)
  invariant card (coureur = 2)
  méthode créer (l : Ensemble (réf Licence))
fin
```

Exemple 5-16 : Adaptation d'une classe applicative.

6.5 Contraintes sur les classes applicatives et sur les classes interactives

Les classes applicatives ne peuvent hériter que des classes applicatives et les classes interactives ne peuvent hériter que des classes interactives.

7 Classes projets et objets persistants

7.1 Définition

Dans le chapitre 4 nous avons introduit la notion de **projet** comme l'encapsulation d'une base d'objets applicatifs persistants et d'un ensemble d'applications exécutables par l'utilisateur. Nous avons également noté l'analogie qui existe entre un projet et un objet :

- Dans un objet, les données sont accessibles par des attributs et dans un projet, les objets persistants sont accessibles par des racines de persistance.
- Un objet réalise un ensemble de traitements définis par des méthodes et un projet réalise un ensemble de traitements définis par les applications activables par l'utilisateur.

Une **classe projet** est un modèle de projets comme une classe est un modèle d'objets. Une classe projet est définie par trois éléments <ncp, Rp, Ra> :

- ncp est le nom de la classe projet,
- Rp regroupe l'ensemble des racines de persistance de la classe projet,
- Ra regroupe l'ensemble des racines d'applications de la classe projet.

Graphiquement une classe projet est représentée de la façon suivante :

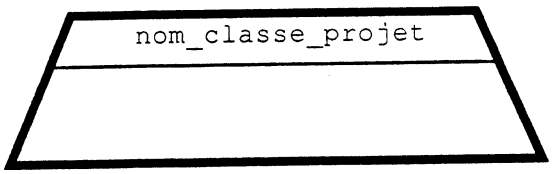


Figure 5-21 : Représentation graphique d'une classe projet.

Les projets sont des entités conceptuelles et à ce titre on pourrait préciser différents aspects : identification des projets, relations entre projets, structuration des projets, réutilisation des classes projets par héritage, etc. On notera que certains systèmes opérationnels offrent des concepts proches de la notion de projet : le SGBDOO O_2 [ODeu89] propose le concept de «schéma» et le langage PEPLOM [Dech93] introduit la notion de «module» (cf. chapitre 6). Nous nous limitons à définir la persistance des objets et les points d'entrée des applications interactives.

7.2 Racines de persistance

Les objets persistants encapsulés dans un projet sont des objets applicatifs alors que les objets interactifs ont une durée de vie limitée à celle de l'application qui les a créés. Les objets applicatifs persistants d'un projet sont accessibles par les racines de persistance de ce projet.

Une **racine de persistance** est un identificateur défini dans une classe projet et qui désigne un objet applicatif de la même façon qu'un attribut est un identificateur défini dans une classe et qui désigne un autre objet. Une racine de persistance est définie par un couple (nrp, CA) où nrp est un identificateur qui donne le nom de la racine, et CA la classe applicative qui détermine la classe de l'objet désigné par la racine. Bien entendu, l'objet désigné effectivement par la racine nrp peut appartenir à une classe sous-type de la classe CA (relation «est-un»).

L'objet désigné par une racine de persistance peut être partageable, entre la racine de persistance et des objets applicatifs, ou non partageable. Dans ce dernier cas, l'objet désigné par la racine de persistance ne peut pas être désigné directement par une autre racine ou par un objet. Graphiquement, une racine de persistance est représentée par :

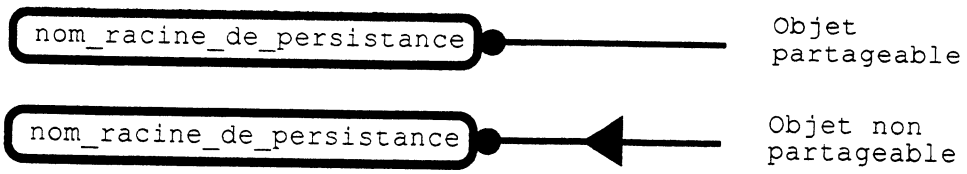


Figure 5-22 : Représentation graphique d'une racine de persistance.

```
classe projet Gestion_Compétitions_Sportives
  racines de persistance        (*4 racines de persistance*)
       compétitions : Ensemble(réf Compétition)
       compétition_courante : réf Compétition
       clubs : Ensemble(réf Club)
       licenciés : Ensemble(réf Licencié)
fin
```

Exemple 5-17 : Classe projet de l'application «Gestion des compétitions sportives».

Nous n'abordons pas ici la délicate question du choix des racines de persistance. En particulier, nous ne savons pas actuellement si il est souhaitable de minimiser le nombre des racines ou au contraire d'associer une racine de persistance à chaque classe applicative dont les objets doivent persister. Il est certain que la réponse à cette question dépend du système d'applications à développer, du choix des informations à extraire et des chemins d'accès nécessaires. Une plus grande expérience de la méthode est nécessaire avant d'apporter une réponse plus précise à cette question.

7.3 Schéma de la base d'objets

La définition des données persistantes et des liens persistants qui les relient est une information importante pour le concepteur et constitue un des apports des bases de données. Par analogie, la notion de schéma de base d'objets est défini dans MOSAÏC pour mettre en évidence l'organisation des données persistantes et des liens persistants.

Nous rappelons que nous avons retenu le principe d'indépendance entre la notion de classe et la persistance : une classe applicative peut générer à la fois des objets persistants et des objets temporaires. Par ailleurs, nous avons donné la priorité à l'encapsulation sur la persistance : un objet applicatif peut encapsuler des méthodes, des données temporaires (variables) et des données «potentiellement persistantes» (attributs). La mise en évidence des aspects persistants n'est donc pas aussi simple que dans le modèle relationnel dans lequel tous les nuplets d'une relation non virtuelle persistent et toutes les valeurs d'un nuplet persistent.

Le **schéma de la base d'objets** d'une classe projet, noté SBO, est une vue sur les classes applicatives. Les classes retenues dans cette vue sont les classes applicatives accessibles directement ou indirectement à partir des racines de persistance de la classe projet considérée suivant les relations structurelles. Le polymorphisme, rend nécessaire de retenir, dans le SBO, les classes héritant d'une classe du SBO par une relation «est-un». En revanche, les seules informations pertinentes dans ce contexte sont les attributs des classes du SBO ; les variables ne sont pas retenues.

Plus précisément, une classe applicative C appartient au schéma de la base d'objets d'une classe projet P si :

- C est une classe désignée par une racine de persistance de P,
- C est accessible directement ou indirectement par les attributs d'une classe appartenant au SBO de P,
- C hérite par une relation «est-un» d'une classe appartenant au SBO de P.

La figure 5-23 donne un exemple de schéma de base d'objets. Les objets qui persistent dans un projet sont nécessairement instances de classes appartenant au schéma de la base d'objets. Par contre, les objets applicatifs temporaires utilisés pendant l'exécution d'une application peuvent être instances de classes applicatives n'apparaissant pas dans le SBO. Par ailleurs, comme la persistance est indépendante de la notion de classe, les classes applicatives appartenant au SBO peuvent générer à la fois des objets temporaires et des objets persistants.

L'intérêt d'une telle définition est bien entendu limité si on ne propose pas un outil informatique permettant, pour une classe projet et un ensemble de classes applicatives, de construire automatiquement le SBO, de le présenter au concepteur et de lui permettre de travailler interactivement sur le SBO. Un tel outil est présenté dans le chapitre suivant.

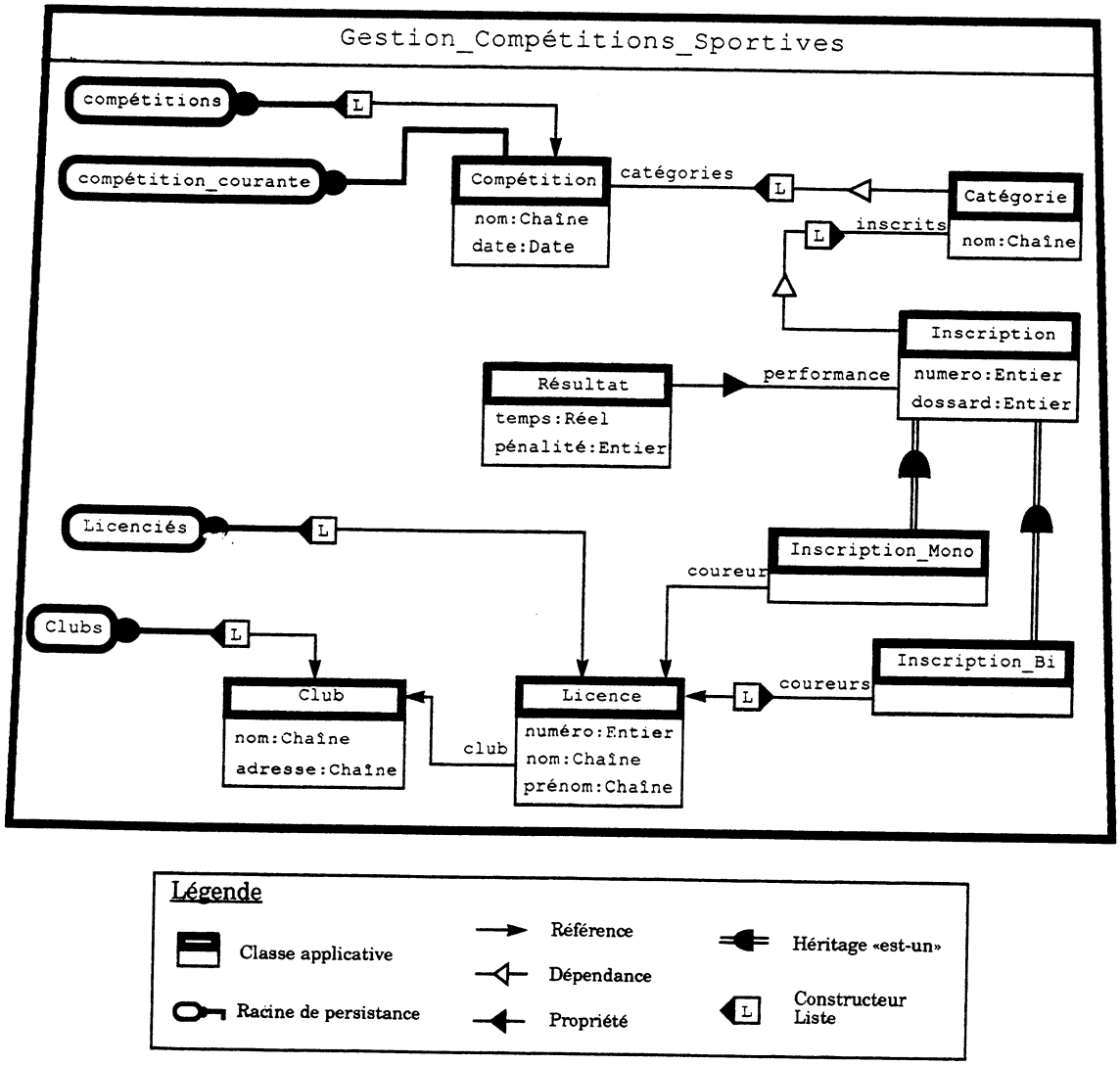


Figure 5-23 : Schéma de la base d'objets de la classe projet `Gestion_Compétitions_Sportives`.

7.4 Contrainte méthodologique

Les attributs d'un objet sont visibles de toutes les méthodes de cet objet. De la même manière, les racines de persistance sont visibles de tous les objets encapsulés dans un projet, qu'ils soient persistants (objets applicatifs) ou temporaires (objets applicatifs ou objets interactifs). Tout objet d'un projet peut donc envoyer directement un message à l'objet désigné par un racine de persistance. Cependant, un tel message introduit une dépendance fonctionnelle entre la classe de l'objet émetteur du message et la classe projet dans laquelle la racine est définie.

Par exemple, dans l'application `Gestion_Compétitions_Sportives` (cf. exem-

ple 5-17 et figure 5-23), si la classe applicative `Catégorie` utilise la racine de persistance `compétitions` ou la racine `compétition_courante`, cette classe applicative est dépendante du projet `Gestion_Compétitions_Sportives` et ne peut être réutilisée dans un projet n'offrant pas la même racine de persistance. Pour limiter les dépendances fonctionnelles et améliorer la réutilisation des classes nous proposons de limiter l'accès des racines de persistance.

Règle méthodologique : Les classes interactives peuvent envoyer des messages aux racines de persistance en revanche, les classes applicatives ne peuvent pas envoyer de message aux racines de persistance.

Les classes applicatives sont ainsi indépendantes des classes projets et elles peuvent être réutilisées plus facilement dans différentes classes projets. Cependant, les objets applicatifs peuvent avoir besoin d'accéder aux objets désignés par les racines de persistance. La règle méthodologique que nous avons retenu n'interdit pas un tel accès. Lorsqu'une méthode M d'un objet applicatif doit accéder à l'objet désigné par une racine de persistance, ce dernier doit alors être passé en paramètre à M lors de son appel.

8 Classes projets et applications

8.1 Racines d'application

Les applications encapsulées dans les projets sont représentées par des racines d'application. Une **racine d'application** représente une application et elle est définie par un couple (nra, Cin) où nra est le nom de la racine d'application et Cin le nom de la classe interactive qui est instanciée au lancement de cette application. Ce premier objet interactif constitue le premier module de l'application.

Figure 5-24 : Représentation graphique d'une racine d'application.

L'exemple 5-18 présente une classe projet pour le système d'applications `Gestion_Compétitions_Sportives` dans laquelle trois applications ont été définies : `inscription`, `dossard` et `course`. Les classes `Compétition`, `Club` et `Licencié` sont des classes applicatives et les classes `Init_inscription`, `Init_dossard` et `Init_course` sont des classes interactives.

```
classe projet Gestion_Compétitions_Sportives

  racine de persistance
        compétitions : Ensemble(réf Compétition)
        compétition_courante : réf Compétition
        clubs : Ensemble(réf Club)
        licenciés : Ensemble(réf Licencié)

  racine d'application
        inscription : Init_inscription
        dossard : Init_dossard
        course : Init_course

fin
```

Exemple 5-18 : Racines de persistance et racines d'applications de la classe projet `Gestion_Compétitions_Sportives`.

Dans une classe projet, il peut exister plusieurs racines de persistance typées par une même classe applicative. Ces différentes racines de persistance peuvent représenter le même objet applicatif ou des objets applicatifs différents. Par exemple, on peut définir différents groupes de personnes en définissant une racine de persistance pour chaque groupe. Chacune de ces racines donne alors accès à des objets distincts mais instances d'une même classe : `Ensemble (Personnes)`. En revanche, on ne peut pas définir plusieurs racines d'application typées par la même classe interactive car une telle situation correspondrait à la définition de plusieurs applications identiques.

8.2 Classe initiale

A l'exécution, une application est composée d'objets interactifs et d'objets applicatifs. Le lancement d'une application correspond donc à l'exécution d'une première méthode sur un objet. Ce premier objet est instance de la classe interactive, que nous appelons classe initiale, désignée par la racine d'application de l'application déclenchée.

Une **classe initiale** est une classe interactive dotée d'une méthode particulière appelée «`lancer_application`» qui contient le code exécuté au lancement de l'application. Nous ne proposons pas de représentation textuelle ou graphique particulière pour la notion de classe initiale.

La méthode `lancer_application` est à rapprocher de la notion de programme principal dans les applications développées à l'aide d'un langage de programmation conventionnel. Le rôle principal de cette méthode est de mettre en place les premiers éléments du dialogue proposé à l'utilisateur (menu principal, fenêtre de lancement, etc.), d'initialiser les classes applicatives utilisées dans l'application et enfin de «donner le contrôle» à l'utilisateur. Ce dernier dispose alors de l'initiative du dialogue et il peut déclencher des opérations via les méthodes des facettes présentations des objets interactifs affichés à l'écran. Ces méthodes doivent être conçues pour réaliser le traitement attendu par l'utilisateur puis elles doivent rendre le contrôle à l'utilisateur.

```
classe projet Gestion_Compétitions_Sportives
  racine de persistance
       compétitions : Ensemble(réf Compétition)
       compétition_courante : réf Compétition
       clubs : Ensemble(réf Club)
       licenciés : Ensemble(réf Licencié)
  racine d'application
       gestion_compétition : Init_compétition
fin


classe interactive Init_compétition (*Classe Initiale*)
  méthode lancer_application (*méthode exécutée au lancement*)
     prologue
     (*passe la main à l'utilisateur*)
     épilogue
  fin

  méthode prologue  (*détermine la course courante, initialise les classes*)
                    (*applicatives, instancie la classe interactive qui*)
                    (*gère le menu général et affiche ce menu général*)
  fin

  méthode épilogue (*ferme toutes les fenêtres*)
  fin
fin
```

Exemple 5-19 : Exemple de classe initiale.

Dans l'exemple 5-19, la classe projet comporte une seule application : `gestion_compétition` typée par la classe interactive initiale `Init_compétition`. Le corps des deux méthodes `prologue` et `épilogue` dépend largement des logiciels de base cibles.

Nous n'avons pas considéré les opérations de bas niveau concernant la gestion de l'écran telles que la gestion des fenêtres (recadrages, recouvrements, etc.), la distribution des événements en provenance de l'utilisateur, etc. Ces opérations, qui dépendent largement du système cible, sont généralement d'une grande complexité. Pendant la phase de conception, il est nécessaire de s'abstraire de ces considérations techniques de bas niveau pour adopter une approche plus globale du fonctionnement de l'application.

8.3 Interface interactive d'une application

A l'exécution, l'interface interactive d'une application est composée d'un ensemble d'objets interactifs. Construire l'interface d'une application nécessite donc d'élaborer l'ensemble des classes interactives utilisées pendant le fonctionnement de l'application. On peut définir l'interface interactive d'une application selon deux points de vue complémentaires : un point de vue structurel et un point de vue fonctionnel.

Définition structurelle : une classe interactive C appartient à l'interface interactive d'une application désignée par une racine d'application R si :

- C est la classe interactive initiale désignée par R,

- C est accessible directement ou indirectement par les attributs d'une classe de l'interface interactive,
- C hérite par une relation «est-un» d'une classe de l'interface.

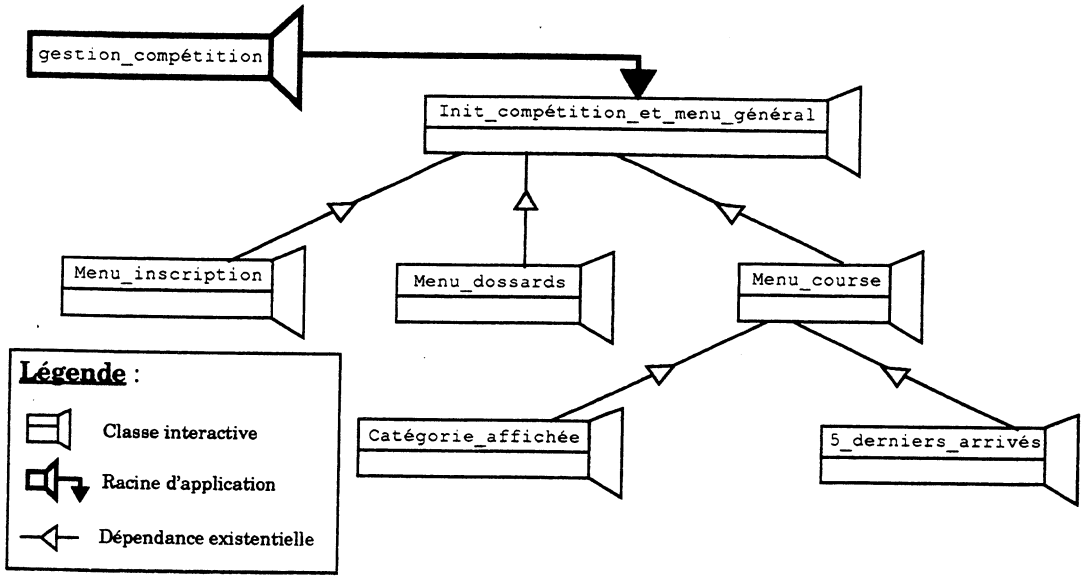


Figure 5-25 : Définition structurelle de l'interface interactive de l'application `gestion_compétition`.

Cette première définition est intéressante car elle donne un point de vue global de l'organisation de l'interface mais elle ne fournit pas nécessairement la liste exhaustive des classes interactives utilisées à l'exécution. En effet, une classe interactive utilisée seulement dans une méthode par le biais d'une variable locale de cette méthode n'apparait pas dans le graphe structurel de l'interface. La définition fonctionnelle est plus précise et elle permet de déterminer toutes les classes interactives susceptibles d'être utilisées pendant le fonctionnement de l'application.

Définition fonctionnelle : une classe interactive C appartient à l'interface interactive d'une application désignée par une racine d'application R si :

- C est la classe interactive initiale désignée par R,
- C est accessible directement ou indirectement par une relation de dépendance fonctionnelle à partir d'une classe de l'interface interactive,
- C hérite par une relation «est-un» d'une classe de l'interface.

Les vues partielles comme les graphes fonctionnels des méthodes permettent d'affiner et de compléter l'interface interactive. Par exemple, les graphes fonctionnels élaborés à partir des méthodes d'une présentation d'une classe interactive permettent de mettre en évidence les conséquences de leur activation sur l'interface interactive.

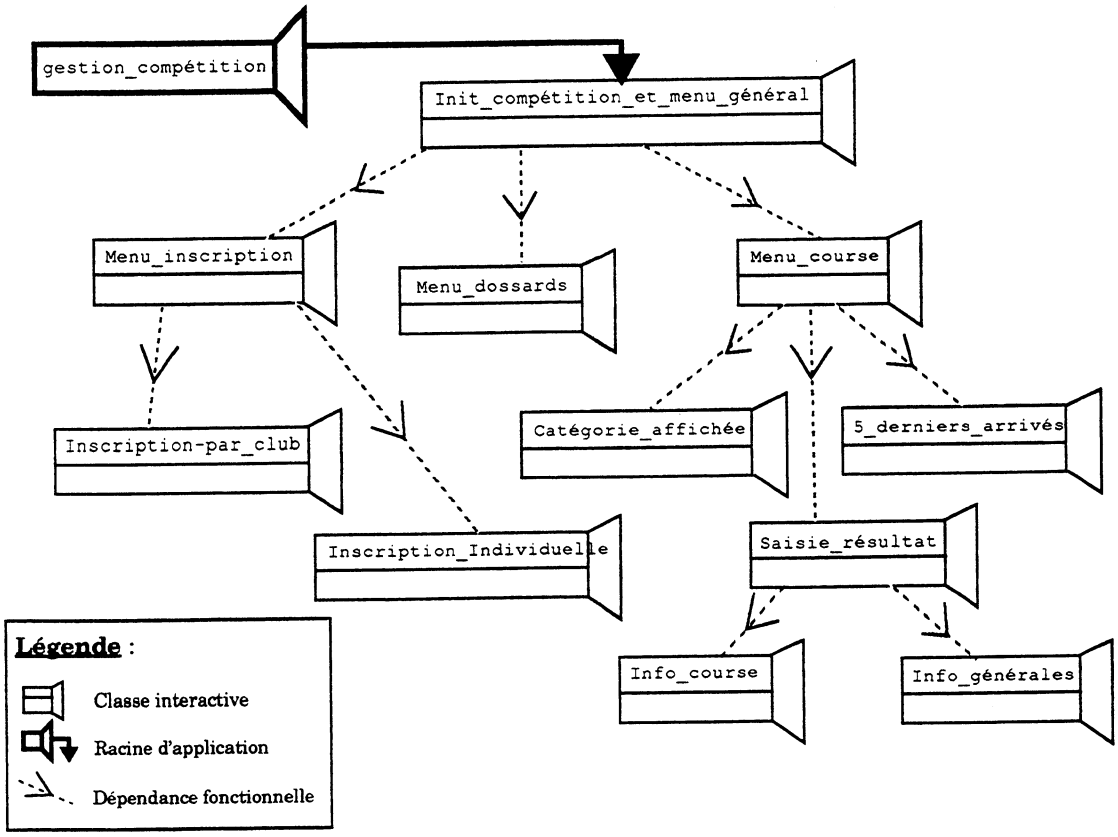


Figure 5-26 : Définition fonctionnelle de l'interface interactive de l'application `gestion_compétition`.

8.4 Corps d'une application

Le corps d'une application est défini de façon analogue à l'interface interactive.

Définition : Soit une application définie par une racine d'application R et dont l'interface interactive est un ensemble de classes interactives noté II. Une classe applicative A appartient au corps, noté CO, de l'application désignée par R si :

- il existe une dépendance fonctionnelle entre une classe interactive de II et A,
- ou il existe une dépendance fonctionnelle entre une autre classe du corps CO et A,
- ou A hérite par une relation «est-un» d'une classe du corps CO.

Les relations structurelles entre les classes interactives reflètent l'organisation structurelle du gestionnaire de dialogue d'une application interactive alors que les relations structurelles entre les classes applicatives reflètent l'organisation du domaine réel

indépendamment des applications interactives. Il ne peut donc pas exister de définition structurelle du corps d'une application équivalente à la définition structurelle de l'interface interactive d'une application puisque les relations structurelles entre classes applicatives n'apportent aucune information sur l'organisation des applications.

Comme le montre la figure 5-27, le graphe fonctionnel élaboré à partir d'une méthode ou de plusieurs méthodes d'une classe interactive permet de compléter et d'affiner la définition du corps en limitant le nombre de classes considérées.

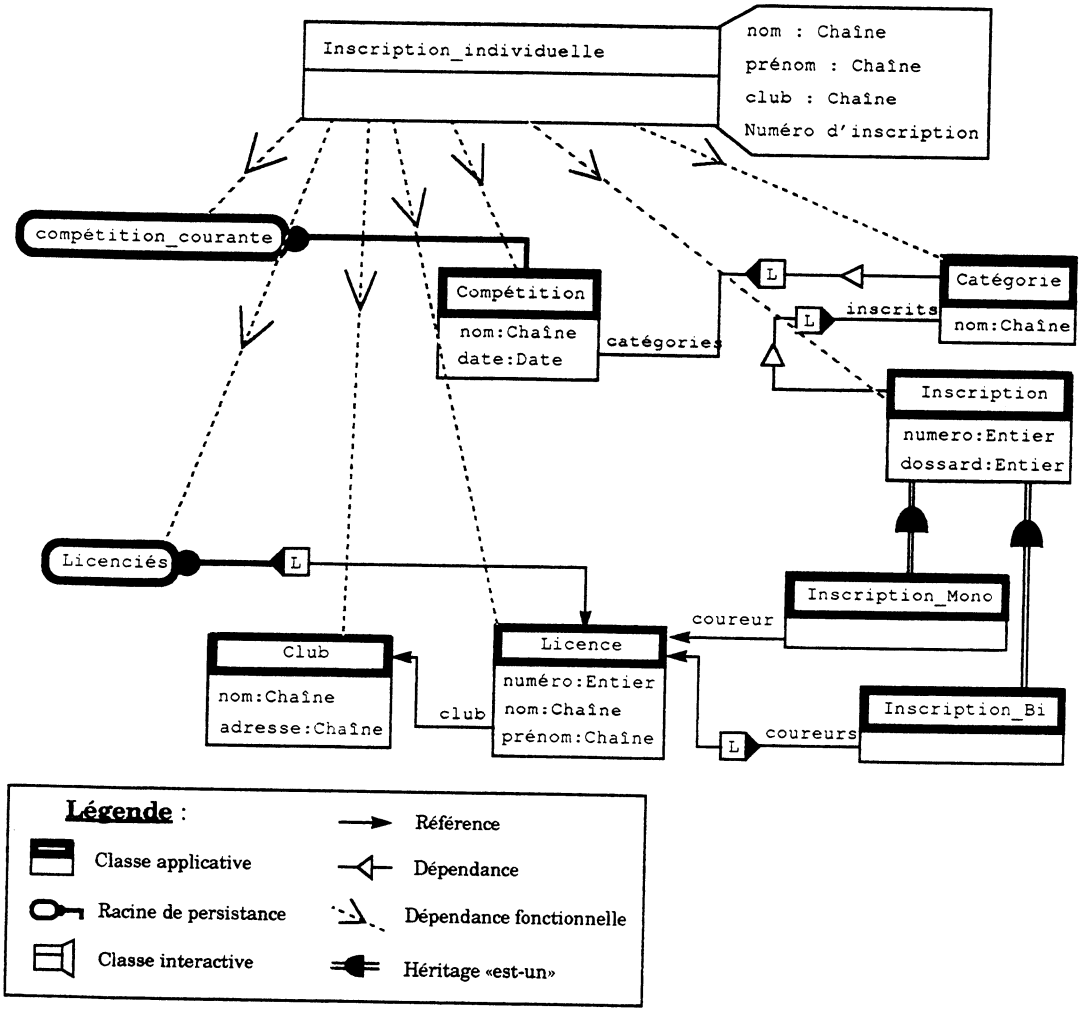


Figure 5-27 : Dépendances fonctionnelles de la classe `Inscription_individuelle`.

8.5 Masquage de l'interface interactive

Dans le chapitre 4, nous avons souligné la nécessité de limiter les échanges entre l'interface interactive et le corps d'une application selon les règles suivantes :

- Une classe interactive peut librement référencer, par ses attributs et ses varia-

bles, des classes applicatives et utiliser, par des messages, des classes applicatives.

- Une classe applicative peut interroger l'utilisateur via un ensemble limité de services qui masquent l'organisation de l'interface interactive. Les classes applicatives ne peuvent donc pas référer ni utiliser directement les classes interactives.

Pour le corps d'une application, l'interface interactive est donc masquée par un ensemble de services. Lorsque c'est nécessaire, le corps de l'application peut donc prendre l'initiative du dialogue avec l'utilisateur et l'interroger sans que les classes applicatives soient pour autant dépendantes fonctionnellement des classes interactives. Ces règles ont été retenues à cause de la différence d'évolutivité entre le corps et l'interface interactive d'une application (cf. chapitre 4).

Chaque service requis par le corps d'une application sur l'interface interactive est représenté par une méthode sur une classe particulière que nous appelons **classe d'interrogation**. Chaque classe d'interrogation contient les méthodes appelées par une méthode primaire. Une classe d'interrogation est toujours virtuelle : elle n'est jamais instanciée et les méthodes qu'elle contient n'ont pas de corps. Chaque classe d'interrogation doit être spécialisée par une relation «est un» dans une classe interactive pour définir les corps des méthodes appelées par une méthode primaire d'une classe applicative du corps de l'application. Pour le corps, l'interface interactive est donc masquée par un ensemble de classes d'interrogation. La figure 5-28 illustre la réalisation concrète de ce masquage à l'aide de la relation de sous-typage (héritage «est-un»).

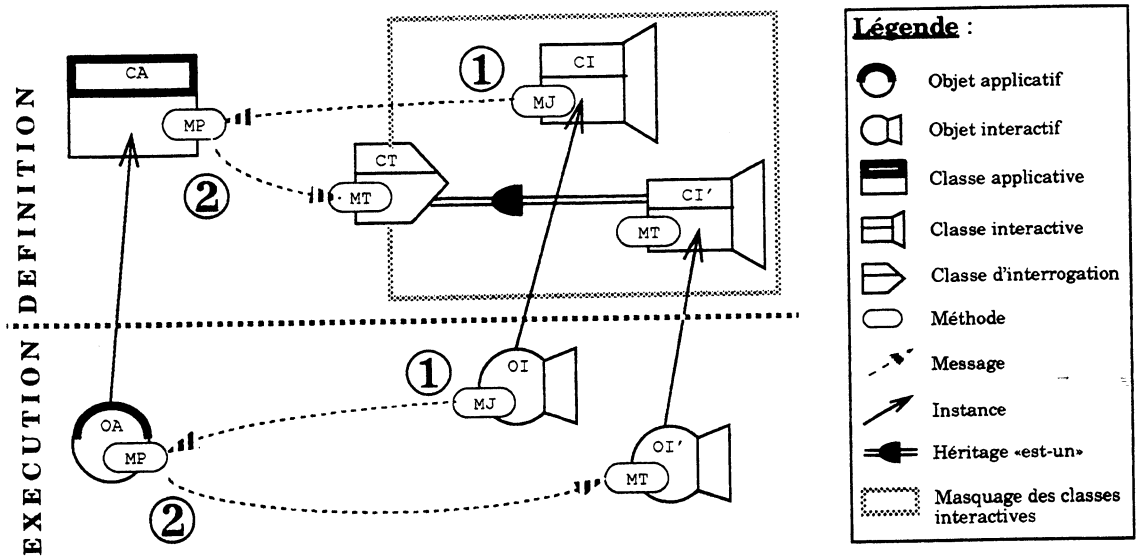


Figure 5-28 : Encapsulation de l'interface interactive.

Considérons une méthode primaire MP définie dans une classe applicative CA et appelée par une méthode MJ définie dans une classe interactive CI. Si les paramètres nécessaires au déroulement complet de la méthode MP ne peuvent pas tous être déter-

minés avant l'appel, MP doit interroger l'utilisateur pour compléter les paramètres.

On définit une classe d'interrogation CT pour traiter les interrogations de MP vers l'utilisateur. CT contient, sous forme de méthodes, les requêtes posées par MP à l'utilisateur. Cette classe CT est spécialisée dans une classe interactive CI' qui n'est pas nécessairement distincte de CI. Dans CI', le concepteur définit le traitement effectué par la méthode MT héritée de la classe CT.

La signature de MP est complétée par un paramètre qui désigne l'objet interactif interrogé par MP à l'exécution. Cet objet interactif contient la méthode MT chargée de collecter auprès de l'utilisateur les paramètres manquant à MP. Le type du paramètre complémentaire est CT et le polymorphisme est utilisé à l'exécution pour passer en argument un objet de CI' sous-type de CT.

Ainsi, les classes applicatives sont dépendantes fonctionnellement des classes d'interrogation mais pas des classes interactives. La modification des classes interactives ou la modification de l'organisation de l'interface interactive n'affectent pas le corps de l'application. Par ailleurs, la classe CA et la méthode MP peuvent être réutilisées dans une autre application, il suffit de définir dans cette autre application, une classe interactive héritant de la classe d'interrogation CT.

La mise en œuvre du principe de masquage de l'interface interactive peut paraître complexe. On rappelle cependant que les prises de contrôle de l'initiative du dialogue par le corps doivent rester aussi limitées que possible pour laisser l'utilisateur libre de la conduite du dialogue. Pour cette raison, nous n'avons pas souhaité définir de nouveau concept.

Nous avons considéré ici uniquement les accès à l'interface provenant directement ou indirectement d'une action déclenchée par l'utilisateur. Nous n'avons pas approfondi les accès requis par des événement ne provenant pas de l'utilisateur comme : un top d'horloge, l'insertion d'une disquette, etc.

8.6 Conflit entre deux règles méthodologiques

L'utilisation de MOSAÏC fait apparaître un conflit entre deux règles méthodologiques : la séparation entre le corps et l'interface interactive et le principe de localité et d'autonomie (cf. chapitre 3). Nous illustrons ce conflit à l'aide d'un exemple qui porte sur la phase de déroulement du concours dans l'application «Gestion de compétitions Sportives».

Pendant le déroulement du concours, l'utilisateur de l'application peut saisir les performances d'un compétiteur, afficher le classement d'une catégorie, imprimer ce classement, etc. Un classement général est établi pour l'ensemble des coureurs indépendamment des catégories. Pendant le déroulement de la compétition, une fenêtre présentant le classement général est affichée en permanence. Le maintien de ce classement général tout au long du déroulement du concours est une compétence qui dépend du domaine d'application et non du dialogue. Ce classement doit donc être réalisé par une classe applicative, par exemple par la classe `Compétition`.

Pour obtenir ce classement général indépendant des catégories, on définit sur la classe `Compétition` une fonction dont la signature est : `classement_général : Liste(réf Inscription)`, comme le montre l'exemple 5-20. A l'exécution, cette liste peut comporter à la fois des inscriptions individuelles (`Inscription_Mono`) et des inscriptions en équipage (`Inscription_Bi`).

```
classe applicative Compétition
  exporte fonction nom, catégories, classement_général
  attribut      nom : Chaîne
                catégories : Liste(réf Catégorie)
  variable
       classement_général : Liste (réf Inscription)
fin

classe applicative Catégorie
  attribut      nom : Chaîne
                inscrits : Liste(réf Inscription)
fin

classe applicative Inscription
  attribut      numéro_inscription : Entier
                dossard : Entier
fin

classe applicative Inscription_Mono      est-un Inscription
  attribut      coureur : réf Licence
fin

classe applicative Inscription_Bi        est-un Inscription
  attribut      coureurs : Ensemble (réf Licence)
  invariant card (coureurs = 2)
fin
```

Exemple 5-20 : Classement général de l'ensemble des compétiteurs.

Considérons maintenant que l'on souhaite afficher différemment les `Inscription_Mono` et les `Inscription_Bi` : on affiche le `nom` et le `prénom` pour les premières et seulement les deux `noms` pour la seconde. Pour réaliser un tel affichage, on peut envisager deux organisations comme le montre l'exemple 5-21.

Dans la première solution, le principe de séparation entre l'interface interactive et le corps est respecté ; les classes applicatives n'effectuent pas d'entrées/sorties avec l'utilisateur. Par contre la classe `Classement_général` doit demander le type des objets avant d'agir puisque l'affichage diffère selon les objets. Les principes de la programmation orientée objet et plus particulièrement le principe d'autonomie et de localité ne sont pas respectés (cf. chapitre 3 sections 3.2 et 3.3). Par ailleurs, une telle organisation suppose l'existence d'une instruction permettant de connaître le type des objets à l'exécution notée `classe(i)` dans notre exemple.

Dans la seconde solution, le principe d'autonomie et de localité est respecté, les objets ne sont pas manipulés «au loin» et chaque objet applicatif gère lui-même son affi-

Solution n°1

```
classe interactive Classement_général
  ...
  pour tout i dans
  Compétition.classement_général
  faire
     si classe(i) = Inscription_Mono
     alors (*affichage des nom*)
        (*et prénom du coureur i*)
     sinon (*affichage des noms*)
          (*des deux licenciés de i*)
  ...
fin

classe applicative Compétition
  fonction classement_général
  variable
       classement_général : Liste
                  (réf Inscription)
fin

classe applicative Inscription
fin

classe applicative Inscription_Mono
  est-un Inscription
  attribut
     coureur : Licence
fin

classe applicative Inscription_Bi
  est-un Inscription
  attribut
     coureurs : ensemble (réf Licence)
  invariant card (coureurs = 2)
fin
```

Solution n°2

```
classe interactive Classement_général
  ...
  pour tout i dans
     Compétition.classement_général
        faire i.afficher_class_gén
  ...
fin

classe applicative Compétition
  fonction classement_général
  variable
       classement_général : Liste
                  (réf Inscription)
fin

classe applicative Inscription
  méthode afficher_class_gén
          (* vide *)
  fin_méthode
fin

classe applicative Inscription_Mono
  est-un Inscription
  attribut
     coureur : Licence
  méthode afficher_class_gén
     (* affichage des nom et prénom *)
     (* du coureur *)
  fin_méthode
fin

classe applicative Inscription_Bi
  est-un Inscription
  attribut
     coureurs : ensemble (réf Licence)
  invariant card (coureurs = 2)
  méthode afficher_class_gén
     (* affichage des noms des *)
     (* deux coureurs *)
  fin_méthode
fin
```

Exemple 5-21 : Deux organisations possibles pour l'affichage du classement général d'une compétition.

chage. On peut donc afficher différemment les `Inscription_Mono` et les `Inscription_Bi`. Cependant, dans ce cas, le principe fondamental de séparation entre interface interactive et corps d'une application est violé puisque les objets applicatifs effectuent des opérations d'entrée/sortie avec l'utilisateur.

Aucune de ces deux solutions n'est donc satisfaisante. Il existe une solution intermédiaire qui consiste à remplacer la méthode `afficher_class_gén` par une fonction retournant une chaîne de caractères valant «`nom+prénom`» dans `Inscription_Mono` et «`nom+nom`» dans `Inscription_Bi`. Cependant, cette solution n'est guère plus satisfaisante que les précédentes puisque les classes applicatives restent sensibles à une modification du format des informations affichées.

Il convient donc de trouver un compromis. Pour y parvenir, nous nous appuyons sur trois remarques :

- La remise en cause de la séparation entre interface interactive et corps conduit à fragiliser le corps en le rendant sensible à toute modification de l'interface.
- Ce conflit n'a été constaté qu'entre l'interface interactive et le corps des applications.
- La possibilité de tester la classe d'un objet peut être limitée aux cas où des objets interactifs utilisent des objets applicatifs.

Nous proposons donc la règle méthodologique suivante :

Règle méthodologique : Au sein du corps d'une application ainsi qu'au sein de son interface interactive, les classes et les méthodes doivent être conçues pour respecter le principe d'autonomie et de localité et on ne doit pas tester la classe d'un objet avant d'envoyer un message. Seuls les objets interactifs peuvent, lorsque c'est nécessaire, tester la classe des objets applicatifs qu'ils manipulent.

9 Conclusion

Dans ce chapitre, nous avons présenté en détail les concepts et les langages que nous proposons pour concevoir un système d'applications. Nous avons cherché d'une part à spécialiser les concepts de base des techniques à objets et d'autre part à les enrichir pour en faciliter l'utilisation. Ainsi nous avons défini différentes catégories de classes, de méthodes et de relations. Les relations structurelles sont clairement distinguées des relations fonctionnelles et une dimension dynamique les complète. Toutefois, les états, les règles d'intégrité et les assertions ont actuellement un rôle limité de documentation du schéma de conception. Un mécanisme de preuve basé sur la dimension dynamique ainsi que la prise en compte de la généricité sont à approfondir pour offrir une méthode plus complète. Les notions de schéma de base d'objets, de corps d'une application et d'interface interactive, habituellement définies et représentées physiquement

par des composants spécialisés bien individualisés ont dû être redéfinies pour prendre en compte les spécificités de notre approche.

L'exploitation des notions de «schéma de la base d'objets persistants», de corps et d'interface interactive est malaisée sans le recours à un outil automatisé pour extraire ces notions de l'ensemble des classes composant un système d'applications. La présentation d'un tel outil est l'objet du chapitre suivant. Auparavant, nous aurons explicité la programmation d'une conception MOSAÏC en O2 puis en PEPLOM et présenté brièvement une démarche de conception.

Chapitre 6

Outils de conception et éléments d'une démarche

1 Introduction

Pour être utilisable efficacement, une méthode de conception doit assister le programmeur dans sa démarche de conception et dans la production des programmes exécutables. Lorsque la méthode se limite à un ensemble de règles d'utilisation d'un langage de programmation ou d'un système particulier de gestion de bases de données, la production des programmes est immédiate. En revanche, si la méthode de conception s'appuie sur un modèle qui lui est propre, elle doit proposer des règles de traduction des concepts de ce modèle vers des langages ou des SGBD opérationnels. Notre volonté a été de nous appuyer avant tout sur des règles méthodologiques de conception et nous avons défini MOSAÏC indépendamment d'un LPBO particulier. Dans les sections 2 et 3 de ce chapitre, nous montrons comment traduire un schéma de conception MOSAÏC pour obtenir un programme exécutable ou plus précisément un squelette d'application écrit en O_2 [ODeu89] et en PEPLOM [Dech93].

Pour guider le concepteur dans l'élaboration d'un schéma de conception, une démarche doit compléter les modèles et les langages proposés. Une telle démarche comporte plusieurs étapes totalement ou partiellement ordonnées. Dans MOSAÏC, nous nous sommes limités à l'élaboration d'une solution technique répondant aux spécifications des applications informatiques sans aborder la spécification complète du système d'information ni les contraintes organisationnelles imposées par sa mise en place. La méthode MERISE en revanche couvre tout le cycle de vie d'un système d'information. Comme cette méthode est maintenant largement connue et utilisée en France, il nous a semblé intéressant d'étudier les possibilités d'intégration de MOSAÏC dans MERISE. La section 4 présente quelques éléments d'une démarche de conception adaptée à MOSAÏC et propose une intégration avec MERISE.

Les outils automatisés constituent le quatrième élément d'une méthode de conception après les modèles, les langages et la démarche. Les outils sont très variés, depuis le simple outil d'édition et d'archivage des documents de conception jusqu'au système expert susceptible de suggérer des éléments de solutions au concepteur en passant par les outils de vérification, de simulation ou de génération de code. Nous avons vu dans les chapitres 4 et 5 que la notion de classe a un rôle central dans MOSAÏC et qu'elle regroupe des informations de natures différentes : structurelles, fonctionnelles et dynami-

ques. De nombreuses autres notions sont «calculées» à partir des classes : le schéma de la base d'objets, l'interface interactive d'une application, son corps, etc. Pour exploiter plus facilement ces différentes notions et pour maîtriser la complexité des classes, l'assistance d'un outil automatisé est indispensable. Par ailleurs, la phase de traduction vers un système cible est souvent laborieuse et répétitive. Une grande part de cette tâche peut être prise en charge par un outil pour produire un programme complet ou un squelette d'application qu'il est nécessaire ensuite de compléter. Dans la section 5, nous présentons les spécifications et la réalisation de l'outil OCAPI d'aide à la conception.

2 Programmation O_2 d'une conception MOSAÏC

2.1 SGBDOO O_2

Dans le modèle O_2[1] [ODeu89], la notion d'objet est distincte de celle de valeur. Une **valeur** est une donnée non partageable appartenant à un type de base ou à un **type** (type concret) défini par une combinaison de constructeurs appliqués à des types de base. Un **objet** est une instance partageable d'une classe. Une **classe** comporte une structure de données (type) et des méthodes. Les méthodes d'une classe peuvent être publiques ou privées et les éléments de la structure de données peuvent être publics (lecture/écriture ou lecture seule) ou privés. L'utilisation de l'héritage est réservée aux classes. L'héritage définit une relation de sous-typage et suit les règles de covariance.

Un **schéma** regroupe un ensemble de classes, de types, d'objets nommés, de procédures et d'applications. Une **base** est une «instance persistante» d'un schéma. Contrairement aux modèles de données habituels, une classe O_2 ne regroupe pas les objets qu'elle a générés. Un **objet nommé** est un objet défini dans un schéma et accessible directement par son nom à partir de toutes les classes du schéma. Les objets nommés forment les racines de persistance et la persistance se propage ensuite par les références entre objets. Les objets nommés constituent les points d'entrée dans la base d'objets persistants. Un schéma peut **exporter** (resp. **importer**) vers (resp. depuis) d'autres schémas des classes et des types.

Une **application** est définie dans un schéma et peut être lancée par l'utilisateur. Une application est composée de variables locales à l'application et de transactions. Les **variables** sont visibles et accessibles seulement des transactions et leur durée de vie est égale à celle de l'application. Une **transaction** réalise une suite atomique d'opérations sur la base d'objets. Les transactions publiques peuvent être activées par l'utilisateur via un menu présenté automatiquement par O_2. En revanche, aucune transaction ne peut être déclenchée par des objets. O_2 offre un ensemble de classes spécialisées pour faciliter la programmation de l'interface entre l'application et l'utilisateur (boutons, choix, boîtes de dialogue, etc.).

---

1. Dans cette section, nous nous appuyons sur la version 3.3.1 (1992) du SGBDOO O2.

```
schema Gestion_d'une_Compétition
  named object compétition_courante : Compétition

  application Inscription
    variables...
    program    inscription_individuelle,
               annulation_d'une_inscription,
               inscription_par_club...
  end (*Inscription*)

  class Compétition
    type tuple(  catégories : set (Catégorie)...
               )
       method ajout_catégorie (c : Catégorie)...
  end (*Compétition*)

end (*Gestion_d'une_Compétition*)
```

Exemple 6-1 : Schéma O_2 de l'application «Gestion de compétitions sportives».

2.2 Principes généraux de programmation

Nous présentons dans cette section et dans les suivantes, la technique proposée pour programmer un schéma de conception MOSAÏC à l'aide du SGBDOO O_2.

Les types de base et les constructeurs O_2 permettent de définir des valeurs simples ou composées. En MOSAÏC, les classes de base et les classes constructeurs génèrent des objets non partageables qui sont aisément représentables à l'aide des types de base et des constructeurs O_2.

Dans MOSAÏC, le caractère partageable dépend des relations entre objets et tout objet, quelque soit sa classe, peut être propriété propre d'un autre objet. En O_2 en revanche, un objet est toujours partageable. La propriété ainsi que la dépendance existentielle doivent donc être simulées.

En MOSAÏC la propriété indique que l'objet o désigné ne peut pas être référencé ni modifié par un autre objet que son propriétaire ; mais il peut être consulté si l'attribut qui le désigne est public. Cette relation est simulée en O_2 en dupliquant l'objet propre o chaque fois qu'il apparaît dans une affectation comme «`att:=o`» et chaque fois qu'il est passé en paramètre dans un message comme «`meth(o)`». Ainsi l'objet o n'est pas partagé.

La dépendance existentielle est plus délicate à mettre en œuvre. Elle impose, lorsqu'un objet o dépend existentiellement d'un objet o', de maintenir la liaison réciproque de toute référence d'un autre objet o" vers o. Cette liaison réciproque est utilisée pour mettre à jour l'objet o" lorsque o est supprimé.

O_2 et MOSAÏC adoptent des approches différentes pour l'héritage. O_2 lie l'héritage au sous-typage et suit les règles de covariance alors que MOSAÏC distingue le sous-typage et la réutilisation (adapte) et suit les règles de contravariance. Comme en prati-

que, la possibilité de généralisation des arguments permise par la contravariance est rarement utilisée, le sous-typage MOSAÏC peut être programmé à l'aide de l'héritage O_2. La programmation de la relation d'adaptation MOSAÏC dépend des modifications introduites dans la sous-classe. Si ces modifications respectent les règles d'héritage d'O_2 alors, ce mécanisme peut être utilisé sinon, le texte de la surclasse doit être recopié puis modifié.

Les classes facettes MOSAÏC sont représentées par des classes O_2.

Dans MOSAÏC, les fonctions permettent l'accès aux données d'un objet, O_2 offre la possibilité de limiter l'accès aux éléments de la structure de données d'un objet à la seule lecture.

MOSAÏC		O_2
Classe	⟶	Classe
Classe de base	⟶	Type de base
Constructeur	⟶	Constructeur
Classe facette	⟶	Classe
Référence	⟶	Référence
Dépendance	⟶	Référence + lien réciproque
Propriété	⟶	Référence privée (+ méthode)
Fonction	⟶	Attribut en lecture seule
Héritage «est-un»	⟶	Héritage
Héritage «adapte»	⟶	Héritage | Duplication de code

Figure 6-1 : Règles générales de traduction MOSAÏC-O_2.

Nous étudions maintenant plus précisément la programmation des applications.

2.3 Programmation des classes projets

Les notions de classe projet et de projet de MOSAÏC sont très proches de celles de schéma et de base d'O_2. Les racines de persistance MOSAÏC sont traduites sous forme d'objets nommés O_2 et les racines d'application sous forme d'applications O_2. Notons que MOSAÏC permet de préciser des contraintes sur le partage ou non des objets accessibles via les racines de persistance. La traduction des applications est étudiée dans la section suivante.

```
<--MOSAÏC-

classe projet Nom_projet
  racine de persistance
```

```
        nom_racine_persistante1 : réf Nom_classe_applicative1
        nom_racine_persistante2 : prop Nom_classe_applicative2
  racine d'application
        nom_application : Nom_classe_initiale
fin_classe_projet


-02-->

schema Nom_projet
  named object nom_racine_persistante1 : Nom_classe_applicative1;
  named object nom_racine_persistante2 : Nom_classe_applicative2;

  application nom_application
     ...
  end
end
```

Exemple 6-2 : Représentation d'une classe projet à l'aide d'un schéma O_2.

2.4 Programmation des applications MOSAÏC

Une application O_2 est composée d'un ensemble de transactions et de variables accessibles seulement depuis le corps des transactions. Elle possède éventuellement deux transactions particulières «`init`» et «`exit`» exécutées automatiquement en début et en fin d'exécution de l'application. L'activation de l'application par l'utilisateur provoque la présentation d'un menu composé des différentes transactions qui composent l'application. Ces transactions sont alors activables par l'utilisateur mais en aucun cas elles ne peuvent être activées par des objets. En MOSAÏC, une application est définie par une racine constituée par une classe initiale munie d'une méthode «`lancer_application`». L'activation de l'application a pour effet d'instancier la classe initiale et d'exécuter la méthode `lancer_application`. On peut envisager deux solutions pour programmer en O_2 une application MOSAÏC.

La première solution est applicable lorsque l'application MOSAÏC comporte un menu principal composé d'actions qui peuvent constituer des transactions, elle peut alors être représentée par une application O_2. Le traitement réalisé dans la méthode `lancer_application` est transféré dans la transaction `init` et chaque action du menu principal de l'application MOSAÏC est programmé comme une transaction de l'application O_2. Cette solution comporte cependant des inconvénients. La granularité des transactions peut s'avérer trop importante lorsque par exemple les actions présentées dans le menu principal sont complexes et font de nombreuses modifications sur les objets persistants. Par ailleurs, le programmeur ne peut pas contrôler finement l'évolution de l'écran. Notamment, il ne peut pas interdire dynamiquement l'activation d'une transaction du menu principal de l'application O_2 ou modifier son apparence.

La seconde solution consiste à représenter la classe initiale d'une application MOSAÏC par une classe O_2. L'application O_2 est alors réduite à une transaction unique programmée dans la transaction «`init`» et dont le rôle est de créer un objet de la classe initiale et d'activer la méthode `lancer_application`. Cette solution a l'avantage de

laisser le programmeur maître de l'évolution de l'interface : il peut interdire ou autoriser temporairement et dynamiquement l'activation de n'importe quelle action du menu principal. En revanche, la granularité de la transaction est moins fine que dans la première solution puisque cette fois, l'application complète constitue une seule transaction.

Aucune de ces deux solutions n'est donc pleinement satisfaisante et le programmeur doit choisir entre la première solution qui privilégie la finesse des transactions au détriment du respect des spécifications externes et la seconde qui permet de mieux maîtriser l'évolution du dialogue entre l'utilisateur et l'application mais conduit à des transactions de granularité peu fine.

On peut noter que dans une nouvelle version d'O_2, on peut déclarer une transaction à l'intérieur d'une méthode par deux instructions spécialisées : «`transaction`» et «`validate`» (cf. exemple 6-3). Cette fonctionnalité constitue une facilité considérable : le programmeur peut garder la maîtrise complète de l'évolution de l'écran (solution 2) et définir des transactions de granularité beaucoup plus fine que dans les deux solutions présentées. En effet, les méthodes primaires réalisent un traitement qui maintient la cohérence des objets applicatifs. Comme les messages envoyés par les classes interactives aux classes applicatives sont limités aux méthodes primaires, chaque message peut être encapsulé, dans la classe interactive qui l'envoie, dans une transaction comme le montre l'exemple 6-3.

```
schema Gestion_d'une_Compétition

  class Compétition                                  (*classe applicative MOSAÏC*)
     ...
     method affecter_les_dossards                    (*méthode primaire MOSAÏC*)
     ...
  end

  named object Compétition_courante : Compétition

  class Menu_attribution_des_dossards                (*classe interactive MOSAÏC*)
  ...
     method attribuer_les _dossards                  (*méthode traitant une action*)
        ...                                  (*déclenchable par l'utilisateur*)
        transaction;        (*début de la transaction*)
           Compétition_courante -> affecter_les_dossards
        validate;           (*fin de la transaction*)
        ...
     end
  end

end
```

Exemple 6-3 : Traduction d'un appel à une méthode primaire par une transaction O_2.

Cette solution efficace doit être utilisée avec attention car O_2 ne gère pas les transactions imbriquées. Il est donc nécessaire de limiter l'utilisation de ce mécanisme de transaction aux appels des classes interactives vers les classes applicatives puisque les méthodes primaires d'une classe applicative peuvent être librement utilisées par les autres classes applicatives. Il est également nécessaire de s'assurer que les appels des

classes applicatives vers les classes interactives, via les classes d'interrogation, ne déclenchent pas de nouvelles transactions.

2.5 Programmation des classes applicatives

Dans O_2, comme dans MOSAÏC, la persistance est indépendante de la notion de classe. En revanche, en O_2 tous les attributs d'un objet persistant persistent et propagent la persistance alors qu'en MOSAÏC, les variables des classes applicatives ne persistent pas et ne propagent pas la persistance.

La solution que nous avons retenue pour représenter une classe applicative MOSAÏC comportant à la fois des caractéristiques temporaires et des caractéristiques persistantes est d'utiliser deux classes O_2. La première classe regroupe les attributs et les méthodes persistants (cf. chapitre 5 section 4.5.2), la seconde classe regroupe les variables et les méthodes temporaires et ces deux classes sont reliées par deux références réciproques. Cependant, comme O_2 propage systématiquement la persistance par les références, il est nécessaire, en fin d'application, de supprimer les liens entre les objets de la première classe (objets persistantes) et leurs homologues de la seconde classe (objets temporaires). Pour cela on utilise un objet nommé et deux classes (cf. exemple 6-4). L'objet nommé `Objets_partiellement_temporaires` regroupe tous les objets applicatifs partiellement temporaires. La classe `Partie_Temporaire` représente la famille des classes O_2 décrivant les parties temporaires. La classe `O_partiellement_temporaire` représente la famille des classes O2 décrivant des objets applicatifs partiellement temporaires.

```
schema Nom_schéma

  named object Objets_partiellement_temporaires :
                                        set(O_partiellement_temporaire)
     method début_application  (*initialise l'ensemble à vide*)

     method fin_application    (*appelle supprimer_partie_temporaire sur*)
                               (*tous les objets de l'ensemble*)
  end


  class O_partiellement_temporaire       (*classe virtuelle*)
     type tuple (partie_temporaire : Partie_Temporaire)

     method promotion                 (*crée la partie temporaire*)
     method promu : boolean           (*vrai si self->partie_temporaire ≠ nil*)
     method supprimer_partie_temporaire (*supprime la partie_temporaire*)
  end


  class Partie_Temporaire                (*classe virtuelle*)
     type tuple (partie_persistante : O_partiellement_temporaire)
  end

end
```

Exemple 6-4 : Mécanisme de gestion des objets applicatifs partiellement temporaires en O2.

Une classe applicative en partie temporaire est alors représentée par deux classes O2. La première hérite de la classe `O_partiellement_temporaire` et la seconde de la classe `Partie_Temporaire`. Les attributs et les méthodes hérités doivent être redéfinis comme le montre l'exemple 6-4.

```
<--MOSAÏC-
 classe applicative Nom_classe_applicative
   attribut nom_attribut1 : réf Nom_classe1,          (*persistant*)
   variable nom_variable2 : prop Nom_classe2,         (*temporaire*)

   méthode nom_méthode1 persistante
   méthode nom_méthode2 temporaire
 fin


-O2-->
 class Nom_classe_applicative
                                (*regroupe les caractéristiques persistantes*)
   inherit O_partiellement_temporaire
   type tuple (
        nom_attribut1 : Nom_classe1,
        partie_temporaire : Nom_classe_applicative_partie_temporaire )
                                (*référence vers la partie temporaire*)

   method promotion  (*crée la partie temporaire et insère l'objet dans*)
                     (*l'objet nommé Objets_patiellement_temporaires*)

   method supprimer_partie_temporaire     (*supprime la partie temporaire :*)
                                          (*self->partie_temporaire = nil*)
   method nom_méthode1
 end


 class Nom_classe_applicative_partie_temporaire
                                (*regroupe les caractéristiques temporaires*)
   inherit Partie_Temporaire
   type tuple(
        nom_attribut2 : Nom_classe2,
        partie_persistante : Nom_classe_applicative)
                                (*référence vers la partie persistante*)
   method nom_méthode2
 end
```

Exemple 6-5 : Traduction d'une classe applicative MOSAÏC en deux classes O2.

Il est intéressant de remarquer l'analogie qu'il existe entre l'approche <langage de programmation> et l'approche <base de données>. Dans la première, on manipule essentiellement des données temporaires et il faut, en fin de programme, sauvegarder les données persistantes. Dans la seconde, on manipule essentiellement des données persistantes et il faut, en fin de programme, effacer les données temporaires.

On notera maintenant que certains systèmes comme ODE [BGJR92] offrent la possibilité de distinguer dans une même classe des attributs temporaires et des attributs persistants. Avec un tel système, la traduction d'une classe applicative MOSAÏC est immédiate.

Par ailleurs, on notera qu'au sein d'une classe applicative MOSAÏC, les méthodes peuvent accéder librement aux attributs, aux variables et aux autres méthodes de cette classe. Pour maintenir cette liberté, les deux classes O_2 représentant une classe applicative ne doivent rien «se cacher» : tous les attributs et toutes les méthodes doivent être publiques l'une pour l'autre. Comme O_2 n'offre pas de mécanisme d'exportation sélective, les autres classes O_2 peuvent également accéder aux caractéristiques privées de ces deux classes. Il est donc nécessaire d'effectuer un contrôle du respect du masquage avant la programmation, au moment de la conception.

2.6 Programmation des classes interactives

O_2 offre un ensemble de classes spécialisées et combinables qui permettent de proposer un dialogue riche et souple à l'utilisateur. Nous programmons chaque classe interactive MOSAÏC à l'aide de deux classes O_2 : la première représente la classe interactive elle-même (la partie contrôle) et la seconde représente la partie présentation. La classe O_2 chargée de gérer la présentation prend en charge seulement l'affichage et elle est élaborée par composition ou spécialisation d'une ou plusieurs classes spécialisées O_2. Cette classe présentation communique seulement avec sa classe contrôle associée et avec l'utilisateur. La classe O_2 chargée du contrôle peut dialoguer avec sa classe présentation, avec les autres «classes contrôles» et avec les classes applicatives. Cette classe contrôle n'effectue aucune entrée/sortie directe avec l'utilisateur de l'application.

Les classes spécialisées offertes par O_2 permettent de construire des objets complexes pour effectuer des échanges de données riches entre le programme et l'utilisateur. La représentation des actions déclenchables par l'utilisateur, sous forme de menus (déroulants) ou de boutons actifs, est délicate car il n'existe pas de classe prédéfinie de nature «menu» ou «bouton actif». Pour afficher des boutons actifs, nous avons utilisé deux techniques.

La première solution consiste à définir les actions attachées à un objet graphique (bouton actif ou item d'un menu) comme des méthodes de cet objet. Les méthodes de l'objet peuvent ensuite être présentées à l'utilisateur sous la forme d'un menu déroulant prédéfini par O_2 et attaché à l'objet affiché. Cette solution permet de contrôler finement les actions déclenchables par l'utilisateur à l'aide de différentes instructions qui limitent les méthodes apparaissant dans le menu ou qui interdisent ou autorisent leur déclenchement par l'utilisateur. Par contre, la présentation sous forme de menu ne respecte pas toujours les spécifications externes et elle n'est pas toujours intuitive ni pratique pour l'utilisateur. Par ailleurs certaines classes spécialisées O_2 (la classe `Dialog_Box` par exemple) ne permettent pas un tel affichage des méthodes.

La seconde technique consiste à définir pour chaque fenêtre, un classe pour présenter les données et une ou plusieurs classes pour présenter les actions activables dans cette fenêtre. Dans ce cas, les actions sont aussi présentées dans le menu déroulant prédéfini et les présentations graphiques de ces différentes classes sont liées pour constituer un objet graphique unique. Cette solution permet de se rapprocher des

spécifications externes et, dans certains cas, c'est la seule solution applicable. Cependant, elle entraîne une surcharge de travail pour le programmeur en augmentant le nombre de classes et l'écran obtenu n'est pas toujours agréable pour l'utilisateur.

En se limitant aux classes offertes par O_2, il n'existe donc pas actuellement de solution permettant de respecter au plus près les spécifications externes. La possibilité de remplacer la présentation prédéfinie des méthodes d'un objet sous forme de menu déroulant par une définition individuelle de leur représentation graphique et de leur localisation dans la fenêtre graphique constituerait une facilité importante.

2.7 Test de la classe d'un objet

Dans la section 8.6 du chapitre 5, nous avons souligné la nécessité, pour un objet interactif, de pouvoir tester le type d'un objet applicatif pour lui appliquer un traitement spécifique. Nous avons rencontré cette nécessité une seconde fois avec les classes O_2 spécialisées dans l'interaction. Par exemple la classe «`Dialog_Box`» permet de construire des fenêtres complexes à partir des classes spécialisées «`Label`, `Single_selection`, `Button_box`, `Radio_box`, etc.» sous-classes de la classe «`Component`» et qui comportent des méthodes et des attributs spécifiques (cf. exemple 6-6).

```
class Dialog_box
  type tuple (
     ed_name : string,
     structure : list(list(Component)) ) (*liste de composants de la "boîte"*)
  method init(a_ed_name : string, a_structure : list(list(Component)),
  method element(i : integer, j : integer) : Component,...
end

class Component
  type tuple (    ed_name : string )
end

class Button
  inherit Component
  type tuple (    label : string )
end

class Radio_Box
  inherit Component
  type tuple (    title : string,
                  buttons : list(Button),
                  selected : integer )
end

class Multiple_selection
  inherit Component
  type tuple (    title : string,
                  items : list(string),
                  selected : list(integer) )
end
```

Exemple 6-6 : Classes spécialisées O_2 permettant de construire des fenêtres complexes.

On crée un objet de la classe `Dialog_box` en initialisant sa `structure` avec des objets des sous-classes de la classe `Component`. Lorsque l'objet a été affiché et éventuellement modifié par l'utilisateur, on souhaite généralement récupérer les données actualisées. Pour cela, chaque classe «`Label, Single_selection, Button_box, Radio_box`, etc.» offre des attributs ou des méthodes spécifiques qu'il faut appeler pour chaque élément de la `structure`. Lorsqu'on accède aux éléments de la `structure` ils sont perçus comme des objets de la classe `Component` et on ne peut donc pas utiliser leurs caractéristiques spécifiques. Il faut donc soit maintenir un accès redondant en définissant un attribut permettant d'accéder directement à chaque élément de la `structure`, soit être capable de déterminer la classe de chaque objet de la `structure` et de lui envoyer les messages spécifiques en fonction de la classe à laquelle il appartient.

Comme le montre la figure 6-7, O_2 permet par une combinaison d'instructions, de tester le type d'un objet et de lui envoyer des messages spécifiques. Dans la boucle `for`, l'objet désigné par la variable `i` est perçu comme étant de classe `Inscription` et l'instruction `class_of` permet de déterminer la classe précise de l'objet `i` (`Inscription_Mono` ou `Inscription_Equipe`). Pour utiliser les caractéristiques spécifiques de `i` selon sa classe (les attributs `coureur` ou `coureurs`), il faut affecter l'objet désigné par `i` à une variable dont la classe correspond à celle de l'objet par les instructions : «`insc_mono = (o2 Inscription_Mono) i`» ou «`insc_équipe = (o2 Inscription_Equipe) i`» selon le cas.

```
class Inscription
  type tuple (    numero : integer,...)...
end

class Inscription_Mono
  inherit Inscription
  type tuple (    coureur : Licencié)...
end

class Inscription_Equipe
  inherit Inscription
  type tuple (    coureurs : set(Licencié))...
end

class Compétition
  type tuple (    classement_général : list(Inscription))...
end

named object Compétition_courante : Compétition

class Affichage_classement_général
  type...
  method affiche {
     O2 Inscription i;
     O2 Inscription_Mono insc_mono;
     O2 Inscription_Equipe insc_équipe;
     ...
     for (i in Competition_courante->classement_général)
     {  if ( i->class_of == insc_mono ->class_of )
        {     insc_mono = (o2 Inscription_Mono) i ;
              (*afficher le licencié insc_mono->coureur*)
        } else
```

```
        {     if ( i->class_of == insc_équipe->class_of ) {
              {  insc_équipe = (o2 Inscription_Equipe) i ;
                 (*afficher l'ensemble de licenciés insc_équipe->coureurs*)
              }
        }
    };...
  end
end
```

Exemple 6-7 : Test du type d'un objet et appel de ses méthodes spécifiques.

2.8 Remarques

Comme nous venons de le voir, la traduction d'une conception MOSAÏC en O_2 est aisée à mettre en œuvre et elle peut être largement automatisée.

La traduction d'une classe applicative nécessite une partition de ses caractéristiques selon leur caractère persistant ou temporaire et cette partition peut être réalisée automatiquement à partir des informations contenues dans le schéma de conception. La traduction initiale des classes interactives et plus directe mais les facettes présentations nécessitent un travail de programmation important pour prendre en compte les spécificités d'O_2.

Dans la dernière version d'O_2, le nouveau mécanisme de transaction permet de programmer des transactions très fines et les règles méthodologiques MOSAÏC garantissent l'absence de transactions imbriquées.

3 Intégration MOSAÏC - Aristote

3.1 Projet Aristote

Aristote est un projet de recherche commun au Laboratoire de Génie Informatique de l'institut IMAG et au Centre de Recherche BULL de Grenoble. Ce projet, lancé en 1988, a pour objectif d'élaborer un environnement de Conception et de Développement d'Applications Bases de Données en s'appuyant sur les techniques à objets. Le projet s'articule autour de trois centres d'intérêts :

- la définition d'un modèle de base destiné à constituer la source d'un générateur d'applications multicibles ; ce modèle est actuellement intégré dans le langage PEPLOM de programmation de bases de données,
- l'élaboration d'un poste de travail comportant à la fois des outils d'aide à la programmation d'applications et des outils d'aide à la conception,
- la validation du modèle, du langage et des outils sur des applications concrètes, notamment des applications médicales ou de génie logiciel.

Le modèle de base Aristote a largement évolué au cours du projet [ABD*90, DD91, Dech92] et nous nous appuyons sur la version actuelle du langage PEPLOM [Dech93] que nous présentons brièvement.

3.2 Langage PEPLOM

PEPLOM (PErsistent Programming Language for Object Management) [Dech93] est un langage de programmation de base de données dont l'objectif est d'intégrer de façon homogène les fonctionnalités des SGBD dans un langage de programmation développé à partir de C++. Nous présentons brièvement les éléments de PEPLOM utiles pour la traduction d'un schéma de conception MOSAÏC.

Dans PEPLOM, les valeurs sont distinguées des objets. Une **valeur** est une donnée non partageable appartenant à un **type de base** ou à un **type concret** élaboré à l'aide de constructeurs (nuplet, liste, ensemble, tableau ou union). Un **objet** encapsule des données et des méthodes, il est partageable et il est obtenu par instanciation d'un type abstrait.

La définition d'un **type abstrait** est composée de quatre parties :

- L'**héritage** introduit une relation de sous-typage entre les types abstraits. Cet héritage n'autorise pas la redéfinition des attributs et respecte les règles de contravariance pour les méthodes (les résultats peuvent être spécialisés alors que les arguments peuvent être généralisés).
- La **structure** d'un type abstrait décrit le type concret de la valeur (les données) encapsulée dans chaque objet de ce type abstrait.
- Les **fonctions** sont des méthodes de consultation dont l'activation n'entraîne pas d'effet de bord sur la valeur de l'objet ni sur celle des autres objets.
- Les **procédures** sont les méthodes qui effectuent des modifications sur la valeur de l'objet ou dont l'activation entraîne des modifications dans d'autres objets.

PEPLOM propose trois catégories de liens structurels entre objets : la **référence** vers un objet partageable, la **propriété** vers un objet non partageable et le lien **symétrique** qui représente une relation binaire. Ces liens sont portés par les attributs et le système maintient automatiquement les deux accès réciproques d'un lien symétrique.

Nous n'approfondissons pas le langage de programmation proprement dit dont une des particularités est d'intégrer des accès ensemblistes et déclaratifs dans des structures plus classiques de C++.

```
typedef abstract Inscription {
    struct {  int numéro_inscription;
              int dossard;
              Licence coureur;     (*Licence est un autre type abstrait*)
              struct {  ushort temps;
                        ushort points;
                        } performance
```

```
        } value
    functions {     ushort total;...}
    procedures {    affecter_dossard (int dos);...}
};
```

Exemple 6-8 : Type abstrait en PEPLOM.

En complément des types abstraits, PEPLOM offre un mécanisme de structuration supplémentaire : le module. Un **module** encapsule des objets et des valeurs persistants accessibles par des variables persistantes, des objets et des valeurs temporaires accessibles par des variables temporaires et des traitements distingués en fonctions et procédures. Cette notion est à rapprocher de la notion de module des langages modulaires.

Un module peut **exporter** des variables, des procédures et des fonctions. Un module peut **importer** d'autres modules et utiliser les caractéristiques qu'ils exportent. Cette importation définit la visibilité et limite les relations de dépendances entre modules.

```
moddef Gestion_compétitions {
    uses Gestion_données_administratives_fédérales;

    ref Compétition persist liste_compétitions <>;
    ref Compétition compétition_courante;
    ref Licence persist liste_licenciés <>;
    ref Club persist liste_clubs <>;

    functions {struct { int dossard;
                        string(30) nom;
                        string(20) prénom;
                        } liste_des_dossards <>;...
    }
    procedures {    effectuer_les inscriptions;
                    affecter_les_dossards;
                    gérer_la_compétition;...}
};
```

Exemple 6-9 : Module en PEPLOM.

Dans PEPLOM, la persistance est définie indépendamment du typage : tout objet peut persister quelque soit son type. La persistance est propagée par les attributs des objets persistants à partir des racines de persistance. Les racines de persistance sont d'une part les variables persistantes des modules et d'autre part les objets rendus explicitement persistants à leur création ou pendant leur utilisation. On peut accéder directement à l'ensemble des objets persistants ainsi qu'à l'ensemble des objets temporaires de chaque type abstrait.

La notion de transaction n'est pas actuellement prise en compte dans PEPLOM et des recherches se poursuivent sur ce point.

Nous présentons maintenant une traduction possible d'une conception MOSAÏC en un programme PEPLOM.

3.3 Représentation des classes projets en PEPLOM

On peut envisager plusieurs solutions pour représenter une classe projet MOSAÏC en PEPLOM : plusieurs modules pour une classe projet ou un module par classe projet (cf figure 6-2).

3.3.1 Plusieurs modules par classe projet

Pour représenter une classe projet à l'aide de plusieurs modules PEPLOM, il est nécessaire de la décomposer et on peut envisager pour ce faire différents critères de décomposition (cf. solutions 1, 2 et 3 figure 6-2).

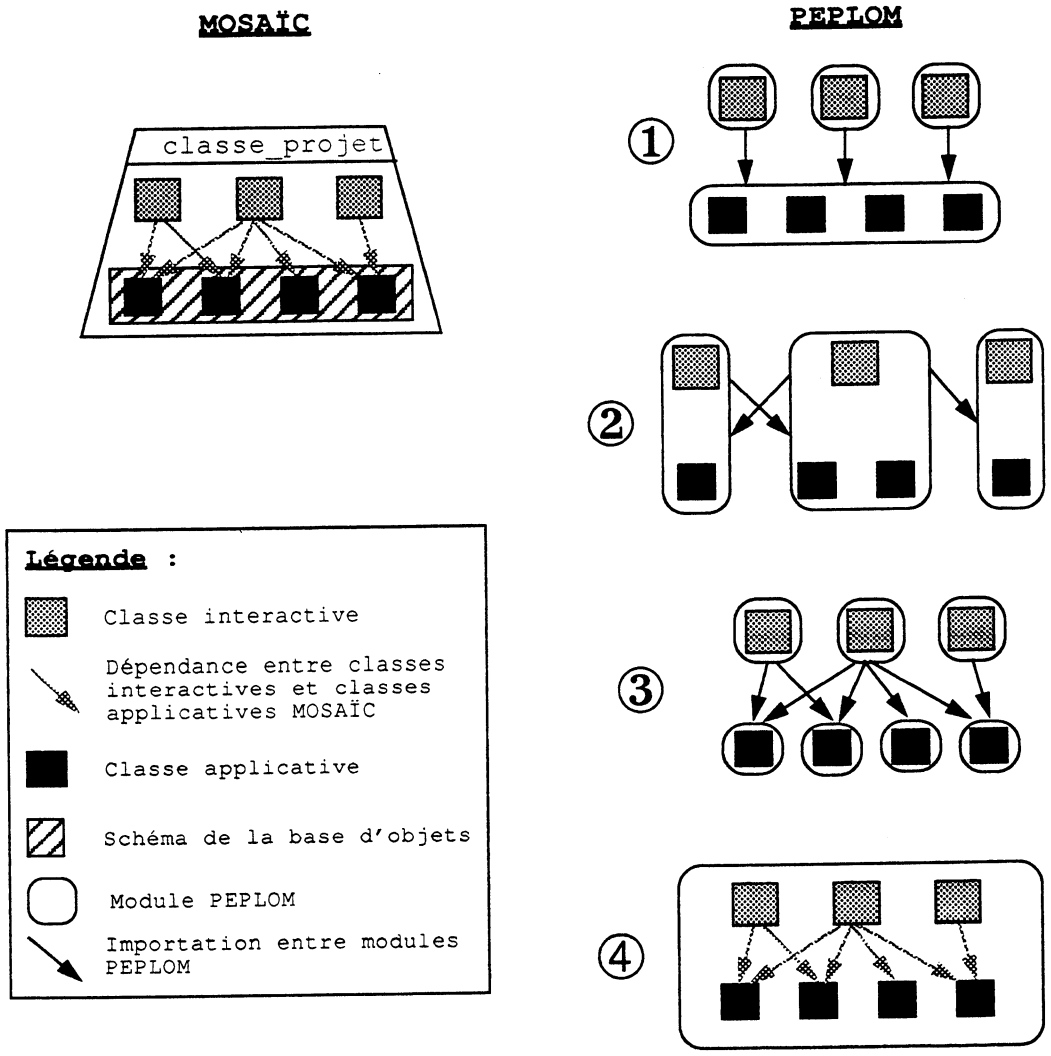


Figure 6-2 : Traductions d'une classe projet MOSAÏC en modules PEPLOM.

On peut distinguer un module chargé de gérer la base d'objets applicatifs et associer, à chaque application, un module pour gérer son interface interactive (cf. schéma ①

figure 6-2). Cette solution se rapproche de l'architecture habituelle qui distingue la base de données et les applications mais elle complique largement les communications entre l'interface interactive d'une application et son corps en interdisant tout accès direct des objets interactifs vers les objets applicatifs. Par ailleurs cette solution impose de transformer les méthodes primaires appelées par les classes interactives sous forme de procédures exportées par le module chargé de gérer les objets persistants.

- Le schéma de la base d'objets persistants (SBO) peut être décomposé et géré par différents modules sans perte d'informations seulement si le SBO, considéré comme un graphe non orienté, n'est pas connexe. Dans le cas contraire, toute décomposition entraîne la rupture d'un arc et donc la perte de relations structurelles entre des classes applicatives. Si le SBO n'est pas connexe, chaque sous-graphe connexe peut être géré par un module particulier et on peut envisager d'encapsuler, dans un même module PEPLOM, certaines interfaces interactives avec des sous-graphes du SBO (cf. schéma ② figure 6-2) ou de les séparer complètement (cf. schéma ③ figure 6-2)

Les gains obtenus en terme de modularité avec ces différentes solutions ne nous paraîssent pas suffisants en regard des contraintes imposées pour les communications entre les objets interactifs et les objets applicatifs.

3.3.2 Un module par classe projet

Compte tenu de ces différentes remarques, nous préconisons de représenter une classe projet MOSAÏC par un module PEPLOM unique (cf. schéma ④ figure 6-2 et exemple 6-10). La correspondance entre ces deux concepts est alors la suivante : les racines de persistance MOSAÏC sont représentées par des variables persistantes en PEPLOM alors que les racines d'application sont traduites sous forme de procédures.

```
<--MOSAÏC-

 classe projet Nom_projet

   racine de persistance
        nom_racine_persistante1 : réf Nom_classe_applicative1
        nom_racine_persistante2 : prop Nom_classe_applicative2

   racine d'application
        nom_application : Nom_classe_initiale

 fin_classe_projet


-PEPLOM-->

 moddef Nom_projet {

     ref Nom_classe_applicative1 persist nom_racine_persistante1;
     own Nom_classe_applicative2 persist nom_racine_persistante2;
```

```
    procedures {    nom_application;
                    }
};
```

Exemple 6-10 : Représentation d'une classe projet MOSAÏC à l'aide d'un module PEPLOM unique.

Dans la suite nous adoptons cette dernière solution : un module par classe projet.

3.3.3 Représentation des classes en PEPLOM

De façon générale, une classe MOSAÏC est représentée à l'aide d'un type abstrait PEPLOM.

Les attributs sont représentés dans la structure du type abstrait avec quelques différences de notations notamment pour les constructeurs (ensemble = {}, liste = <>, nuplet = struct, etc.). On notera que PEPLOM offre deux des trois relations MOSAÏC : la référence et la propriété. La dépendance existentielle avec partage doit être simulée ; pour cela le langage offre une facilité : les liens symétriques.

La partie fonctionnelle d'une classe MOSAÏC est très riche. Les méthodes sont représentées sous forme de fonctions et de procédures. Les informations concernant les messages émis et les dépendances engendrées sont utiles pendant la conception mais ne sont pas représentées explicitement en PEPLOM.

Les états de la partie dynamique peuvent éventuellement être représentés sous forme de fonctions booléennes en PEPLOM mais le langage n'offre pas actuellement de facilité particulière pour définir la cohérence des objets.

```
<--MOSAÏC-

 classe Nom_classe
   exporte    fonction nom_attribut1, nom_attribut3
              méthode nom_méthode1, nom_méthode2

   attribut
      nom_attribut1 : réf Nom_classe1
      nom_attribut2 : prop Nom_classe2
      nom_attribut3 : Ensemble(prop Nom_classe3)
      nom_attribut4 : Liste(réf Nom_classe4)

      nom_attribut5 : Nuplet(   nom_attribut6 : prop Nom_classe6;
                                nom_attribut7 : réf Nom_classe7)

   état nom_état : proposition_logique

   méthode nom_méthode1 : Nom_classe8;
   méthode nom_méthode2;
 fin


-PEPLOM-->
```

```
typedef abstract Nom_classe {
    struct {   own Nom_classe2 nom_attribut2;
               ref Nom_classe4 nom_attribut4<>;

               struct {   own Nom_classe6 nom_attribut6;
                          ref Nom_classe7 nom_attribut7;
                          } nom_attribut5;
               public :
               ref Nom_classe1 nom_attribut1;
               own Nom_classe3 nom_attribut3{};
               } value

    functions {  nom_état : boolean }

    procedures { nom_méthode1 : Nom_classe8;
                 nom_méthode2 }
};
```

Exemple 6-11 : Traduction générale d'une classe MOSAÏC en un type abstrait PEPLOM.

3.4 Représentation des classes applicatives

Une classe applicative MOSAÏC comportant seulement des caractéristiques persistantes ou seulement des caractéristiques temporaires est traduite directement sous forme d'un type abstrait PEPLOM unique (cf. exemple 6-11).

Les classes facettes d'une classe applicative peuvent être représentées à l'aide de types abstraits et d'attributs représentant des liens symétriques.

En PEPLOM comme en O_2, la persistance d'un objet entraîne le persistance de toutes ses données et la persistance se propage par les attributs. Lorsqu'une classe applicative MOSAÏC possède à la fois des caractéristiques persistantes (attributs ou méthodes persistantes) et des caractéristiques temporaires (variables ou méthodes temporaires), elle doit être représentée à l'aide de deux types abstraits PEPLOM selon une technique analogue à celle utilisée en O_2 (cf. exemple 6-12). Le premier type abstrait regroupe les caractéristiques persistantes de la classe applicative et le second ses caractéristiques temporaires. Chaque objet applicatif en partie temporaire et en partie persistant est donc représenté par deux objets PEPLOM.

En fin d'exécution, les parties temporaires des objets applicatifs partiellement temporaires doivent être supprimées. Le mécanisme utilisé pour réaliser cette opération est le même qu'en O_2 : une variable appelée `Objets_partiellement_temporaires` permet d'accéder à tous les objets applicatifs partiellement temporaires. Pour cela on défini un type abstrait générique `O_partiellement_temporaire` qui offre les méthodes minimales partagées par tous les objets partiellement temporaires : `promotion` et `supprimer_partie_temporaire`. L'opération de promotion a pour effet de créer la partie temporaire d'un objet applicatif et de rattacher cet objet à la variable `Objets_partiellement_temporaires`. L'opération de suppression a pour effet de supprimer la partie temporaire. En fin d'exécution d'une application, la méthode `supprimer_partie_temporaire` est appelée sur tous les objets accessibles par la variable `Objets_-`

partiellement_temporaires. Ainsi seules les données persistantes des objets applicatifs persistants sont conservées.

```
-PEPLOM-->

moddef nom_projet {

  ref O_partiellement_temporaire Objets_partiellement_temporaires {};

  procedures {
     nom_application;
        (*une procédure est associée à chaque application MOSAÏC*)

     début_application
        (*procédure appelée en début d'application, elle initialise*)
        (*à «ensemble vide» la variable Objets_partiellement_temporaires*)

     fin_application
        (*procédure appelée en fin d'application, elle applique la méthode*)
        (*supprimer_partie_temporaire à tous les objets accessibles*)
        (*par la variable Objets_partiellement_temporaires*)
  }
};


typedef abstract O_partiellement_temporaire {

  struct { }    (*un attribut «partie_temporaire» doit être défini*)

  function { boolean promu }; (*teste si la «partie_temporaire» existe*)

  procedures {
     promotion;                        (*crée la «partie_temporaire»*)
     supprimer_partie_temporaire       (*supprime la «partie_temporaire»*)
  };
};


typedef abstract Partie_Temporaire
     struct { }    (*un attribut «partie_temporaire» doit*)
                   (*être ajouté dans les sous-classes*)
};
```

Exemple 6-12 : Gestion des objets applicatifs partiellement temporaires en PEPLOM.

Une classe applicative MOSAÏC comportant à la fois des caractéristiques persistantes et des caractéristiques temporaires est alors transformée en deux types abstraits héritant des types abstraits O_partiellement_temporaire et Partie_Temporaire selon la technique présentée dans l'exemple 6-13.

```
<--MOSAÏC-

classe applicative Nom_classe_appli
  attribut nom_attribut1 : réf Nom_classe1          (*persistant*)
  variable nom_variable2 : prop Nom_classe2         (*temporaire*)
```

```
  méthode nom_méthode1 persistante
  méthode nom_méthode2 temporaire
fin


-PEPLOM-->

typedef abstract Nom_classe_appli : O_partiellement_temporaire {
  struct {  ref Nom_classe_appli_partie_temporaire partie_temporaire
                              inv value.partie_persistante;
            ref Nom_classe1 nom_attribut1;
          } value

  function { boolean promu }; (*promu = (partie_temporaire==nil)*)

  procedures {
     nom_méthode1;
     promotion; (*partie_temporaire=new(Nom_classe_appli_partie_temporaire)*)
     supprimer_partie_temporaire (*partie_temporaire=nil*)
     }
};


typedef abstract Nom_classe_appli_partie_temporaire : Partie_Temporaire {
  struct {  ref Nom_classe_appli partie_persistante
                              inv value.partie_temporaire;
            own Nom_classe2 nom_attribut2;
          } value
  procedures { nom_méthode2 }
};
```

Exemple 6-13 : Traduction d'une classe applicative MOSAÏC en PEPLOM.

En PEPLOM, le symbole «:» désigne une relation d'héritage. On notera qu'en O_2, les deux attributs «`partie_temporaire`» et «`partie_persistante`» peuvent être définis dans les classes génériques `O_partiellement_temporaire` et `Partie_Temporaire`, car O_2 suit les règles de covariance et autorise la spécialisation des attributs. En PEPLOM ce n'est pas possible car les règles d'héritage ne permettent pas de spécialiser les attributs.

Comme pour le système O_2 (cf. 2.5) les deux types abstaits PELOM doivent être totalement publics. Aucune caractéristique ne doit être privée pour permettre notamment aux méthodes temporaires d'accéder librement aux caractéristiques persistantes.

3.5 Propositions d'extensions pour le langage PEPLOM

Notre brève expérience nous conduit à proposer différentes extensions à PEPLOM pour faciliter la programmation d'applications.

Nous n'avons pas pu aborder la programmation de l'interface interactive en PEPLOM car ce langage n'offre pas actuellement de types spécialisés pour gérer le dialo-

gue. Une première extension consiste donc à compléter les types de base en s'inspirant éventuellement du SGBDOO O_2, sans omettre les classes spécialisées permettant de définir des actions déclenchables par l'utilisateur. Ces actions peuvent physiquement être représentées par des méthodes sur des objets et présentées à l'utilisateur comme des items dans des «menus» ou comme des «boutons actifs».

Nous avons souligné la nécessité de pouvoir connaître le type des objets manipulés pour pouvoir éventuellement leur appliquer des traitements différents et nous avons indiqué comment réaliser une telle opération en O_2. Pour conserver un contrôle statique du typage et offrir la possibilité d'adapter un traitement au type d'un objet, on peut définir une nouvelle instruction conditionnelle (cf. exemple 6-14).

```
typedef abstract Type0...

typedef abstract Type1 : Type0...        (*sous-type de type0*)

typedef abstract Type2 : Type0...        (*sous-type de type0*)

...
ref Type0 nom_var;          (*l'objet désigné par nom_var peut être*)
                            (*de type : Type0, Type1 ou Type2*)
...
case_type_of nom_var {
  Type1:{          (*cette branche est compilée en considérant nom_var comme*)
                   (*de type Type1 et elle est parcourue si l'objet*)
         };        (*désigné par nom_var est effectivement de type Type1*)

  Type2:{          (*cette branche est compilée en considérant nom_var comme*)
                   (*de type Type2 et elle est parcourue si l'objet*)
         };        (*désigné par nom_var est effectivement de type Type2*)

  otherwise {      (*cette branche est compilée en considérant nom_var comme*)
                   (*de type Type0 et elle est parcourue si le type de*)
            };     (*l'objet désigné par nom_var n'est pas Type1 ni Type2*)

}; (*end_case_type*)
```

Exemple 6-14 : Instruction conditionnelle testant le type d'un objet.

Une telle facilité est nécessaire d'une part pour les communications entre le corps et l'interface interactive et d'autre part pour les communications entre les classes interactives et les classes de base du logiciel cible pour gérer les présentations. Pour respecter le principe d'autonomie et de localité, l'utilisation de cette instruction doit être limitée à ces deux cas.

En PEPLOM comme en O_2, la programmation des classes applicatives ayant une partie temporaire est complexe car, dans ces deux systèmes, toutes les données d'un objet persistant persistent et propagent la persistance. La programmation des classes applicatives serait largement facilité si PEPLOM permettait de définir des attributs non

persistants, comme c'est le cas dans certains systèmes [BGJR92], ou au moins un attribut particulier ne propageant pas la persistance.

Aucun mécanisme de transaction n'a encore été fixé dans PEPLOM. Cependant, un mécanisme défini à partir des messages envoyés par les classes interactives vers les classes applicatives peut constituer un bon compromis entre la complexité de sa mise en œuvre et la finesse des transactions ainsi définies.

Enfin, comme nous l'avons remarqué dans le cas du système O_2 (cf. 2.5), un mécanisme d'exportation sélective comme il en existe en Eiffel ou en C++, permettrait de programmer une classe applicative à l'aide de deux types abstraits PEPLOM tout en assurant le respect du masquage spécifié dans le schéma de conception MOSAÏC.

3.6 Remarques

Les règles de traduction d'un schéma de conception MOSAÏC vers un programme PEPLOM sont très voisines de celles utilisées avec O_2. Cependant, PEPLOM est un prototype de recherche moins complet qu'O_2 notamment pour la gestion du dialogue homme-machine et pour les mécanismes de transaction. Par ailleurs, des travaux supplémentaires sont nécessaires pour exploiter plus efficacement la notion de module PEPLOM.

4 Eléments d'une démarche

4.1 MOSAÏC dans le cycle d'abstraction de MERISE

En génie logiciel, on distingue généralement trois niveaux de préoccupation : l'expression des besoins (la spécification), la conception et la réalisation. MOSAÏC s'intéresse seulement à l'étape de conception. En base de données, le processus de développement est également organisé selon trois niveaux de préoccupations : le niveau conceptuel, le niveau interne décomposé en niveau logique et niveau physique et le niveau externe. MOSAÏC intervient au niveau interne et plus particulièrement au niveau logique.

La méthode MERISE [TRC85] s'appuie sur un cycle d'abstraction composé de trois niveaux principaux : conceptuel, organisationnel et opérationnel. Dans chacun de ces niveaux, on distingue les données et les traitements pour obtenir six modèles (cf. chapitre 2 section 3.3.2) : le modèle conceptuel des données (MCD), le modèle conceptuel des traitements (MCT), le modèle logique des données (MLD), le modèle organisationnel des traitements (MOT), le modèle physique des données (MPD), le modèle opérationnel des traitements (M Op. T).

Comme nous l'avons remarqué dans le chapitre 2, la méthode MERISE classique n'est pas adaptée pour développer une application à l'aide d'un langage de programmation de bases d'objets. En effet, la solution technique proposée par MERISE sépare strictement les données persistantes et les programmes ce qui ne permet pas de tirer parti des spécificités des LPBO : encapsulation des données et des traitement, cohabitation d'objets temporaires et d'objets persistants. MOSAÏC a été développée pour assister le concepteur dans la définition d'une architecture logicielle adaptée aux LPBO, mais elle ne prend pas en compte la spécification de l'application.

MOSAÏC et MERISE peuvent être combinées pour une adaptation aux facilités offertes par les LPBO sans remettre en cause complètement l'investissement des entreprises dans la méthode MERISE. Les trois modèles de spécification (MCD, MCT et MOT) sont conservés pour établir les spécifications du système mais ils doivent être complétés par les spécifications externes qui décrivent le comportement apparent du système pour l'utilisateur. MOSAÏC remplace ensuite le modèle logique des données (MLD) et le niveau opérationnel (MPD et MOp.T) comme le montre la figure 6-3.

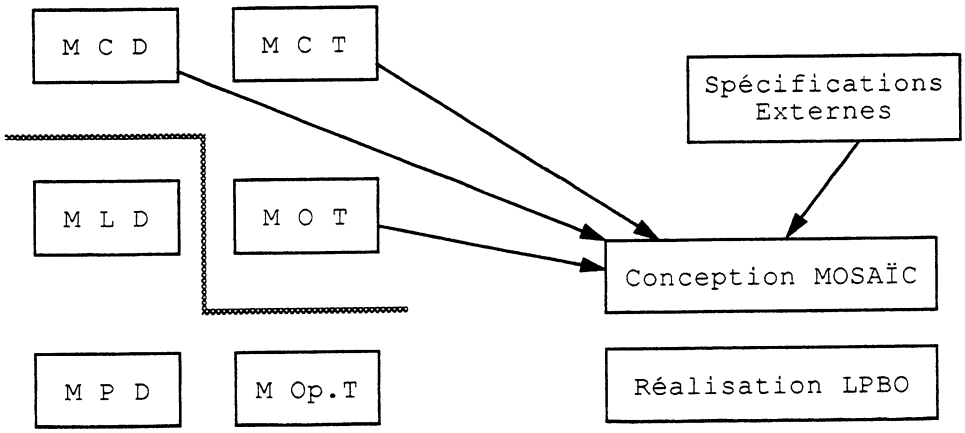


Figure 6-3 : Adaptation du cycle d'abstraction MERISE pour MOSAÏC.

La dichotomie données/traitements fondamentale dans MERISE ne facilite pas particulièrement la mise en évidence des classes. Cependant, une première ébauche des classes applicatives et du schéma de la base d'objet MOSAÏC peut être élaborée à partir du modèle conceptuel des données. Cette ébauche doit être ensuite complétée pour prendre en compte les objets ayant un rôle seulement fonctionnel et qui n'apparaissent pas dans le MCD MERISE. Cette catégorie d'objets a un rôle capital dans une approche de conception dirigée par les objets.

Il faut noter que depuis 1990, la méthode MERISE subit de nombreuses évolutions et s'adapte aux apports des techniques à objets [Roch91].

4.2 Etapes d'une conception MOSAÏC

4.2.1 Généralités

La définition d'une démarche de conception adaptée à MOSAÏC est délicate car la conception est une activité créatrice qui fait largement appel à l'expérience du concepteur et qui peut difficilement être menée selon un processus rigide organisé en étapes dont l'enchaînement et le contenu sont strictement fixés par avance. Un guide souple constitué de différentes étapes peut être très utile pour le concepteur novice. Dans une approche à objets, la conception est progressive et hautement itérative et les concepts fondamentaux des techniques à objets incitent au développement rapide de prototypes qui grâce aux facilités offertes par l'héritage, sont facilement réutilisables.

La démarche de conception que nous présentons suppose que le concepteur dispose des spécifications du système à concevoir, sous forme textuelle, graphique ou formelle. La séparation modulaire initiale entre le corps d'une application et son interface interactive permet de mener la conception et la validation de ces deux composants de façon largement parallèle. La démarche est décomposée en deux phases : la conception globale puis la conception détaillée, et en quatre étapes : la conception globale des classes applicatives et des classes interactives et la conception détaillée des classes applicatives et des classes interactives (cf. figure 6-4).

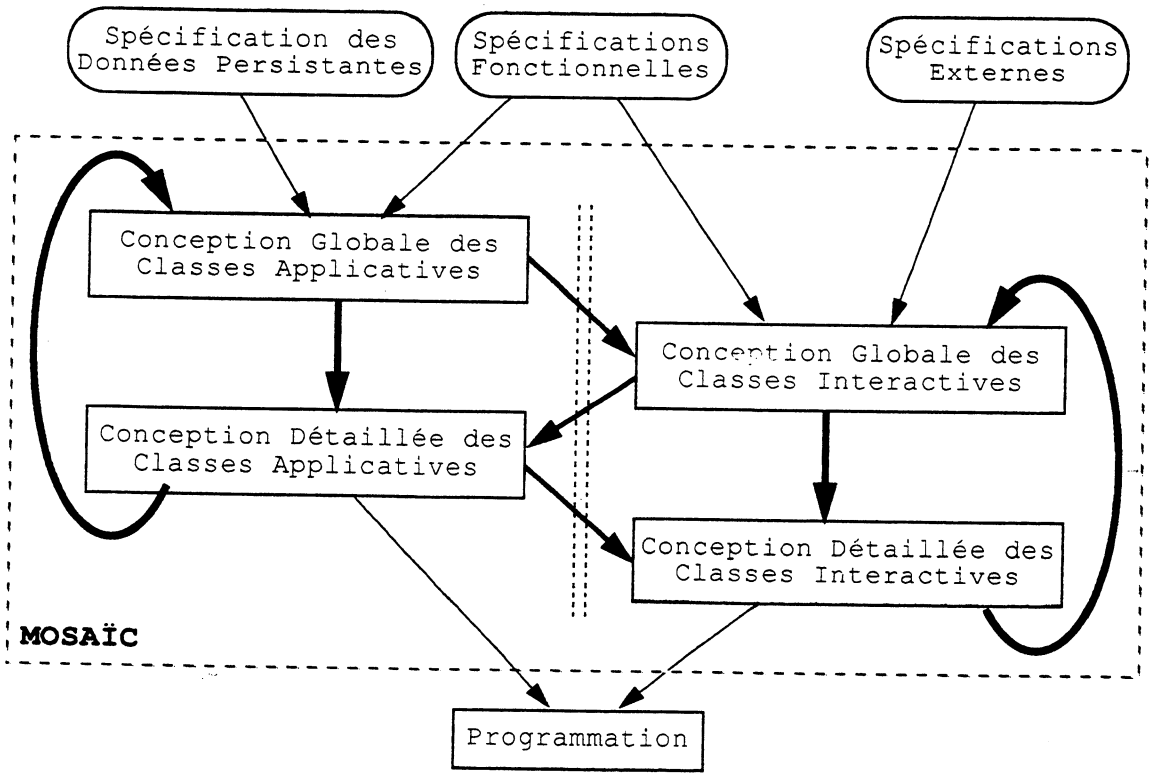


Figure 6-4 : Etapes d'une démarche de conception MOSAÏC.

Chaque étape comporte un ensemble de tâches à réaliser mais l'ordre dans lequel nous les présentons dans les sections suivantes n'indique pas obligatoirement un ordre chronologique strict de leur réalisation.

4.2.2 Conception globale des classes applicatives

L'objectif de la conception globale des classes applicatives est de fixer l'architecture du schéma de la base d'objets du projet ainsi que l'architecture du corps des applications. Pour cela, le concepteur doit mener à bien différentes tâches :

- Identifier les classes applicatives par un processus de modélisation du domaine de l'application sans oublier de modéliser le contexte dans lequel le système s'inscrit. Par exemple dans l'application «gestion d'une compétition sportive» il doit exister une classe `Compétition`, ou lors de la conception du système d'information d'une organisation, il doit exister une classe représentant cette organisation.
- Définir les propriétés et les relations structurelles des classes applicatives en étudiant précisément le partage éventuel des objets qui détermine la nature des relations structurelles. Certaines classes peuvent, à ce stade, ne pas avoir de propriétés ou de relations structurelles avec les autres classes, c'est notamment le cas des classes d'objets ayant seulement un rôle fonctionnel dans le système d'applications.
- Mettre en évidence les états remarquables des objets lorsque les objets ont des cycles de vie riches et qu'ils sont susceptibles d'évoluer entre leur création et leur destruction. Dans une classe, la mise en évidence de différents états peut conduire à définir plusieurs facettes si les cycles de vie des différentes données ne sont pas tous identiques.
- Attribuer les fonctions principales de l'application extraites des spécifications fonctionnelles en choisissant, pour chaque fonction, un objet responsable de sa réalisation. Les fonctions sont en général représentées par des méthodes primaires. Cette phase peut donner lieu à la création de nouvelles classes lorsque la responsabilité d'une fonction ne peut pas être attribuée à une classe applicative existante.
- Utiliser l'héritage pour factoriser les définitions et organiser les classes en distinguant la réutilisation représentée par la relation d'adaptation, qui ne permet pas la substitution, et le sous-typage, qui supporte le polymorphisme et qui peut être interprété comme une relation d'inclusion ensembliste.

Différentes heuristiques ont été proposées pour identifier les classes d'un programme réalisé avec un langage à objets [Ferb90, Meye88] ; nous ne les développons pas ici. L'étape de conception globale des classes applicatives précéde généralement les autres étapes. Cependant, elle peut concerner tout le système ou, dans la perspective d'un développement progressif, une partie seulement du système [Bézi86]. Cette étape peut être suivie par la conception globale des classes interactives ou par la conception dé-

taillée des classes applicatives.

4.2.3 Conception globale des classes interactives

Un processus similaire à la conception globale des classes applicatives est conduit pour l'interface interactive de chaque application. Pour cela, le concepteur doit auparavant fixer les racines d'applications et les classes initiales des applications. Ensuite, les tâches suivantes doivent être menées pour chaque application :

- Identifier les classes interactives par un processus de modélisation du dialogue à partir des spécifications externes. L'expérience du concepteur peut l'amener, dès cette tâche, à introduire des classes interactives nécessaires au bon fonctionnement de l'interface comme par exemple des classes pour gérer la cohérence des informations affichées, pour réaliser les opérations «défaire/refaire», «couper/copier/coller» ou pour gérer les erreurs.
- Définir les attributs et les méthodes des présentations des classes interactives représentant des objets interactifs présentés à l'utilisateur. Ces attributs représentent les informations présentées à l'utilisateur et ces méthodes correspondent aux actions déclenchables par l'utilisateur.
- Définir les relations structurelles entre les classes interactives (il s'agit souvent de relations hiérarchiques matérialisées par un lien de dépendance existentielle) et vers les classes applicatives (il s'agit généralement de références).
- Mettre en évidence les états remarquables des objets. Ils permettent tout d'abord de spécifier les évolutions de l'interface : le caractère activable ou non des actions, le caractère modifiable ou non des informations. Ils permettent ensuite de mettre en évidence la provenance des informations affichées lorsqu'elles correspondent à des données contenues dans des objets applicatifs. Cette étape peut conduire à l'ajout de nouvelles classes interactives pour gérer la cohérence des informations affichées.
- Utiliser l'héritage pour organiser les classes de la même façon que pour les classes applicatives.

Nous ne proposons pas d'heuristique pour identifier les classes interactives et les travaux sur ce point sont, à notre connaissance, assez peu nombreux. Notons cependant les travaux menés dans le cadre du modèle PAC [NC91, NC92].

4.2.4 Conception détaillée des classes applicatives

L'objectif de la conception détaillée des classes applicatives est de préciser le fonctionnement du système :

- Choisir les algorithmes; leur définition peut être réalisée de différentes façons (langue naturelle, langage algorithmique, langage d'un système cible, etc.) et peut nécessiter, en plus des variables locales de la méthode, l'introduction de variables de classes.

- Mettre en évidence les messages et les dépendances à partir des algorithmes retenus. Cette tâche peut être menée sans que les algorithmes soient totalement fixés.
- Créer les classes d'interrogation lorsqu'une méthode d'une classe applicative doit interroger l'utilisateur pour avoir un complément d'information via l'interface interactive. Dans ce cas, la signature de la méthode doit comporter un argument supplémentaire qui désigne l'objet interactif chargé de traiter la requête du corps de l'application vers l'utilisateur.
- Définir les nouvelles méthodes requises pour traiter les messages émis soit par les objets applicatifs, elles sont alors souvent secondaires, soit par les objets interactifs, elles doivent alors être primaires.
- Fixer les racines de persistance de la classe projet.
- Limiter les services utilisables en définissant les signatures des classes et contrôler que les règles de sous-typage sont vérifiées lorsque de telles relations sont définies. Généralement, les attributs sont publics et les variables sont privées.
- Généraliser les classes pour faciliter leur réutilisation dans les applications futures.

4.2.5 Conception détaillée des classes interactives

Pendant la conception détaillée des classes interactives, le concepteur fixe le fonctionnement de l'interface interactive et les relations entre l'interface interactive et le corps de l'application :

- Choisir les algorithmes comme pour les classes applicatives sans oublier la méthode `lancer_application` portée par la classe initiale et activée au lancement de l'application.
- Définir les méthodes requises pour réaliser les méthodes de la facette présentation. Ces méthodes peuvent utiliser les méthodes des autres classes interactives et les méthodes primaires des classes applicatives.
- Définir les classes interactives chargées de réaliser les méthodes des classes d'interrogation.
- Limiter les services utilisables par les signatures.
- Généraliser les classes interactives pour améliorer leur réutilisabilité.

5 OCAPI : un outil d'aide à la conception

5.1 Choix de l'outil

Les outils d'aide à la conception constituent le quatrième type de composant d'une méthode après les modèles, les langages et la démarche. Le rôle de ces outils est d'assister le concepteur dans sa tâche de conception. Dans MOSAÏC, l'absence de mécanisme de structuration autre que la classe et la richesse de cette notion rend indispensable le recours à un outil pour mener à bien une conception d'un système d'applications.

On peut distinguer différentes catégories d'outils. Les éditeurs, éventuellement graphiques ou syntaxiques, les outils d'archivage, de gestion de versions sont des outils de base habituels et indispensables. Les simulateurs sont utiles pour permettre aux futurs utilisateurs de tester l'ergonomie de l'application. En aval des éditeurs, les générateurs et les compilateurs génèrent, à partir d'un schéma de conception, des éléments de programmes plus ou moins complets pour un ou plusieurs systèmes cibles. En amont des outils de base, on trouve les outils que l'on peut réellement qualifier d'«aide à la conception» et qui sont capables de suggérer au concepteur des éléments de solution ou des améliorations.

L'outil OCAPI [Pudd92] développé est un outil de base limité aux fonctions d'édition et de génération. La première fonction est réalisée par un éditeur graphique à l'aide duquel le concepteur élabore un schéma de conception en respectant la syntaxe du langage graphique MOSAÏC. La génération est effectuée automatiquement et permet d'obtenir, à partir d'un schéma de conception, un squelette d'application PEPLOM.

Nous précisons tout d'abord les spécifications d'OCAPI avant de justifier la technique de réalisation et de présenter brièvement l'outil effectivement développé.

5.2 Spécifications de l'outil

5.2.1 Présentation générale

Dans l'outil OCAPI le mode graphique a été privilégié pour permettre une conception plus aisée et plus conviviale. Il s'appuie pour cela sur le langage graphique de MOSAÏC présenté dans le chapitre 5. L'outil permet au concepteur d'ouvrir différentes fenêtres constituant autant de «zones de travail» et de travailler successivement et librement dans ces fenêtres. OCAPI doit donc gérer plusieurs fils de dialogues et assurer la cohérence des différentes vues présentées au concepteur. Certains concepts n'ont pas de représentation graphique (les états par exemple) et les autres nécessitent souvent la saisie d'informations complémentaires textuelles. La saisie de ces informations complémentaires est réalisée dans des formulaires qui ne doivent pas être bloquant pour permettre au concepteur de suspendre temporairement la saisie du formulaire pour consulter des informations ou travailler dans une autre fenêtre.

Nous présentons brièvement les principaux éléments des spécifications externes de

l'outil OCAPI.

Toutes les classes (applicatives, interactives, projets, etc.) produites pendant une conception sont regroupées dans un dictionnaire. A tout moment, les classes du dictionnaire sont accessibles sous différentes formes : une liste de classes applicatives, une liste de classes interactives, etc. Pour faciliter l'utilisation du dictionnaire qui est souvent très volumineux, OCAPI offre un mécanisme de structuration des classes selon différentes vues. Chaque vue est présentée dans une fenêtre et on distingue les vues explicites et les vues calculées. Chaque vue peut être présentée selon différents points de vue en fonction du contexte dans lequel se place le concepteur.

5.2.2 Différentes vues

5.2.2.1 Vues explicites

Une vue explicite est créée et détruite explicitement par le concepteur et elle persiste entre deux sessions de travail. Une vue explicite possède un nom, donné par le concepteur, permettant de l'identifier dans l'ensemble des vues. Le contenu d'une vue explicite est élaboré manuellement par le concepteur qui indique explicitement les classes apparaissant dans la vue. Pour cela, le concepteur peut copier une classe dans une autre vue ou dans les listes des classes et la coller dans la vue courante. Il peut également afficher une classe liée par un attribut, un message ou une relation d'héritage à une classe présente dans la vue. Le concepteur peut créer de nouvelles classes, ces classes sont insérées dans la vue courante ainsi que dans le dictionnaire des classes.

5.2.2.2 Vues calculées

Une vue calculée est le résultat d'une requête, éventuellement paramétrée, appliquée au dictionnaire. Le concepteur peut travailler dans une vue calculée pendant une session de travail mais une vue calculée ne persiste pas entre deux sessions de travail. Les vues calculées prédéfinies sont les suivantes :

- La vue «graphe structurel» prend comme argument une classe, une racine de persistance ou une racine d'application et affiche les classes accessibles par des relations structurelles à partir de l'argument.
- La vue «graphe fonctionnel» prend comme argument une méthode (resp. une classe) et affiche le graphe fonctionnel de cette méthode (resp. l'ensemble des dépendances fonctionnelles : les classes utilisées directement ou indirectement par cette classe).
- La vue «graphe hiérarchique» prend comme argument une classe et affiche les sous-classes de l'argument.
- La vue «schéma de la base d'objets» affiche le schéma de la base d'objets de la classe projet courante (cf. chapitre 5 section 7.3).
- La vue «interface interactive» prend comme argument une racine d'application et affiche l'interface interactive établie à partir des dépendances fonctionnelles (cf.

chapitre 5 section 8.3).

- La vue «corps» prend comme argument une racine d'application et affiche le corps de l'application correspondant établi à partir des dépendances fonctionnelles (cf. chapitre 5 section 8.4).

5.2.2.3 Contextes

Le contenu d'une vue, explicite ou calculée, peut être affiché selon différents contextes. Dans chaque contexte, seules certaines caractéristiques des classes de la vue sont présentées à l'utilisateur. On distingue quatre contextes :

- le contexte structurel dans lequel seuls les attributs sont affichés,
- le contexte fonctionnel qui présente seulement les méthodes, les messages et les dépendances fonctionnelles,
- le contexte externe présentant les caractéristiques publiques,
- le contexte détaillé qui affiche toutes les caractéristiques des classes de la vue.

5.2.3 Différentes fenêtres et commandes associées

5.2.3.1 Fenêtre principale

Au cours d'une session de travail, le concepteur travaille dans une seule classe projet, mais il peut ouvrir différentes vues. Par soucis d'homogénéité, le concepteur dispose dans toutes les vues des mêmes commandes mais les autorisations de déclenchement de ces commandes dépendent du contexte. Par exemple, lorsque l'interface interactive d'une application est affichée dans une vue, l'utilisateur peut ajouter dans cette vue de nouvelles classes interactives mais il ne peut pas ajouter de classes applicatives. De même, lorsqu'une vue est présentée selon un contexte structurel, seuls des attributs peuvent être ajoutés. Lorsqu'une commande n'est pas utilisable, elle est présentée à l'utilisateur sous une forme grisée et elle n'est pas activable. La figure 6-5 présente la fenêtre principale d'OCAPI composée d'une zone de travail contenant les racines de persistance et les racines d'application de la classe projet courante.

Dans le menu **Fichier**, la commande **Contrôler** permet d'effectuer les vérifications de typage sur les messages envoyés et la commande **Générer** provoque la génération du squelette d'application PEPLOM correspondant à la classe projet courante.

On notera dans le menu **Edition**, les commandes habituelles **Couper**, **Copier**, **Coller** et **Dupliquer**. L'action de strois premières est virtuelle : **Couper** supprime une classe de la vue courante et **Copier/Coller** permet de rendre une classe visible dans une vue mais ne modifie pas le contenu du dictionnaire de classe. La commande **Dupliquer** en revanche, permet de copier physiquement une classe et donc d'ajouter une nouvelle classe au dictionnaire après renommage.

Les commandes du menu **Classe** ne sont pas activables dans la fenêtre principale. Dans cette fenêtre, le concepteur peut seulement placer des racines de persistance et

des racines d'applications accessibles via les commandes du menu **Racines**.

Dans le menu **Vue** on retrouve les commandes permettant de créer des vues explicites ou d'obtenir les vues calculées prédéfinies (graphes). On notera que la commande **Contexte** n'est pas activable dans la fenêtre principale.

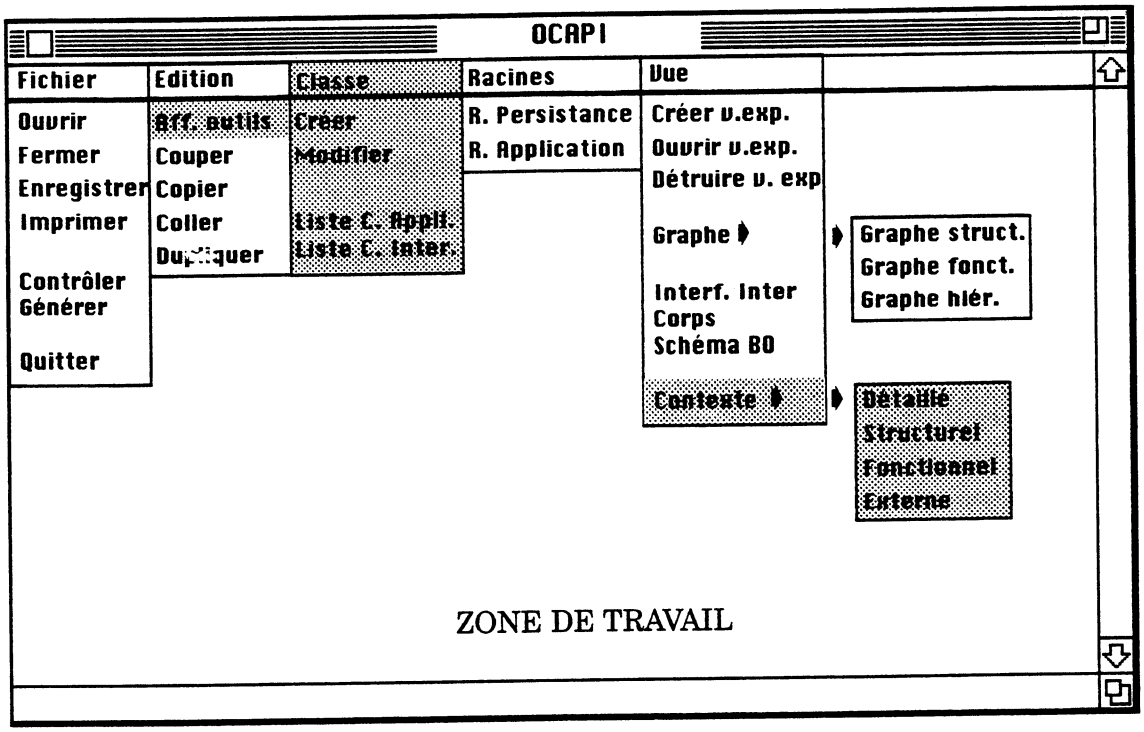


Figure 6-5 : Commandes disponibles dans la fenêtre principale d'OCAPI.

5.2.3.2 Fenêtre de vue

L'essentiel du travail de conception se déroule dans les vues et plus particulièrement dans les vues explicites. La figure 6-6 montre l'organisation d'une vue : les commandes utilisables, une palette d'outils et une zone de travail dont le contenu respecte la syntaxe du langage graphique MOSAÏC. Les commandes utilisées couramment sont regroupées dans une palette détachable qui permet : les sélections, la création des principales classes, la création des liens structurels, fonctionnels et des relations d'héritage. Cette palette permet également de créer des attributs, des méthodes et des états.

Le menu **Classe** permet de créer des classes ou de modifier une classe préalablement sélectionnée dans la zone de travail. Les commandes **Liste C. Appli.** et **Liste C. Inter.** affichent la liste des classes applicatives et la liste des classes interactives présentes dans le dictionnaire. Une classe est visible dans une vue explicite soit parce que le concepteur l'a créée dans cette vue soit parce qu'il l'a copiée depuis une autre vue ou depuis une des listes de classes.

Dans le menu **Vue**, l'activation de la commande **Graphe** doit être précédée par la sélection d'une classe, d'une méthode ou d'un attribut selon le type de graphe souhaité. La commande **Contexte** permet d'afficher les éléments de la fenêtre courante selon différents points de vue.

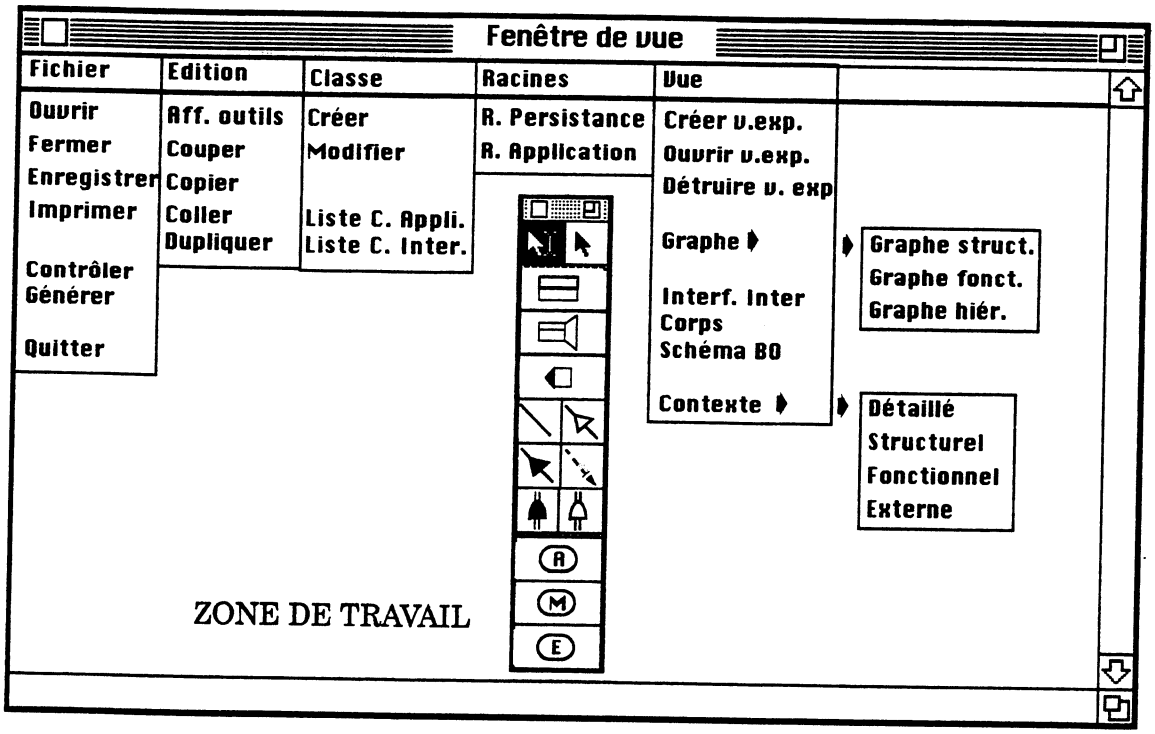


Figure 6-6 : Commandes disponibles dans une vue OCAPI.

5.2.3.3 Vérifications et génération

Les règles d'utilisation du modèle MOSAÏC présentées dans les chapitres 4 et 5 doivent être vérifiées par l'outil, comme par exemple les contraintes liées aux communications entre les classes interactives et les classes applicatives. En l'absence d'un langage de programmation des algorithmes, les autres vérifications effectuées par l'outil sont limitées aux contrôles de la validité des messages émis par les méthodes. Ces messages doivent concerner des méthodes publiques du fournisseur.

La génération de code PEPLOM suit les règles de traductions présentées dans la section 3.2 de ce chapitre. Ultérieurement, lorsque des classes spécialisées PEPLOM permettront de programmer l'interface interactive d'une application, le générateur sera étendu pour traduire les classes interactives.

5.3 Réalisation de l'outil OCAPI

5.3.1 Choix de réalisation

L'objectif a été d'une part de développer rapidement un prototype et d'autre part de pouvoir aisément modifier et réutiliser ce prototype. Une première solution consistait à programmer OCAPI directement à l'aide d'un environnement graphique comme X-windows ou la Toolbox du Macintosh. Cette solution permet de limiter l'impact du choix technique sur l'interface d'OCAPI et de respecter au plus près les spécifications externes présentées brièvement dans la section 5.2. L'inconvénient majeur de cette solution est que la maîtrise de tels environnements graphiques est longue et fastidieuse. Le choix s'est porté sur une solution plus aisée à mettre en œuvre : l'utilisation du générateur d'AGL GraphTalk [RX91]. L'approche générateur de GraphTalk permet de développer rapidement des prototypes aisés à modifier. En revanche, GraphTalk impose différentes contraintes sur le fonctionnement de l'atelier généré ; elles ont été jugées mineures à ce stade de développement et face aux avantages offerts par l'utilisation d'un tel générateur.

5.3.2 GraphTalk

GraphTalk est un générateur d'AGL fondé sur un modèle à objets. La phase de génération d'un atelier est précédée par une phase appelée «métamodélisation». Le métamodèle élaboré pendant cette phase décrit le modèle supporté par le futur outil à l'aide des concepts offerts par GraphTalk. Un métamodèle du modèle MOSAÏC a été élaboré à l'aide des concepts offerts par GraphTalk. A partir de ce métamodèle, GraphTalk a généré automatiquement l'atelier OCAPI.

Pour décrire le métamodèle, GraphTalk offre un langage graphique comportant trois métaclasses comportant des instances prédéfinies [RX91] :

- La métaclasse **MétaObjet** comporte quatre instances : Objet, Graphe, Lien et Propriété.
- La métaclasse **MétaLien** comporte six instances : Composition, Héritage, Affectation, Connexion, Exclusion et Inclusion.
- La métaclasse **MétaPropriété** comporte quatre catégories d'instances pour chacune des classes instances de la métaclasse MétaObjet. Chaque catégorie de propriétés permet de préciser les caractéristiques d'une instance de MétaObjet. Par exemple, la classe Objet possède les propriétés suivantes : Nom, Démons (pour attacher des démons appelés lors des occurrences de différents événements remarquables de la vie d'un objet : création, destruction, modification, etc.), Forme (pour définir sa forme graphique), Méthodes (pour définir des traitements associés à la classe), etc.

Le processus de métamodélisation a consisté à décrire chaque concept du modèle MOSAÏC à l'aide des classes proposées par GraphTalk : la nature du concept, sa forme graphique, ses propriétés et ses règles de comportement. La phase métamodélisation est essentiellement graphique mais des programmes en langage C ont du être ajoutés pour programmer les démons et pour réaliser des commandes spécifiques d'OCAPI. C'est le cas pour la commande de génération qui a nécessité l'écriture de différentes procédures C.

Pour écrire ces programmes, GraphTalk offre une boîte à outils dont les fonctions permettent d'une part d'accéder aux différents objets gérés par l'atelier et d'autre part d'intervenir dans le dialogue avec l'utilisateur.

5.3.3 Version actuelle d'OCAPI

La version actuelle d'OCAPI intègre une large part du langage graphique MOSAÏC et des spécifications présentées dans la section 5.2 qui sont le résultat de différentes expérimentations avec les versions successives de l'outil développé au cours de l'année 1992 par Nicolas Pudda [Pudd92]. OCAPI a nécessité l'écriture d'environ 3000 lignes de code C dont 25% sont des appels à la boîte à outils de GraphTalk.

Les notions prises en compte actuellement sont celles de classe projet, de classe applicative, de classe interactive, de constructeur, de classe de base, de racine de persistance et de racine d'application. Les trois relations structurelles sont supportées ainsi que les messages entre les méthodes et les deux relations d'héritage. En revanche, les dépendances fonctionnelles et les états ne sont pas pris en compte actuellement.

Dans la version actuelle d'OCAPI, la génération de code PEPLOM est limitée à la production d'un squelette donnant la définition du module principal et la définition des en-têtes des classes applicatives ; cette première réalisation limitée montre la faisabilité d'un atelier plus complet.

6 Conclusion

La technique de programmation d'un schéma de conception MOSAÏC en O2 a été expérimentée avec succès sur une partie de l'application «Gestion de compétitions sportives» (cf. Annexe C). En revanche les règles de traduction vers PEPLOM n'ont pas pu être validées concrètement car le compilateur a été disponible tardivement et seulement partiellement. Cependant, compte tenu des similitudes entre O2 et PEPLOM, la traduction ne devrait pas poser de difficultés majeures. Des extensions du langage PEPLOM nous permettront de tester plus complètement la méthode et l'outil sur des applications concrètes en particulier pour programmer les classes interactives.

L'utilisation de GraphTalk a permis de développer rapidement un prototype d'atelier et de le modifier progressivement au cours de versions successives qui ont suivi les évolutions de notre modèle MOSAÏC.

L'outil OCAPI décrit et partiellement réalisé est indispensable pour appliquer MOSAÏC. Il serait intéressant de le compléter par des outils d'aide à la conception susceptibles de suggérer au concepteur des éléments de l'architecture du système à concevoir. Un tel outil a été développé pour le modèle PAC [NC92] et pourrait être adapté à MOSAÏC étant donné les similitudes de ces deux modèles en ce qui concerne la gestion du dialogue homme-machine.

Chapitre 7

Conclusion

1 Intérêt du travail

La phase de conception à laquelle nous nous sommes limités permet d'élaborer l'architecture des systèmes d'applications mais ne s'intéresse pas aux spécifications de ces systèmes supposées définies pendant la phase d'analyse. Les systèmes d'applications sont caractérisés par des données persistantes partagées par différentes applications hautement interactives.

Ce travail prend sa source dans un double constat : d'une part, les méthodes de conception existantes sont mal adaptées aux langages de programmation de bases d'objets et d'autre part, elles sont insuffisantes pour aboutir à des architectures logicielles permettant de répondre aux exigences actuelles des systèmes d'applications. Aucune méthode de conception ne prend en compte, selon des techniques actuelles, les trois dimensions d'un système d'applications : la définition des données persistantes partagées, la conception des fonctions spécifiques des applications et l'élaboration d'un gestionnaire de dialogue permettant un échange souple et riche à l'initiative de l'utilisateur.

Les méthodes d'analyse et de conception de systèmes d'information sont généralement antérieures à l'apparition des technologies à objets et ne proposent donc pas de guide pour élaborer une solution technique à l'aide de telles technologies. Les méthodes de conception orientées objets intègrent les techniques à objets mais s'intéressent seulement au développement des programmes à l'aide de langages de programmation sans tirer parti des avantages offerts par les Langages de Programmation de Bases d'Objets.

Le résultat principal de notre travail constitue un ensemble de propositions pour une méthode de conception adaptée au développement de systèmes d'applications à l'aide d'un Langage de Programmation de Bases d'Objets combinant des services provenant de trois domaines : de la programmation, des bases de données et des systèmes interactifs.

1.1 Contexte initial

Les trois domaines (la programmation, les bases de données et les systèmes interactifs) ont longtemps évolué de façon indépendante conduisant au développement de techniques spécifiques. Le développement d'une application complète nécessite la combinaison et l'intégration de différentes techniques actuelles ou plus anciennes selon l'environnement utilisé. Ainsi des bases de données relationnelles normalisées peuvent être utilisées par des programmes structurés, des interfaces homme-machine sophistiquées combinées avec des fichiers traditionnels, etc.

L'organisation logicielle conventionnelle d'une application s'appuie sur un modèle d'architecture implicite dans lequel on distingue : la base de données et les programmes d'applications. Le schéma de la base de données est conçu comme un modèle du domaine à l'aide des concepts offerts par le logiciel de base utilisé pour gérer les données persistantes (Fichier, Réseau, Relationnel, etc.). La structure des programmes dépend du critère de décomposition appliqué (fonctionnel, machines abstraites, orienté objet, etc.) et du langage de programmation utilisé (procédural, modulaire, orienté objet, etc.). Les modèles d'architecture proposés pour concevoir des systèmes interactifs s'appuyent sur deux composants principaux : l'interface interactive et le composant fonctionnel (cf. figure 7-1).

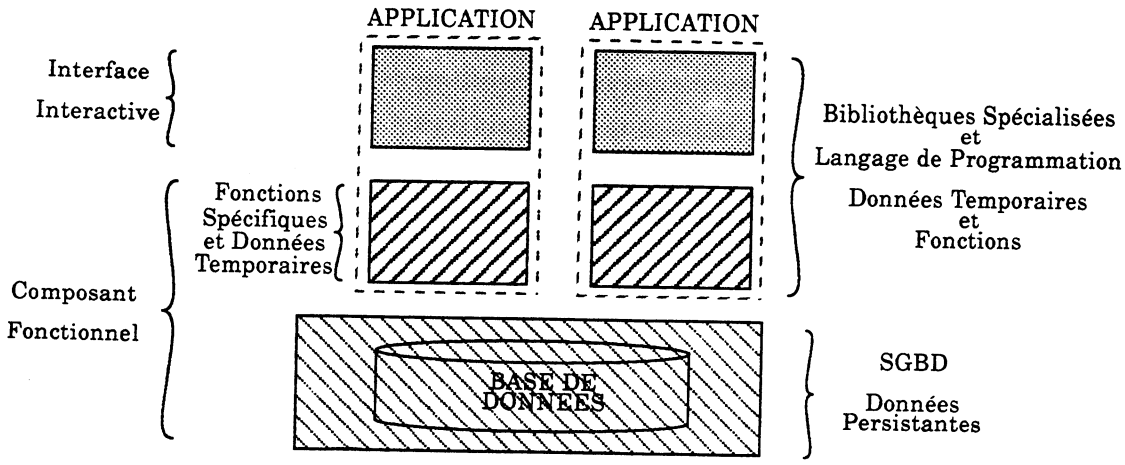


Figure 7-1 : Modèle conventionnel d'architecture.

Les techniques à objets ont été adoptées par la plupart des domaines de l'informatique mais leur interprétation diffère en fonction du contexte dans lequel elles sont utilisées. Dans les bases de données, un objet reste souvent un élément regroupant des données potentiellement persistantes alors qu'en programmation, il représente un module logiciel offrant un ensemble de fonctions. Dans les systèmes interactifs conçu à l'aide des modèles multiagents, l'objet est un concept bien adapté à la programmation des agents chargés de gérer le dialogue entre l'utilisateur et l'application.

1.2 Nature des propositions

Les Langages de Programmation de Bases d'Objets sont fondés sur le concept d'objet encapsulant des données et des méthodes et ils gèrent à la fois des objets persistants et des objets temporaires. Ils offrent souvent des classes spécialisées adaptées à la gestion du dialogue homme-machine. Dans certains LPBO, un objet persistant peut comporter à la fois des données persistantes et des données temporaires.

Dans ce nouveau contexte, le modèle d'architecture MOSAÏC propose une organisation logicielle originale des systèmes d'applications qui exploite les services offerts par les Langage de Programmation de Bases d'Objets. Ce modèle d'architecture remplace la séparation stricte entre les données persistantes et les programmes d'application par une décomposition plus souple du système en classes interactives et classes applicatives. Une application est composée d'une interface interactive, constituée d'objets interactifs, et d'un corps constitué d'objets applicatifs temporaires ou persistants. Les classes applicatives sont obtenues par un processus de modélisation du domaine d'application. Les classes interactives sont obtenues par un processus de modélisation du dialogue homme-machine. Les objets applicatifs comme les objets interactifs sont développés avec le même Langage de Programmation de Bases d'Objets (cf. figure 7-2).

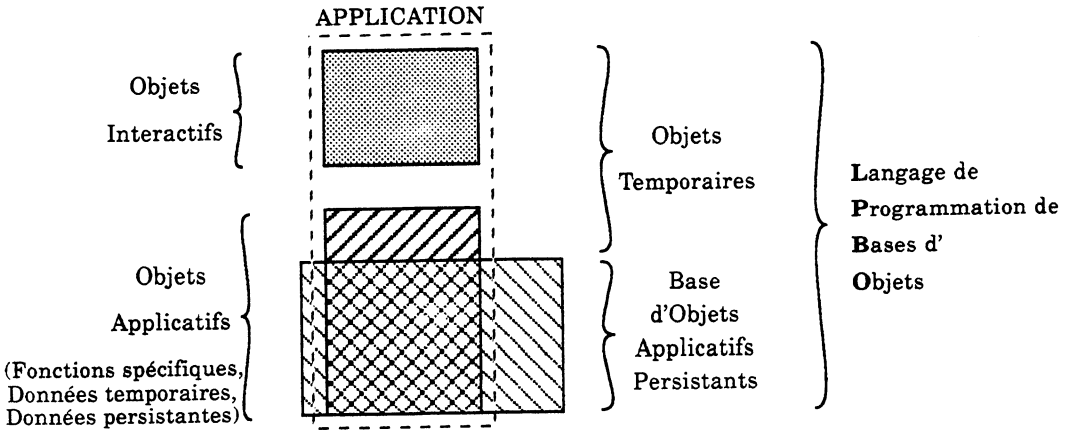


Figure 7-2 : Modèle d'architecture pour une application.

Un système d'applications est alors conçu comme une projet composé d'une base d'objets applicatifs persistants et d'applications (cf. figure 7-3). Chaque application possède son interface propre et les objets interactifs ne sont pas partageables. Les objets applicatifs qui constituent le corps des applications peuvent être partagés par plusieurs applications. Les classes interactives et les classes applicatives sont réutilisables dans différentes applications.

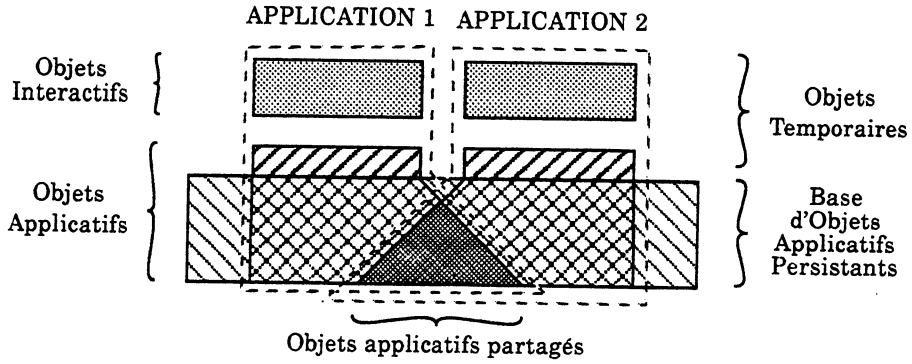


Figure 7-3 : Modèle d'architecture pour un système d'applications.

A partir de ce modèle général d'architecture, nous nous sommes attachés à préciser les concepts d'un modèle supporté par un langage textuel et un langage graphique. Les principes qui nous ont guidés pour définir le modèle et les langages sont d'une part une bonne orthogonalité des concepts et d'autre part une richesse et une souplesse suffisantes pour prendre en compte toutes les dimensions d'un système d'applications : structurelle, fonctionnelle et dynamique. Nous avons distingué :

- les classes interactives, les classes applicatives et les classes projets,
- les attributs, les variables et les relations structurelles (référence, dépendance existentielle et propriété),
- les méthodes, les messages et les dépendances fonctionnelles,
- les états, les règles d'intégrité, les préconditions et les postconditions.

Nous avons expérimenté sommairement une technique de traduction permettant de programmer une conception MOSAÏC en O_2 et en PEPLOM. Ces expérimentations nous ont permis de proposer quelques éléments d'une démarche de conception adaptée à notre approche. L'outil OCAPI développé avec GraphTalk constitue un premier niveau d'aide à la conception.

2 Bilan

Le principal objectif de nos travaux était de proposer une méthode de conception utilisable pour développer un système d'applications à l'aide d'un Langage de Programmation de Bases d'Objets. Sur ce point, les premières expérimentations nous ont permis de constater que nous avions partiellement atteint nos objectifs. Nous avons pu concevoir l'application «Gestion de Compétitions Sportives» à l'aide de MOSAÏC puis la pro-

grammer partiellement à l'aide du SGBDOO O_2. Cependant, les limites actuelles d'O_2 en ce qui concerne la présentation des opérations déclenchables par l'utilisateur ne nous ont pas permis de respecter complètement les spécifications externes de cette application.

MOSAÏC permet de systématiser la recherche et la conception des composants d'un système d'applications suivant différentes catégories selon la partie considérée de ce système d'applications. La comparaison de MOSAÏC avec les autres méthodes de conception orientées objets (cf. figure 7-4) montre que les apports principaux de notre méthode sont d'une part une bonne adéquation aux LPBO et d'autre part une prise en compte homogène et avec des techniques à objets actuelles, des trois dimensions d'un système d'applications : la définition des données persistantes, la conception des fonctions spécifiques et l'élaboration du gestionnaire de dialogue.

Nos travaux concrétisent une remise en cause de la séparation conventionnelle entre les données persistantes et les programmes tout en conservant les acquis des systèmes antérieurs, comme la notion de schéma de base de données représentée dans MOSAÏC par le schéma de la base d'objets.

Les bénéfices attendus d'une telle remise en cause sont une meilleure cohésion des classes, un couplage interclasse plus faible et une plus grande évolutivité des applications et ainsi qu'une réutilisabilité accrue des classes. Si le gain en termes de couplage et de cohésion est indéniable, puisque la persistance n'est plus un critère fondamental de décomposition, nous n'avons pas actuellement de résultat sur l'efficacité de nos propositions ni sur leur facilité d'utilisation. Cependant, compte tenu de la croissance constante de la puissance des ordinateurs et la persistance de la crise du logiciel, la qualité des programmes doit primer sur l'efficacité. Par contre, seule une utilisation intensive de MOSAÏC sur différentes applications apportera des résultats concernant la facilité d'utilisation.

L'inconvénient majeur de notre approche est de lier fortement les programmes aux données persistantes et ainsi risquer de nuire à l'évolutivité des programmes ou du schéma de la base d'objets. Nous avons restreint la visibilité des classes par des règles méthodologiques pour limiter le couplage entre classes et améliorer l'évolutivité des applications. Une évolution du schéma de la base d'objets a souvent des répercussions sur les programmes qui utilisent ces données ; avec notre approche, ces modifications sont limitées aux classes décrivant ces données et aux classes clientes. Une évolution des programmes entraîne une modification des classes mais n'affecte pas directement les données persistantes or, dans un LPBO, les évolutions de schéma sont plus aisées à gérer lorsqu'elles concernent les méthodes que lorsqu'elles concernent les données persistantes.

	OOD 91	OOA/OOD	Class/Resp.	OMT	Mecano	MOSAÏC
Phases du cycle de vie	Conception	Analyse-Conception	Conception	Analyse-Conception	Conception	Conception
Type d'applications	Temps Réel	Toutes	?	Toutes	Toutes	Appli. BD Interactives mono-utilisat.
Systèmes cibles	ADA (LOO)	LOO	LOO	LOO	LOO	LPBO SGBDOO
Modèle d'Architecture	non	4 composants spécialisés	non	non	non	Schéma de BO Classes spécialisées
Composant Fonctionnel	oui	oui	oui	oui	oui	oui
Composant Base de Données	spécification au niveau logique précisions dans le Composant Fonctionnel	1 composant spécialisé pas de concepts spécifiques	non	non	non	Schéma de la Base d'Objets Classes applicatives
Composant Gestion du Dialogue	non	1 composant spécialisé pas de concepts spécifiques	non	non	non	Interface interactive Classes interactives
Différents Schémas ou Diagrammes	Diag. de classes, d'objets et de transitions Diag. de modules, de processus	Diagramme de classes	Diagramme de classes	Sch. objet, Sch. dynamique, Sch. fonctionnel	Diagramme de classes	Schéma de la Base d'Objets Interface Interactive Corps d'une application
Différentes catégories de classes	classe, classe utilitaire	classe, classe&objets	classe	classe	classe classe abstraite composite	cl. applicative cl. interactive cl. facette classes projets
Différentes relations structurelles	relation	relation, tout/partie	partie-de	associations binaires, n-aires	communication composition, util. secondaire	propriété dép. existentielle référence(bidir)
Différentes relations fonctionnelles	message	message	message	diagramme de flots de données	communication composition util. secondaire	message dépendance
Expression de la dynamique	diagramme de transitions	diagramme de transitions	-	diagramme de transitions	pré/post cond. invariant	pré/post cond. règles d'intégr.
Différentes relations d'héritage	héritage, métaclasse	héritage	sorte-de, analogue-à	héritage	spécialise, réalise, adapte, implante, se comporte comme, fusionne	est-un, adapte

Figure 7-4 : Tableau comparatif des méthodes de conception orientées objets (cf. figure 3-8).

Après différentes évolutions, le modèle et le langage développés dans le projet Aristote se sont stabilisés fin 1992 sous la forme du langage PEPLOM. Nous avons participé à l'élaboration des premières versions du modèle Aristote, puis nous nous sommes attachés à définir notre modèle et notre méthode sans référence à un système cible particulier. Ces travaux nous permettent maintenant de suggérer différentes extensions à des langages comme PEPLOM, comme l'introduction de classes spécialisées pour programmer le dialogue homme-machine, un mécanisme d'exportation sélective et l'ajout d'une instruction particulière permettant d'exploiter le type d'un objet sans remettre en cause un typage fort. L'approche MOSAÏC est bien adaptée pour développer une application en PEPLOM et nous avons proposé des règles de traduction. L'outil OCAPI, dédié à MOSAÏC, permet de créer et de maintenir des schémas de conception de systèmes d'applications.

3 Perspectives

Ce travail doit être poursuivi dans plusieurs directions.

La méthode MOSAÏC est actuellement limitée aux applications mono-utilisateur sans parallélisme. Les systèmes d'applications sont par essence des systèmes multi-utilisateurs. Il est donc nécessaire d'approfondir la méthode pour prendre en compte cet aspect. Un tel travail doit préciser le rôle et le comportement des objets applicatifs utilisés simultanément par plusieurs utilisateurs. Pour cela, le comportement des données temporaires encapsulées dans un objet applicatif persistant, lorsque cet objet est partagé par plusieurs applications, doit être affiné. Par ailleurs, la notion de transaction n'apparaît pas explicitement et elle doit être précisée en relation avec la notion de méthode primaire. Une étude a déjà démarré dans ce sens au sein du projet Aristote [Mach92].

Hormis différentes règles méthodologiques, nous n'avons pas abordé la délicate question des critères de qualité. Pour approfondir cet aspect et fournir au concepteur un ensemble de règles et éventuellement une métrique, on peut s'appuyer sur les rôles fixés aux différentes classes MOSAÏC. Un objet applicatif a un double rôle : celui de composant modulaire au sein d'une application et celui d'élément regroupant un ensemble de données potentiellement persistantes. Les critères de qualité applicables aux classes applicatives doivent combiner ceux du génie logiciel et ceux des bases de données. En particulier, il serait souhaitable d'aboutir à des degrés de normalité des classes analogues aux formes normales du modèle relationnel. Un objet interactif est un composant modulaire du gestionnaire de dialogue. Les critères de qualité applicables aux classes interactives doivent combiner ceux du génie logiciel et ceux du domaine des interfaces homme-machine.

Une validation de MOSAÏC peut être envisagée selon deux axes complémentaires :

une approche formelle et une approche expérimentale. Nous sommes convaincus de la nécessité d'une formalisation mais nous n'avons pas abordé ce point d'une part pour des raisons de temps et d'autre part parcequ'il n'existe pas actuellement, malgré différentes propositions, de formalisation consensuelle des concepts de base des techniques à objets. L'expérimentation réalisée avec MOSAÏC est limitée et insuffisante pour juger de l'efficacité de nos propositions. Aussi doit-elle être complétée par des expérimentations sur différentes applications et menées par différents concepteurs. De telles expérimentations nécessitent le développement d'une nouvelle version de l'outil OCAPI pour constituer un environnement plus complet et plus efficace. De nouvelles recherches sont nécessaires pour développer des outils facilitant la modélisation et la validation des besoins des utilisateurs [Roll92] sans oublier la formation à ces nouveaux concepts [Bézi90].

Nous avons limité nos travaux à l'étape de conception puis nous avons indiqué comment programmer une conception MOSAÏC en O_2 et en PEPLOM. Nous avons évoqué la complémentarité possible entre MERISE et MOSAÏC, mais, pour constituer une méthode complète, il est nécessaire de proposer une méthode d'analyse mieux adaptée à nos propositions. L'avantage attendu de l'intégration des techniques à objets dans des modèles de spécification est une meilleure continuité dans le processus de développement. Cependant, l'utilisation des techniques à objets pendant l'étape d'analyse doit apporter la preuve qu'elles conduisent à de meilleurs résultats pour les objectifs de l'analyse : instauration d'un langage commun entre les acteurs de l'entreprise, amélioration des échanges entre les utilisateurs et les informaticiens et élaboration de spécifications plus cohérentes et plus complètes. Il serait souhaitable à terme de disposer de véritables schémas conceptuels orientés objets [BBC*91].

Avec l'augmentation de la puissance des ordinateurs, la généralisation des réseaux et l'apparition constante de nouvelles technologies, il est nécessaire de définir des concepts permettant d'une part de bâtir un système d'applications de façon modulaire et d'autre part d'intégrer les applications existantes. Dans cette perspective, nous avons introduit la notion de projet. Ce concept de projet et plus particulièrement celui de projet composé sont à approfondir et à expérimenter sur des systèmes d'applications de grande dimension.

Pour conclure, la richesse croissante des logiciels de base, l'augmentation de la complexité des applications et les exigences toujours plus pointues des utilisateurs nous poussent à penser que le temps où les recherches et les développements en programmation, en bases de données et en interfaces homme-machine pouvaient être menés de façon indépendante est révolu. C'est cette conviction qui a guidé nos réflexions et qui nous a conduits à intégrer les travaux récents menés dans ces différents domaines en nous appuyant sur la notion d'objet adoptée par la plupart des domaines de l'informatique. Cependant, le chemin est encore long avant d'avoir la certitude que les Langages de Programmation de Bases d'Objets sont utilisables efficacement et avant de disposer d'une méthode complète et robuste adaptée à de tels logiciels de base.

Bibliographie

[AB84] S. Abiteboul, N. Bidoit
Non First Normal Form relations : an algebra allowing data restructuring
ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1984.

[AB87] M.P. Atkinson, O.P. Buneman
Types and Persistence in Database Programming Languages
ACM Computing Surveys, Vol. 19, N° 2, June 1987.

[ABD*89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik
The Object-Oriented Database System Manifesto
DOOD89, Deductive and Object Oriented Databases, Kyoto, Japan, 1989.

[ABD*90] M. Adiba, F. Brissaud, P. Dechamboux, B. Defude, F. Exertier, J-P. Giraudin, S. Haj Houssain, C. Lenne
Modèle de base pour ARISTOTE (version 1)
Rapport Aristote, RAP 006, LGI-BULL-IMAG, Grenoble, Septembre 1990.

[ABG92] L. Alvares, F. Brissaud, J-P. Giraudin
Une approche objet adaptée à la conception d'applications de bases de données
7o Simpósio Brasiliero de Banco de Dados, Porto Alegre, Brésil, 13-15 Mai 1992.

[Abri72] J-R. Abrial
Projet SOCRATE
Rapport Technique, Université de Grenoble, 1972.

[Abri74] J-R. Abrial
Data semantics - Data Base Management
Klimbie J.W. and al eds, North Holland, Amsterdam, 1974.

[Abri78] J-R. Abrial
Manuel du langage Z
Document technique EDF, Paris, 1978.

[AC89] M. Adiba, C. Collet
L'approche Objet pour les Bases de Données
Survey Aristote, SUR 001, LGI-BULL-IMAG, Grenoble, Septembre 1989.

[ACC81] M. Atkinson, K. Chisholm, W. Cockshott
PS-algol : an Algol with a Persistent heap
ACM SIGPLAN Notices, 17(7), July 1981.

[Adre89] F. Adreit
Contribution à la spécification d'interfaces homme-SGBD
Thèse de Doctorat, USTL, Montpellier, Novembre 1989.

[ANSA89] ANSA
ANSA Reference Manual, Release 01.00
APM Cambridge Ltd., March 1989.

[ANSI75] ANSI / X3 / SPARC
Study Group on Data Management Systems : Interim Report 75-02-07
ACM SIGMOD Newsletter, FDT, Vol. 7, N°2, 1975.

[Aris91] Aristote
Générateur d'applications orientées objet, multimédia
Rapport Aristote, RAP 014, LGI-BULL-IMAG, Grenoble, Juin 1991.

[BB93] F. Barbier, J. Bézivin
Traçabilité : la clef de la conception par objets
Actesde la journée Technologie Objet, Nantes, Avril 1993.

[BBC*91] F. Brissaud, J. Brunet, C. Cauvet, J-P. Giraudin, M. Moreno, E. Penny, C. Rolland
Modélisation orientée objet : bilan d'une expérience
Congrès INFORSID 91, Paris, Juin 1991.

[BBL91] G. Blair, H. Bowman, R. Lea
Basic Concepts 1 : Objects, Classes and Inheritance
dans [BGHS91].

[BDMN73] G. Birtwistle, O. Dahl, B. Myhrhaug, K.Nygaard
SIMULA begin
Petrocelli Charter, New York, 1973.

[Beer89] C. Beeri
Formal Models for Object Oriented Databases
DOOD89, Deductive and Object Oriented Databases, Kyoto, Japan, 1989.

[Bézi86] J. Bézivin
Langages objets et prototypage
Rapport de Recherche 86-9, Laboratoire Informatique de Brest, 1986.

[Bézi90] J. Bézivin
Teaching Object-Oriented Methods
Panel in TOOLS, 2nd international conference, Paris, June 1990.

[BG90] F. Brissaud, J-P. Giraudin
Les Associations dans les Modèles de Données
Survey Aristote, SUR002, LGI-BULL-IMAG, Grenoble, Janvier 1990.

[BG91] F. Brissaud, J-P. Giraudin
Différents styles de réalisation en O2 et Aristote pour une même application
VIIèmes Journées "Bases de Données Avancées" - BD3, Lyon, Septembre 1991.

[BGHS91] G. Blair, J. Gallagher, D. Hutchison, D. Shepherd
Object-Oriented Languages Systems and Applications
PITMAN PUBLISHING 1991.

[BGJR92] A. Biliris, N.H. Gehani, H.V. Jagadish, W.D. Roome
ODE Object Database and Environment
Technical Report, AT&T BELL Laboratories, New-Jersey, 1992.

[BJ66] C. Böhm, G. Jacopini
Flow Diagram, Turing Machines And Langages With Only Two Formation Rule
Communications of the ACM, Vol. 9, N°5, May 1966.

[BJ79] D. Bert, P. Jacquet
Generic Abstract Data Types
5th annual conference, wc 2-1, IFIP, May 1979.

[BMP*90] J-C. Boussard, C. Michel, J-P. Partouche, R. Rousseau, M. Rueher
Une évaluation du langage Eiffel
Technique et Science Informatiques, Avril 1991.

[Boda86] F. Bodart
IDA : une Méthode de Conception Assistée des Systèmes d'Information
Génie Logiciel, ADI, N° 4, 1986.

[Booc82] G. Booch
Object Oriented Design
Ada Letters Vol. 1, N° 3, March/April 1982.

[Booc83] G. Booch
Software Engineering With ADA
Benjamin/Cummings Publishing Company, Inc., 1983.

[Booc86] G. Booch
Object-Oriented Development
IEE Transaction on Software Engineering, Vol. SE-12, February 1986.

[Booc91] G. Booch
Object Oriented Design with Applications
Benjamin/Cummings Publishing Company, Inc., 1991.

[Borg85] A. Borgida
Features of Languages for the flexible Handling of Exception in Informations Systems
ACM Transaction on Database Systems, 10(4), Decemder 1985.

[BOS91] P. Butterworth, A. Otis, J. Stein
The Gemstone Object Database Management System
Communication of the ACM 34(10), October 1991.

[Brac83] R.J. Brachman
What IS-A Is and Isn't : An Analysis of Taxonomic Links in Semantic Networks
Computer, Vol. 16, N° 10, October 1983.

[Bris90] F. Brissaud
Des associations pour un modèle à objets
4èmes Journées «Pratique des Méthodes et Outils Logiciels d'Aide à la Conception de Systèmes d'Information», Nantes, Septembre 1990.

[Bris91a] F. Brissaud
Eléments d'un modèle conceptuel orienté objet adapté aux approches Programmation et Bases de Données
Rapport Aristote, RAP0011, LGI-BULL-IMAG, Grenoble, Février 1991.

[Bris92a] F. Brissaud
Une nouvelle approche des objets pour la conception de systèmes d'information
6èmes Journées Internationales des Sciences Informatiques, Tunis, Tunisie, Mai 1992.

[Bris92b] F. Brissaud
Notion d'objet dans la conception de systèmes d'information
5th International Conference on : «Putting into practice Methods and Tools for Information System Design», Nantes, Septembre 1992.

[Brun91] J. Brunet
Comparative view of object-oriented analysis and conceptual modelling techniques
BUSINESS CLASS Project n°5311, Esprit II report n°BC.R.TS.T31.1, 1991.

[Call91] F.W. Calliss
A Comparison of Module Constructs in Programming Languages
ACM SIGPLAN Notices, Vol. 26, No. 1, January 1991.

[Card84] L. Cardelli
A semantics of multiple inheritance
LNCS 173 "semantics of data type", June 1984.

[CB91] C. Collet, E. Brunel
A form system for an object-oriented database system
DEXA'91, 2nd International Conference on Database and Expert Systems Applciations, Berlin, August 1991.

[Cast91] X. Castellani
L'implémentation de concepts fondamentaux du modèle de la méthode MCO d'analyse et de conception des systèmes d'objets
Congrès INFORSID 1991, Paris, Juin 1991.

[CF92] D. Champeaux, P. Faure
A comparative study of object-oriented analysis methods
JOOP, March/April 1992.

[CGL*75] B. Cherbonneau, , M. Galinier, J-P. Lagasse, H. Massie, A. Mathis, J-L . Paul
Programmation Structurée
Rapport N° 112, Université Paul Sabatier, UER Informatique, Toulouse, 1975.

[Chen76] P. Chen
The Entity Relationship Model - Toward an unified view of data
ACM TODS, Vol. 1, N° 1, 1976.

[Chig90] M.H. Chignell
A taxonomy of user interface of terminology
SIGCHI Bulletin, Vol. 21, N° 4, April 1990.

[CN89] J. Coutaz, L. Nigay
General Agents and Rules
RP2/WP1, Amodeus Project Document : D2, December 1989.

[Coda69] Codasyl
A Survey of Generalised Data Base Management Systems
Technical Report, ACM, New-York, 1969.

[Coda71] Codasyl
Feature Analysis of Generalised Data Base Management Systems
Technical Report, ACM, New-York, 1971.

[Codd70] E. Codd
A Relationnal Model of Data for Large Shared Data Banks
Communications of the ACM, Vol. 13, N°6, June 1970.

[Codd79] E. Codd
Extending the Relationnal Model to capture more meaning
ACM Transaction on Database Systems, 4(4), December, 1979.

[Cohe84] T. Cohen
Data abstraction, data encapsulation and object-oriented programming
SIGPLAN Notices, N°1, Vol. 19, January 1984.

[Coll87] C. Collet
Les formulaires complexes dans les bases de données multimédia
Thèse de doctorat, USTMG, Grenoble, Novembre 1987.

[Cout90] J. Coutaz
Interfaces homme-ordinateur
DUNOD, 1990.

[CR89] C. Cauvet, C. Rolland
O : un modèle pour la conception de bases de données orientées objets*
Congrès INFORSID 1989, Nancy, France, 1989.

[CW85] L. Cardelli, P. Wegner
On understanding types, data abstraction and polymorphism
ACM Computing Surveys, 17(4), December 1985.

[CY90] P. Coad, E. Yourdon
Object-Oriented Analysis
Prentice-Hall, Englewood Cliffs, 1990.

[CY91] P. Coad, E. Yourdon
Object-Oriented Design
Prentice-Hall, Englewood Cliffs, 1991.

[DA82] C. Delobel, M. Adiba
Bases de données et systèmes relationnels
DUNOD, 1982.

[Date75] C-J. Date
An Introduction to Database Systems
ADDISON-WESLEY Publishing Company, 1975.

[DD91] P. Dechamboux, B. Defude
Modèle de base pour ARISTOTE (version 2)
Rapport Aristote, RAP 012, LGI-BULL-IMAG, Grenoble, Mai 1991.

[DDH72] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare
Structured programming
Academic Press, London and New York 1972.

[Dech92] P. Dechamboux
PEPLOM : un Langage de Programmation Persistant pour Manipulation de Données Complexes
Rapport Aristote, RAP 013, LGI-BULL-IMAG, Grenoble, Janvier 1992.

[Dech93] P. Dechamboux
Gestion d'objets persistants : du langage de programmation au système
Thèse de doctorat, Université Joseph Fourier, Grenoble I, Février 1993.

[DeMa78] T. De Marco
Structured Analysis and System Specification
Yourdon, New-York, 1978.

[DH89] R. Ducourneau, M. Habib
La multiplicité de l'héritage dans les langages à objets
Technique et Science Informatiques, Janvier 1989.

[Dijk68a] E.W. Dijkstra
GOTO Statement Considered Harmful
Communication of the ACM, 11, 3, March 1968.

[Dijk68b] E.W. Dijkstra
The structure of THE Multiprogramming System
Communication of the ACM, 11, 5, May 1968.

[Dijk68c] E.W. Dijkstra
A Constructive Approach to the Problem of Program Correctness
BIT 8, 1968.

[DLR91] C. Delobel, C. Lécluse, P. Richard
Bases de Données : des systèmes relationnels aux systèmes à objets
InterEditions, 1991.

[DN66] O.J. Dahl, K. Nygaard
SIMULA- An ALGOL-Based Simulation Language
Communications of the ACM, Vol. 9, N°9, September 1966.

[Ferb90] J. Ferber
Conception et programmation par objets
Technologies de pointe, Informatique, Hermès ,1990.

[Gard83] G. Gardarin
Bases de données : les systèmes et leurs langages
Eyrolles, 1983.

[GH78] J-V. Guttag, J-J. Horning
The algebric specification of abstract data type
Acta Informatica, Vol. 10, No. 1, 1978.

[Gira90] J-P. Giraudin
Méthodes d'Analyse et de Conception des Systèmes d'Information
Rapport Aristote, SUR 004, LGI-BULL-IMAG, Grenoble, Octobre 1990.

[Giro90] X. Girod
Mecano a method for object-oriented software construction
TOOL'S90, Paris, 1990.

[Giro91] X. Girod
Conception par Objets - MECANO : une Méthode et un Environnement de Construction d'Applications par Objets
Thèse de doctorat, Université Joseph Fourier, Grenoble I, Juin 1991.

[GMB82] S.J. Greenspan, J. Mylopoulos, A. Borgida
Capturing More Word Knowledge in the Requirement Specification
6th ICSE, IEE, 1982.

[GR83] A. Goldberg, D. Robson
Smalltalk-80 : The Language and its Implementation
Addison Weslay, Reading, MA, 1983.

[GS79] C. Gane, T. Sarson
Structured Systems Analysis
Prentice Hall, 1979.

[Gutt77] J. Guttag
Abstract Data Types and the Development of Data Structures
Communication of the ACM, Vol. 20, N°6, June 1977.

[Habr88] H. Habrias
Le modèle relationnel binaire - Méthode I.A. (NIAM)
EYROLLES, 1988.

[Heit87] M. Heitz
HOOD, une Méthode de Conception Hiérarchisée Orientée Objets pour le développement des Gros Logiciels Techniques et Temps-Réel
Journées ADA France, Bigre No. 57, Décembre 1987.

[HK87] R. Hull, R. King
Semantic Database Modeling : Survey, Applications and Research Issues
ACM Computing Surveys, Vol. 19, N° 3, September 1987.

[HL89] C. Henry, M. Lott
La méthode de conception MACH2
Congrès de Génie Logiciel CGL4, Toulouse, 1989.

[HM83] M. Hammer, D. McLeod
Database Description with SDM : A Semantic Database Model
ACM Transaction on Database Systems, 6(3), September, 1983.

[Hoar69] C.A.R. Hoare
Proof of Program : FIND
Communication of the ACM, Vol. 14, N°1, January, 1971.

[Ichb79] J-D. Ichbiah et al.
Rationale for the Design of the ADA Programming Language
SIGPLAN Notices, June 1979.

[IGL84] *Introduction à la méthode MACH*
IGL Technology, 1984.

[Jack83] M.A. Jackson
System Development
Prentice Hall, 1979.

[Jacq78] P. Jacquet
Les types abstraits génériques, propositions pour un mécanisme d'abstraction dans les langages de programmation
Thèse de 3ème cycle, USTMG, Grenoble I, 1978.

[Jone86] C.B. Jones
Systematic Software Development using VDM
Prentice-Hall, 1986.

[KA90] S. Khoshafian, R. Abnous
Object Orientation : Concepts, Languages, Databases, User Interfaces
John Wiley & Sons, Inc., 1990.

[KBB*87] W. Kim, N. Ballou, J. Banerjee, H.T. Chou, J.F. Garza, D. Woelk
Features of the ORION object-oriented database system
MCC Technical report number ACA-ST-308-87, september 1987.

[KMR*88] S. Krakowiak, M. Meysembourg, M. Riveill, C. Roisin
Modèles d'objets et langage pour la programmation d'applications réparties
Génie Logiciel & Systèmes Experts, N°11, Mars 1988.

[Knut68] D.E. Knuth
The Art of Computer Programming - Tome 1 : Fundamental Algorithms
Addison-Wesley, 1968.

[Krak82] S. Krakowiak
Systèmes intégrés de production de logiciel : concepts et réalisations
Techniques et Sciences Informatiques, Vol. 1, N° 3, 1982.

[LeMo77] J-L. Le Moigne
La Théorie du Système Général
PUF, Paris, 1977.

[LeMo90] J-L. Le Moigne
La modélisation des systèmes complexes
Dunod, 1990.

[LH89] K. Lieberherr, I. Holland
Formulations and Benefits of the Law of Demeter
SIGPLAN Notices, Vol. 24, N° 3, March 1989.

[LHR88] K. Lieberherr, I. Holland, A. Riel
Object-Oriented Programming : An Objective Sense of Style
Proceedings OOPSLA'88, San Diego, California, USA, September 1988.

[LMW79] R.C. Linger, H.D. Mills, B.I. Witt
Structured programming : Theory and Practice
Addison-Wesley Publishing Company, 1979.

[LP91] W. Lalonde, J. Pugh
Subclassing ≠ Subtyping ≠ Is-a
Journal of Object-Oriented Programming, January 1991.

[LZ74] B.H. Liskov, S.N. Zilles
Programming with Abstract Data Types
Proc. ACM SIGPLAN Symposium on Very High Level Language, SIGPLAN Notices, 9, April 1974.

[LZ75] B.H. Liskov, S.N. Zilles
Specification Techniques for Data Abstractions
IEEE Transaction on Software Engineering, SE-1, 1, march 75.

[Mach92] J. Machado
Expression du Parallélisme dans les Applications de Bases de Données
Note Technique Aristote, NOT 015, LGI-BULL-IMAG, Grenoble, Septembre 1992.

[Mart91] H. Martin
Contrôle de la cohérence dans les bases d'objets : une approche par le comportement
Thèse de doctorat, Université Joseph Fourier, Grenoble I, Janvier 1991.

[MB78] B. Meyer, C. Baudoin
Méthodes de Programmation
Editions Eyrolles, 1978.

[Meye88] B. Meyer
Object-Oriented Software Construction
Prentice Hall International, 1988.

[Moit82] A. Moitra
Direct implementation of algebraic specifications of abstract data type
IEEE Transaction on Software Engineering, Vol. SE-8, N°1, 1982.

[MNC*89] G. Masini, A. Napoli, D. Colnet, D. Léonard, K. Tombre
Les langages à objets
InterEditions, 1989.

[NC91] L. Nigay, J. Coutaz
Building User Interfaces : Organizing Software Agents
ESPRIT'91 proceedings, Commission of the European Communities, DG XIII, 1991.

[NC92] L. Nigay, J. Coutaz
PAC-Expert : Towards an Automatic Generation of Dialogue Controllers
ESPRIT Basic Research Action 3066, AMODEUS Project, February 1991.

[Nels91] G. Nelson
System Programming with Modula-3
Prentice-Hall, 1991.

[Norm92] V. Normand
Le modèle SIROCO : de la spécification conceptuelle des interfaces utilisateurs à leur réalisation
Thèse de doctorat, Université Joseph Fourier, Grenoble I, Avril 1992.

[ODeu89] ODeux
The O2 Book
GIP Altaïr, INRIA, Versailles, 1989.

[Parn72] D. L. Parnas
On the criteria to be used in decomposing systems into modules
Communication of the ACM, 15, 5, December 1972.

[Parn76] D. L. Parnas
Some hypothese about the "uses" hierarchy for operating systems
Technical Report, T. H. Darmstadt, Fachbereich Informatick, 1976.

[Peto90] I. Petoud
Génération automatique de l'interface homme-machine d'une application de geston hautement interactive
Thèse de doctorat, Université de Lausanne, Mai 1990.

[Pfaf85] *User Interface Management Systems*
G. E. Pfaff ed., Eurographics Seminars, Springer-Verlag, 1985.

[Pier91] G. Pierra
Les bases de la Programmation et du Génie Logiciel
DUNOD Informatique, 1991.

[PM88] J. Peckham, F. Maryanski
Semantic Data Models
ACM Computing Surveys, Vol. 20, N° 3, September 1988.

[PSW76] D-L. Parnas, J-E. Shore, D. Weiss
Abstract data types defined as classes of variables
Proc. Conf. on Data Abstraction and Definition, SIGMOD FDT, March 1976.

[Pudd92] N. Pudda
Un Langage Graphique et un Atelier pour la Conception d'Applications Interactives orientées Bases d'Objets
Note Technique Aristote, NOT 012, LGI-BULL-IMAG, Grenoble, Octobre 1992.

[PV91] C. Proix, M. Vinesse
Un modèle, une démarche et un outil pour l'analyse par objets
4èmes Journées Inter. «Le génie logiciel et ses applications», Toulouse 1991.

[RBP*91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen
Object-Oriented Modelling and Design
Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[RFB88] C. Rolland, O. Foucaut, G. Benci
Conception des systèmes d'information : La méthode REMORA
Eyrolles, 1988.

[Rich89] P. Richard
Des objets complexes aux bases de données orienté-objet
Thèse de Doctorat, Université de Paris-Sud, Octobre 1989.

[Rigu87] E. Riguidel
La méthode MAC-ADAM
Thomson CSF/SDC Direction Technique, Octobre 1987.

[Roch91] A. Rochfeld
Modèle externe de données et modèle externe objet
4èmes Journées Internationales «Le génie logiciel et ses applications», Toulouse 1991.

[Roll86] C. Rolland
Introduction à la Conception de Systèmes d'Information et panorama des Méthodes disponibles
Génie Logiciel, N°4, 1986.

[Roll92] C. Rolland
Outils CASE : Etat de l'art et perspectives
Journées d'étude «Bases de Données et Génie Logiciel», MASI, Université Pierre et Marie Curie, Paris, Février 1992.

[RX91] Rank Xerox
GraphTalk Métamodélisation
Manuel de Référence, Version 2.3, Septembre 1991.

[Rose87] J-P. Rosen
Méthode de Conception Orientée Objet
Génie Logiciel, N° 9, Novembre 1987.

[Sale84] A. M. Sales
Types abstraits et bases de données
Thèse de Doctorat, USTMG, Grenoble, Avril 1984.

[SFL83] J-M. Smith, S. Fox, T. Landers
ADAPLEX : Rationale and Reference Manual
Four Cambridge Center, Cambridge, Massachusetts, May 1983.

[Ship81] D. Shipman
The Functional Data Model and the Data Language ADAPLEX
ACM Transactions on Database Systems, 6(1), March, 1981.

[SM80] J. Smith, M. Mall
Pascal/R Report
Rapport technique N°66, Fachbereich Informatik, Université de Hambourg, Juillet 1980.

[SM88] S. Schlaer, S-J. Mellor
Object-Oriented Systems Analysis : Modelling the world in Data
Yourdon Press, 1988.

[Spiv89] J.M. Spivey
The Z Notation : A Reference Manual
Prentice-Hall, 1989.

[SS77] J.M. Smith, D.C.P. Smith
Database abstraction : Aggregation and generalisation
ACM Transaction on Databases System, 2, March, 1977.

[Stra88] Strategor
Stratégie, structure, décision, identité - Politique générale d'entreprise
InterEditions, Paris, 1988.

[Stro86] B. Stroustrup
The C++ programming language
Ed. Addison Wesley, 1986.

[Tesl85] L. Tesler
Object Pascal Report
Structured Language Word, Springer Verlag, New-York, 1985.

[TB85] P. Tanner, W. Buxton
Some Issues in Future User Interface Management Systems (UIMS) Development
User Interface Management Systems, Springer-Verlag, Berlin, 1985.

[THO85] THOMSON
Guide du Concepteur MACH
THOMSON-CSF Division Systèmes Electroniques, 1985.

[TRC85] H. Tardieu, A. Rochfeld, R. Colletti
La méthode MERISE - Tome 1 : Principes et outils
Les Editions d'Organisation, 1985.

[Warn81] J-D. Warnier
Logical Construction of Systems
Van Nostrand Reinhold, 1981.

[Wegn87] P. Wegner
Dimensions of Object-Based Language Design
OOPSLA'87 Proceedings, October 1987.

[Wirt71a] N. Wirth
Program Development by Stepwise Refinement
Communication of the ACM, 14, 4, 1971.

[Wirt71b] N. Wirth
The programming language Pascal
Acta Informatica, 1, 1, 1971.

[Wirt82] N. Wirth
Programming in MODULA-2
Springer-Verlag, 1982.

[WWW90] R. Wirfs-Brock, B. Wilkerson, L. Wiener
Designing Object-Oriented Software
Prentice Hall, Inc., 1990.

[YC79] E. Yourdon, L-L. Constantine
Structured Design
Prentice Hall, 1979.

Annexe A

Types abstraits de données, sous-typage et héritage

Apparus au milieu des années 70 les types abstraits de données [Gutt77, LZ74, LZ75, PSW76, BJ79] constituent une généralisation des types de base des langages de programmation. Le type abstrait de données est un mécanisme d'abstraction par lequel on s'intéresse à la sémantique du type que l'on spécifie de façon formelle et déclarative. Les éléments sont considérés de manière abstraite et on se donne une notation pour les décrire ainsi que l'ensemble des opérations qu'on peut leur appliquer et les propriétés de ces opérations.

1 Eléments de base

Un **type abstrait de données** (TAD) définit les opérations applicables aux éléments de ce TAD et les propriétés de ces opérations sans aucune référence à une technique de réalisation particulière des opérations et des éléments qui sont considérés de manière abstraite. Plus précisément, un TAD est défini par : le **nom** du type, les noms des TAD importés, un ensemble d'**opérateurs** permettant de manipuler les éléments du TAD et un ensemble d'**axiomes** pour décrire les propriétés de ces opérateurs par des expressions logiques. Les éléments d'un TAD sont manipulés via des variables comme dans les langages classiques de programmation.

Le nom et les opérateurs d'un TAD définissent l'aspect syntaxique ou signature du type alors que les axiomes décrivent la sémantique du type. Les arguments et les résultats des opérations sont typés et ces opérations se répartissent en trois catégories : un **constructeur** est un opérateur dont le résultat est un élément du TAD considéré construit à partir des arguments, un **accès** est une fonction de consultation retournant un résultat et dont un des arguments est un élément du TAD considéré et un **transformateur** est un opérateur de modification dont le résultat et au moins un des arguments sont du TAD considéré. Quand le domaine de définition d'un opérateur est un sous-ensemble du domaine complet, une précondition peut être associée à cet opérateur.

```
type naturel
importe booléen
opérateurs
   constructeurs    0 : → naturel
                    1 : → naturel
```

```
   accès          pair : naturel → booléen

   transformateur + : naturel × naturel  → naturel

axiomes pair(0) = vrai
        pair(1) = faux
        pair(x + y) = (pair(x) et pair(y)) ou (non pair(x)  et non pair(y))

        0 + x = x
        x + 0 = x
        x + y = y + x
        (x + y) + z = x + (y + z)
fin type naturel
```

Exemple A-1 : Type abstrait de données décrivant le domaine des nombres naturels.

L'approche «types abstraits» appelle deux remarques importantes [Jacq78] :

- La première difficulté avec les spécifications est de s'assurer que la spécification (les axiomes) décrivent bien l'ensemble de valeurs auquel on s'intéresse et rien que cet ensemble. La réponse à cette question est évidemment qu'il est impossible d'en avoir la certitude formelle ; il faudrait en effet disposer d'une spécification de la spécification pour laquelle se poserait le même problème.

- La seconde difficulté est de s'assurer que l'ensemble d'axiomes d'un type est complet et non contradictoire, or cette question est en général indécidable.

2 Types abstraits et sous-typage

On peut définir la relation de sous-typage entre deux types de la façon suivante [LP91] : *«le type T' est un sous-type du type T, noté T' ≤ T, si T' offre au moins le même comportement que T»*. Pour préciser les termes *«au moins le même comportement »*, les concepteurs du langage Emerald [ANSA89] proposent un ensemble de règles aujourd'hui largement admises qui peuvent être résumées par :

- T' offre au moins les mêmes opérations que T, et éventuellement des opérations supplémentaires.

- La liste de paramètres ($a_1 : T'_1$, $a_2 : T'_2$) est un sous-type de la liste de paramètres ($a_1 : T_1$, $a_2 : T_2$) si la proposition $\forall$ i, $T'_i \leq T_i$ est vérifiée.

- Si op est une opération de T, si a est la liste des arguments de op et si r est la liste des résultats de op alors op (a) → r ≤ op (a') → r' si et seulement si a' ≤ a et r ≤ r' où a' est la liste des arguments de op dans T' et r' la liste des résultats de op dans T'.

On remarquera que les arguments et les résultats évoluent en sens inverse vis-à-vis de la relation de sous-typage.

Lorsqu'un type T' est un sous-type d'un type T, un élément de T' est également un élément de T. Un élément de T' peut donc être utilisé à la place d'un élément de T et

par conséquence, le type des éléments de T' est à la fois T' et T. Cette possibilité d'attribuer, sous les contraintes exposées, plusieurs types à un élément est appelée polymorphisme d'inclusion [CW85].

3 Généricité dans les types abstraits

Le type abstrait générique (TAG) [BJ79] est à la fois une généralisation des constructeurs habituels des langages de programmation (ensemble, liste, etc.) et une généralisation des TAD eux-mêmes.

Les constructeurs offerts par les langages classiques pour définir des types concrets sont génériques au sens où ils sont paramétrés par d'autres types. Les fonctions d'accès ou de manipulations associées (accès à un champ, ajout d'un élément, etc.) sont prédéfinies dans le langage et elles sont génériques puisqu'elles travaillent sur des éléments de type variable (ajouter(t : T), retirer(t : T), etc.).

Certains TAD sont caractérisés, au type près, par les mêmes opérateurs et les mêmes axiomes. Ces TAD dénotent alors des familles de valeurs d'organisation similaire. Par exemple les types abstraits définissant des éléments de nature «ensemble d'entiers» ou «ensemble de caractères» ont, au type près, des opérations et des axiomes identiques (union, produit, intersection, etc.).

Un **type abstrait générique** (TAG) est un type abstrait paramétrable par un (ou plusieurs) type(s) abstrait(s) de données. Ce paramètre formel est inconnu lors de la définition du TAG. L'instanciation d'un type générique par un paramètre effectif (TAD) produit un nouveau TAD dont les opérations sont celles du TAG mais dont les types des arguments et des résultats sont totalement déterminés. Les types abstraits génériques constituent donc un mécanisme d'abstraction supplémentaire permettant la représentation d'une famille de types abstraits.

```
type V_3_(T)          (*vecteur de 3 éléments de type T inconnu*)
importe T
opérateurs
   constructeur    v3t : T × T × T → V_3_(T) (*construit un vecteur*)

   accès     c1 :  V_3_(T) → T (*accès au premier élément*)
             c2 :  V_3_(T) → T (*accès au deuxième élément*)
             c3 :  V_3_(T) → T (*accès au troisième élément*)

   transformateurs    ‡ : V_3_(T) × V_3_(T) → V_3_(T) (*addition*)
                      • : T × V_3_(T) → V_3_(T) (*produit*)

   axiomes  c1(v3t(x, y, z)) = x
            c2(v3t(x, y, z)) = y
            c3(v3t(x, y, z)) = z

            v3t(x, y, z) ‡ v3t(x', y', z') = v3t(x+x', y+y', z+z')
            (a ‡ b) = (b ‡ a)
            (a ‡ b) ‡ c = a ‡ (b ‡ c)

            u • v3t(x, y, z) = v3t(u*x, u*y, u*z)
            u • (a ‡ b) = ((u•a) ‡ (u•b)))
```

```
fin type V_3_(T)
```

Exemple A-2 : Type abstrait générique.

L'exemple A-3 montre le TAD `V_3_(entier)` obtenu par instanciation du TAG `V_3_ (T)` de l'exemple A-2 avec le TAD `entier`.

```
type V_3_(entier)  (*vecteur de 3 éléments de type T entier*)
importe entier
opérateur
   constructeur     v3t : entier × entier × entier → V_3_(entier)

   accès    c1 :  V_3_(entier) → entier
            c2 :  V_3_(entier) → entier
            c3 :  V_3_(entier) → entier

   transformateurs     ‡ : V_3_(entier) × V_3_(entier) → V_3_(entier)
                       • : entier × V_3_(entier) → V_3_(entier)

   axiome  ...
fin type V_3_(entier)
```

Exemple A-3 : Type abstrait instancié.

Le sous-typage et la généricité sont deux mécanismes de factorisation et de généralisation complémentaires [Meye88].

4 Propriétés sur les types abstraits de données

Les axiomes donnés dans un type abstrait générique utilisent différents opérateurs pour manipuler le paramètre formel, comme par exemple « = », « + », « ¬ » ou « > ». Ces opérateurs doivent donc être définis pour le TAD constituant le paramètre effectif lors de l'instanciation du TAG. Le paramètre formel d'un TAG ne peut donc pas être remplacé par n'importe quel TAD.

Dans l'exemple A-2, l'opérateur «‡» défini par «‡ : V_3_(T) × V_3_(T) → V_3_(T)» représente , d'après les axiomes, l'addition des vecteurs par addition rang à rang de leurs éléments. Dans l'axiome définissant «‡», l'opérateur d'addition «+» est utilisé sur des éléments du TAD T. Lors d'une instanciation du TAG V_3_(T) le type abstrait candidat pour remplacer T doit donc supporter l'opérateur «+». Le type «entier» par exemple est un bon candidat (l'addition des entiers est définie). Par contre, le TAD «personne» ne peut être accepté que si l'addition de personnes définie par : «+ : personne × personne → personne» existe dans le TAD «personne». De façon identique, la définition de l'opérateur multiplication «•» dans V_3_(T) requière la définition de l'opérateur «*» dans T.

Pour fixer les caractéristiques minimales des TAD acceptés pour instancier un type abstrait générique la notion de propriété a été introduite [Jacq78]. Une **propriété** est définie par : un nom de propriété, un ensemble d'opérateurs caractérisant cette propriété et un ensemble d'axiomes définissant ces opérateurs. Les opérations peuvent être ré-

parties en deux catégories : les opérations fondamentales qui sont les opérations minimales (et les axiomes associés) à supporter par le type pour qu'il vérifie la propriété et les opérations dérivées constituant des opérateurs facultatifs.

Un type abstrait générique peut exiger une propriété P sur le paramètre formel, P devra alors être vérifiée par les paramètres effectifs. On dit qu'un type abstrait T vérifie une propriété P si la définition de T contient les opérateurs de P et l'axiomatisation associée. Dans l'exemple A-2, le TAG V_3_(T) doit exiger que la propriété constituée par les deux opétations «+ et *» soit remplie par les TAD paramètres effectifs. On notera que la relation existant entre P et T est alors une relation de sous-typage : $T \leq P$.

On distingue un type abstrait, un type générique et une propriété :

- un type abstrait définit un ensemble de valeurs,
- un type abstrait générique produit par instanciation un ensemble de types abstraits (éventuellement génériques),
- une propriété caractérise une famille de types abstraits génériques ou une famille de types abstraits de données.

5 Extensions adaptées aux bases de données

Dans le contexte des bases de données les objets manipulés possèdent des caractéristiques particulières [Sale84] :

1 - l'utilisateur doit pouvoir représenter des objets abstraits images des objets réels de son univers,

2 - un même objet peut-être perçu selon plusieurs facettes ; chaque facette détermine les caractéristiques dont la connaissance est fondamentale dans un contexte donné ; la connaissance des caractéristiques des autres facettes est sans importance voire préjudiciable ou interdite dans ce contexte,

3 - un même objet peut appartenir à plusieurs ensembles ; il n'y a qu'une facette significative par ensemble,

4 - un objet peut-être associé avec d'autres objets de plusieurs manières différentes.

Les points 1 et 4 traduisent la notion de partage et les points 2 et 3 la notion de vue. Pour prendre en compte ces points particuliers, A.M. Sales propose d'introduire les notions de p-type et de multifacettage.

Un type abstrait de données permet de décrire un ensemble de valeurs qui sont par définition non partageables. Ces données sont accessibles par des variables et l'opération d'affectation entre deux variables entraîne une duplication de la valeur ce qui rend impossible le partage. La notion de p-type permet de décrire un ensemble d'éléments (objets) partageables pour lesquels l'opération d'affectation n'entraîne pas la duplication de la valeur mais la définition d'un nouvel accès à cette valeur. Comme un type abs-

trait, un p-type est défini par des opérateurs et une sémantique qui ont été étendus pour prendre en compte les aspects liés au partage et aux associations.

Les fonctions attributs sont des opérations dans lesquelles le paramètre est le p-type et dont le résultat est un type ou un p-type. La clé est une fonction particulière ou une composition de fonctions qui permet de distinguer deux occurrences du p-type.

L'expression de la sémantique d'un p-type se fait par des dépendances intrinsèques entre attributs et par des dépendances inter-objets pour les fonctions à résultats de nature p-type.

Le multifacettage permet la représentation de différentes facettes qui enrichissent un noyau de caractéristiques communes à toutes ces facettes.

6 Héritage, covariance et contravariance

6.1 Contraintes du sous-typage sur les méthodes

Dans les langages à objets, l'intérêt principal du sous-typage est de pouvoir substituer à un objet de classe C un objet plus spécialisé de classe C' si toutefois il existe entre C et C' une relation de sous-typage. Les vérifications de type doivent garantir que de telles substitutions n'engendrent pas d'erreur de type : nombre incorrect de paramètres ou types des paramètres incorrects dans une méthode, utilisation d'une méthode ou d'un attribut inexistant, etc. Dans le cadre du sous-typage, on peut distinguer deux grandes catégories de langages à objets : les systèmes contravariants et les systèmes covariants.

Les **systèmes contravariants** s'appuyent sur les règles présentées dans la section précédente et ils ne permettent pas de spécialiser les arguments des méthodes. Avec ces règles, la vérification statique du typage ne nécessite pas de recompilation. Comme les objets sont créés seulement à l'exécution, la substitution d'un objet de C par un objet de C' peut avoir lieu dynamiquement sans nécessiter de recompiler le programme et le sous-typage garantit que le nouvel objet sera capable de traiter l'ensemble des messages traités par le premier. Le respect du sous-typage garantit que la substitution n'introduit pas d'erreur de type.

En revanche, dans les **systèmes covariants** comme Eiffel ou O2, les arguments peuvent être spécialisés. Dans de tels systèmes, les vérifications de types peuvent également être réalisées statiquement mais au prix de différentes recompilations. En effet considérons la méthode m définie dans la classe C par $\underline{m\,(a : D) \rightarrow r}$ et surchargée dans la classe C' en $\underline{m\,(a : D') \rightarrow r}$ telle que D' est un sous-type de D, ce qui est permis par la covariance (cf. exemple A-4).

```
classe D
fin

classe C
   méthode m (a : D) → R
```

```
classe D'
     hérite D
fin

classe C'
     hérite C
```

```
fin                                          méthode m (a : D') → R
                                             fin
```

Exemple A-4 : Exemple d'héritage covariant.

Le message m (x) où x est de type D est correct pour la classe C mais il est incorrect pour C'. Vis-à-vis de ce message un objet de la classe C ne peut pas être remplacé par un objet de C'. Lors de la définition d'une nouvelle classe C' héritant d'une classe C il est donc nécessaire de vérifier à nouveau dans tous les clients de C et de ses sur-classes, que les messages envoyés peuvent être traités correctement par C'. Par ailleurs il est également nécessaire, pour tout client D de C', de vérifier que chaque message m envoyé à D est correctement traité par la classe D et par toutes les sous-classes de D ayant redéfini m. On constate donc que la covariance impose de nombreuses vérifications et qu'il est également nécessaire de recompiler les surclasses si celles-ci utilisent les méthodes redéfinies.

6.2 Contraintes du sous-typage sur les attributs

Les règles d'évolution des attributs dépendent des systèmes. Cependant, on peut remarquer qu'un attribut ne devrait pas pouvoir être spécialisé ni généralisé. En effet, il faut considérer un attribut à la fois comme le résultat d'une fonction (en consultation) et comme l'argument d'une méthode (en affectation/modification). D'après les règles de contravariance, un attribut devrait donc pouvoir être à la fois généralisé et spécialisé. Cependant, aux prix de vérifications supplémentaires mais limitées à la classe dans laquelle est défini l'attribut et à ses sur-classes, un attribut privé peut être généralisé ou spécialisé et un attribut public en lecture seule peut être spécialisé.

D'après les règles de covariance, comme les arguments et les résultats peuvent être spécialisés, un attribut doit pouvoir être spécialisé.

6.3 Héritage et constructeurs

6.3.1 Systèmes à objets

Les constructeurs peuvent être perçus comme des classes à part entière et générant des objets.

Dans ce cas, si les règles de contravariance sont respectées, le ou les paramètres d'un constructeur ne doivet pas être modifiés. En effet ils apparaîssent à la fois en arguments et en résultats de méthodes (méthodes d'insertion ou d'affectation et méthodes d'extraction ou de lecture). Comme pour les attributs, le type effectif d'un constructeur ne peut être ni spécialisé ni généralisé. De nouvelles méthodes ou de nouveaux attributs peuvent être ajoutés, par exemple : nuplet ($a_1 : T_1$, $a_2 : T_2$,..., $a_n : T_n$) ≤ nuplet ($a_1 : T_1$, $a_2 : T_2$).

Les règles de covariance appliquées aux constructeurs lorsque ces derniers sont des classes à part entière permettent de spécialiser le paramètre effectif utilisé dans un constructeur. Dans ce cas les règles sont les suivantes [DLR91] :

- nuplet ($a_1 : T'_1$, $a_2 : T'_2$) ≤ nuplet ($a_1 : T_1$, $a_2 : T_2$) ⇔ ∀ i, $T'_i \leq T_i$.
- nuplet ($a_1 : T_1$, $a_2 : T_2$,..., $a_n : T_n$) ≤ nuplet ($a_1 : T_1$, $a_2 : T_2$).
- liste (T') ≤ liste (T) ⇔ T' ≤ T.
- ensemble (T') ≤ ensemble (T) ⇔ T' ≤ T.

6.3.2 Systèmes à valeurs

Lorsque les constructeurs sont des constructeurs de valeurs, dans une affectation telle que «`identif_1 := identif_2`», l'information contenue dans `identif_1` est remplacée par une copie de l'information contenue dans `identif_2`, car les valeurs ne sont pas partageables

Si `identif_1` est un identificateur de type `constructeur(T)` (`constructeur` ∈ {`ensemble, liste, nuplet`, etc.}) et si `identif_2` est un identificateur de type `constructeur(T')` avec `T'` ≤ `T` alors, dans l'affectation `identif_1 := identif_2`, la valeur contenue dans `identif_2`, de type `constructeur(T')`, peut être recopiée dans une valeur de type `constructeur(T)` puisque `T'` ≤ `T`. Le type de `identif_1` est donc toujours `constructeur(T)` et les messages acceptés par `identif_1` avant l'affectation sont également acceptés après l'affectation.

Dans tous les cas, les règles de la section 6.3.1 pour des systèmes covariants peuvent garantir un typage fort sans nécessité de recompilation.

Annexe B

Langages et modèle MOSAÏC

1 Langage textuel

Pour présenter la syntaxe du langage textuel de MOSAÏC, nous avons retenu les conventions suivantes [Wirt82] :

- [A] signifie "au plus un A",
- { A } signifie "un nombre quelconque de A",
- A | B signifie "A ou B",
- " " (les guillemets) encadrent certains symboles du langage,
- identif_... dénote un identificateur.

Définition des classes :

```
classe_applicative      = classe_applicative identif_classe_applicative
                              [ partie_héritage ]
                              { facette }
                              [ liste_exportation ]
                              [ partie_structurelle ]
                              [ partie_fonctionnelle_applicative ]
                              [ partie_dynamique ]
                          fin

facette                 = facette identif_facette ":" identif_classe_facette

classe_facette          = classe facette identif_classe_facette
                              [ partie_héritage ]
                              [ liste_exportation ]
                              [ partie_structurelle_facette ]
                              [ partie_fonctionnelle_applicative ]
                              [ partie_dynamique ]
                          fin
```

classe_interactive = **classe interactive** identif_classe_interactive
[partie_héritage]
[facette_présentation]
[liste_exportation]
[partie_structurelle]
[partie_fonctionnelle_interactive]
[partie_dynamiquee]
fin

facette_présentation = **facette présentation** ":" identif_classe_présentation
| **facette présentation** corps_facette_présentation **fin**

classe_facette_présentation = **classe facette présentation** identif_classe_présentation
corps_facette_présentation
fin

corps_facette_présentation = [partie_héritage]
[partie_structurelle]
[partie_fonctionnelle_présentation]
[partie_dynamique]

classe_interrogation = **classe interrogation** identif_classe_interrogation
[partie_fonctionnelle_interrogation]
fin

Définition des classes projets:

classe_projet = **classe projet** identif_classe_projet
[racines_de_persistance]
[racines_d'application]
fin

racines_de_persistance = **racine de persistance**
{ identif_racine ":" **réf** | **prop** identif_classe_applicative }

racines_d'application = **racine d'application**
{ identif_application ":" identif_classe_applicative }

Définition des parties structurelles :

partie_structurelle = **attribut** { attribut | attribut_simple }

partie_structurelle_facette = **attribut** { attribut_simple }

attribut = identif_attribut ":" type_attribut

type_attribut = type_attribut_simple

| association_binaire

attribut_simple = identif_attribut ":" type_attribut_simple

type_attribut_simple = classe_de_base
| classe_énum
| construction

association_binaire = nature_attribut identif_classe [réciproque]

classe_de_base = Entier | Réel | Booléen | Caractère | Chaîne

classe_énum = classe_de_base "{" constante { "," constante} "}"

nature_attribut = **réf** | **dép** | **prop**

réciproque = **réf** | **dép**
identif_attribut

construction = construction_ensemble
| construction_liste
| construction_table
| construction_nuplet

construction_ensemble = Ensemble "(" param_construct ")"

construction_liste = Liste "(" param_construct ")"

construction_table = Table "(" entier ".." entier param_construct ")"

construction_nuplet = Nuplet "(" attribut_simple { "," attribut_simple } ")"

param_construct = nature_attribut identif_classe
| construction

Définition de la partie fonctionnelle :

partie_fonctionnelle_applicative = **variable** { variable }
[dépendance]
{ méthode_applicative }

variable = identificateur ":" type_variable

type_variable = type_de_base
| nature_attribut identif_classe
| construction

dépendance = **utilise** { identif_classe }

méthode_applicative = **méthode primaire** | **méthode secondaire**

identif_méthode
liste_paramètres
[assertions]
[**persistante** | **temporaire**]
[**message** { message}]
[**variable** { variable }]
[**début** algorithme]
fin

partie_fonctionnelle_interactive = **variable** { variable }
[dépendance]
{ méthode_interactive }

méthode_interactive = **méthode** identif_méthode
liste_paramètres
[assertions]
[**message** { message }]
[**variable** { variable }]
[**début** algorithme]
fin

partie_fonctionnelle_interrogation= { méthode_interactive }

liste_paramètres = ["(" { argument } ")"] [":" identif_classe]

argument = identif_argument ":" identif_classe

assertions = **précondition** proposition_logique
postcondition proposition_logique

message = identif_classe "•" nom_caractéristique

nom_caractéristique = identif_méthode
| identif_attribut { "•" identif_attribut }
| identif_variable { "•" identif_attribut }.

Définition de la partie dynamique :

partie_dynamique = état invariant

état = **état** [**initial** proposition_logique]
{ identif_état "=" proposition_logique}

invariant = **invariant** { règle_d'intégrité }

règle_d'intégrité = [identif_règle ":"] proposition_logique

Définition de la liste d'exportation :

liste_exportation = **exporte** [liste_fonction] [liste_méthode] [liste_état]

liste_fonction = **fonction** { identif_fonction }

identif_fonction = identif_attribut | identif_variable | identif_méthode

liste_méthode = **méthode** { nom_méthode }

liste_état = **état** { nom_état }

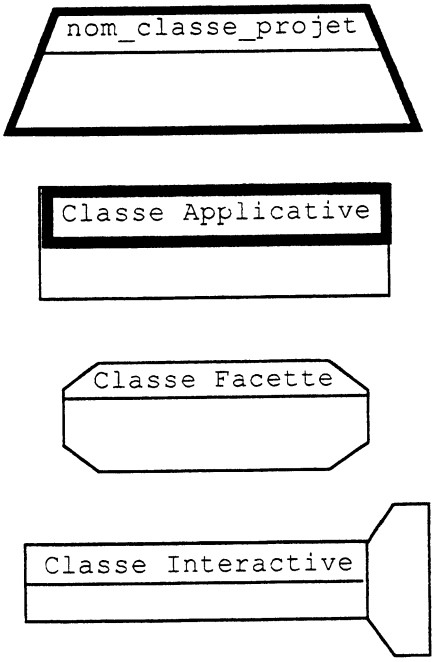
Définition de la partie héritage :

partie_héritage = { nature_héritage identif_classe }

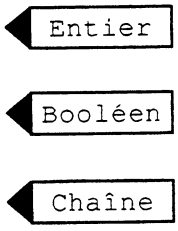
nature_héritage = **adapte** | **est-un**

2 Langage graphique

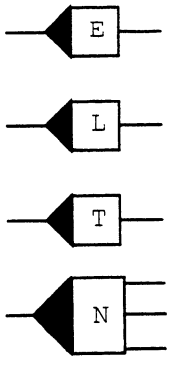
Représentation des classes :



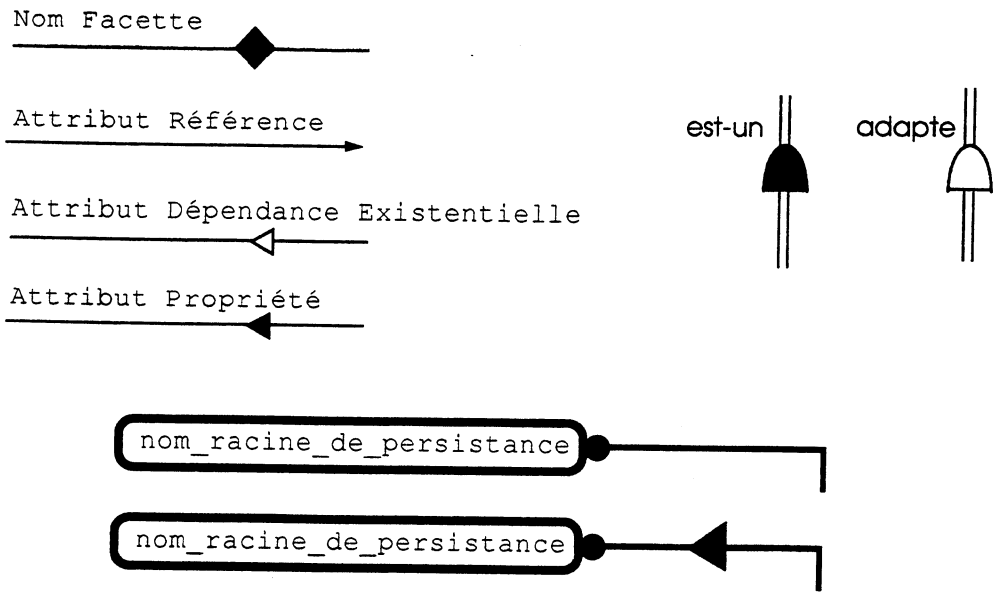
Classes de base :



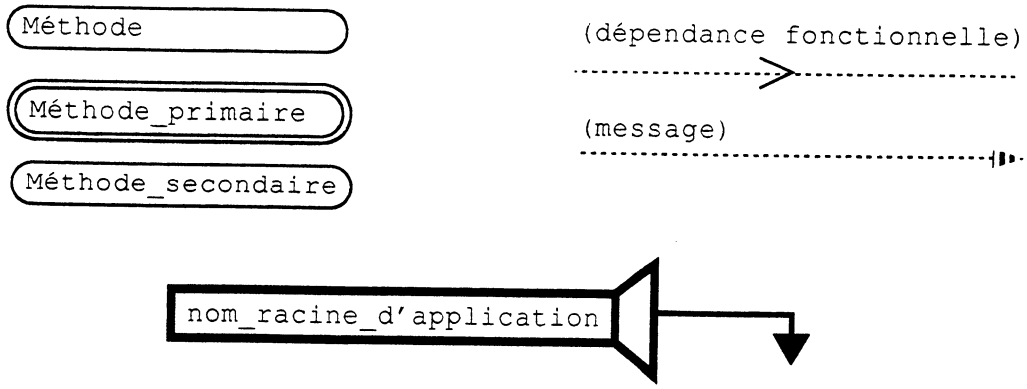
Classes constructeurs :



Représentation des aspects structurels et des héritages :



Représentation des aspects fonctionnels :



3 Modèle documentaire

Définition des classes projets :

Classe_projet

<-(nom)-/-> Chaîne
<-(persistances)-/->> Racine_de_persistance
<-(applications)-/->> Racine_d'application

Racine_de_persistance

<-(nom)-/-> Chaîne
<<-(classe)-/-> {réf, prop} × Classe_applicative_gén

Classe_applicative_gén

= Classe_applicative ∪ Constructeur_applicatif ∪ Classe_de_base

Racine_d'application

<-(nom)-/-> Chaîne
<<-(classe)-/-> Classe_initiale

Classe_initiale

= {C ∈ Classe_interactive, ∃ m ∈ caractéristique(C) ∧ m ∈ Méthode_applicative ∧ nom(m) = "lancer_application"}

Définition des classes applicatives :

Classe_applicative

<-(nom)-/-> Chaîne
<-(signatures)-/->> Caractéristique_applicative
<<-(sous-types)-/->> Classe_applicative
<<-(sous-classes)-/->> Classe_applicative
<-(facettes)-/->> Classe_facette
<-(dépendances)-/->> Classe_applicative ∪ Classe_d'interrogation
<-(caractéristiques)-/->> Caractéristique_applicative
<-(invariant)-/-> Invariant

Classe_facette

<-(nom)-/-> Chaîne
<-(signatures)-/->> Caractéristique_applicative
<<-(sous-types)-/->> Classe_facette
<<-(sous-classes)-/->> Classe_facette
<-(dépendances)-/->> Classe_applicative ∪ Classe_d'interrogation
<-(caractéristiques)-/->> Caractéristique_applicative

invariant
<——————/-> Invariant

Caractéristique_applicative = Attribut_applicatif ∪ Variable_applicative ∪ Méthode_applicative ∪ Etat_applicatif

Définition des attributs et des constructeurs des classes applicatives :

Attribut_applicatif

nom
<-/——————> Chaîne

nature
<<-/——————> {réf, dép, prop}

classe
<<-/——————> Classe_applicative_gén

Constructeur_applicatif = Nuplet_applicatif ∪ Ensemble_applicatif ∪ Liste_applicative ∪ Table_applicative

Nuplet_applicatif

constituants
<——————>> Attribut_applicatif

Ensemble_applicatif

paramètre
<<-/——————> {réf, dép, prop} × Classe_applicative_gén

Liste_applicative

paramètre
<<-/——————> {réf, dép, prop} × Classe_applicative_gén

Table_applicative

bornes (2,2)
<<-/——————>> Entier X Entier

paramètre
<<-/——————> {réf, dép, prop} × Classe_applicative_gén

Définition des méthodes des classes applicatives :

Méthode_applicative

nom
<-/——————> Chaîne

arguments
<<-/——————/->> {réf, dép, prop} × (Classe_applicative_gén ∪ Classe_d'interrogation)

résultat
<<-/——————/-> Classe_applicative_gén

nature
<<-/——————> {primaire, secondaire} × {persistant, temporaire}

précondition
<<-/——————/-> Proposition_logique

postcondition
<<-/——————/-> Proposition_logique

messages
<<-/——————/->> (Classe_applicative × Caractéristique_applicative) ∪ (Classe_d'interrogation × Méthode_interactive)

algorithme
<<-/——————/-> Texte

Variable_applicative

<-nom-/-> Chaîne
<<-nature-/-> {réf, dép, prop}
<<-classe-/-> Classe_applicative_gén

Définition des états et des invariants :

Etat_applicatif

<-nom-/-> Chaîne
<<-valeur-/-> Proposition_logique

Invariant

<<-règles_d'intégrité-/->> Proposition_logique

Définition des classes interactives :

Classe_interactive

<-nom-/-> Chaîne
<-signatures-/->> Caractéristique_interactive
<<-sous-types-/->> Classe_interactive ∪ Classe_d'interrogation
<<-sous-classes-/->> Classe_interactive
<-présentation-/-> Facette_présentation
<-dépendances-/->> Classe_interactive ∪ Classe_applicative
<-caractéristiques-/->> Caractéristique_interative
<-invariant-/-> Invariant

Caractéristique_interactive = Attribut_interactif ∪ Variable_interactive ∪ Méthode_interactive ∪ Etat_interactif

Facette_présentation

<<-sous-types-/->> Facette_présentation
<<-sous-classes-/->> Facette_présentation
<-caractéristiques-/->> Caractéristique_présentation
<-invariant-/-> Invariant

Caractéristique_présentation = Attribut_présentation ∪ Méthode_présentation

Classe_d'interrogation

<-nom-/-> Chaîne
<-signatures-/->> Caractéristique_interactive

Définition des attributs et des constructeurs des classes interactives :

Attribut_interactif
- nom : <-/-> Chaîne
- nature : <<-/-> {réf, dép, prop}
- classe : <<-/-> Classe_interactive_gén

Classe_interactive_gén = Classe_interactive ∪ Constructeur_interactif ∪ Classe_applicative_gén

Constructeur_interactif = Nuplet_interactif ∪ Ensemble_interactif ∪ Liste_interactive ∪ Table_interactive

Nuplet_interactif
- constituants : <-->> Attribut_interactif

Ensemble_interactif
- paramètre : <<-/-> {réf, dép, prop} × Classe_interactive_gén

Liste_interactive
- paramètre : <<-/-> {réf, dép, prop} × Classe_interactive_gén

Table_interactive
- bornes (2,2) : <<-/-> Entier X Entier
- paramètre : <<-/-> {réf, dép, prop} × Classe_interactive_gén

Attribut_présentation
- nom : <-/-> Chaîne
- classe : <<-/-> Classe_présentation_gén
- commentaire : <<-/-/-> Chaîne

Classe_présentation_gén = Constructeur_présentation ∪ Classe_de_base

Constructeur_présentation = Nuplet_présentation ∪ Ensemble_présentation ∪ Liste_présentation ∪ Table_présentation

Nuplet_présentation
- constituants : <-->> Attribut_présentation

Ensemble_présentation
- paramètre : <<-/-> Classe_présentation_gén

Liste_présentation
- paramètre : <<-/-> Classe_présentation_gén

Table_présentation
- bornes (2,2) : <<-/-> Entier × Entier

Annexe C

Application «Gestion de Compétitions Sportives»

1 Description générale

Cette annexe a pour objectif de décrire les spécifications d'un système informatique ainsi que le contexte dans lequel sa mise en place est envisagée. Le domaine d'intérêt est celui de l'organisation de compétitions sportives en canoë-kayak. Le but est d'automatiser le suivi d'une manifestation selon les aspects sportifs exclusivement. Cette application, orientée gestion, est caractérisée par un grand nombre d'objets répartis dans des catégories différentes et par des possibilités d'extensions variées : complexification/généralisation de l'application de base, utilisation/réutilisation d'applications existantes, prise en compte d'aspects temps réel, etc.

1.1 Déroulement général

Une manifestation permet à des personnes de se mesurer à travers une activité sportive, le canoë-kayak, sur un lieu et à une date donnés. Son organisation est à la charge d'un comité d'organisation (CO) et nous décrivons le déroulement général actuel d'une compétition. La section 2 donne une spécification précise des entités manipulées.

La phase d'initialisation de la manifestation permet d'initialiser les différents documents par le nom et la date de la manifestation. Les catégories autorisées à concourir dans la compétition sont fixées dans cette phase et elles sont choisies parmi un ensemble de catégories de référence. Une catégorie de référence est identifiée par un libellé et fixe différentes caractéristiques : le type d'embarcation, le nombre d'équipiers, le sexe et l'âge. Le nombre d'équipiers peut être de 1 (embarcation mono) ou de 2 (embarcation bi).

La phase d'inscription permet aux licenciés souhaitant participer à la compétition de se faire connaître en contractant des inscriptions auprès du CO. Un licencié, appelé aussi athlète ou coureur dans la suite, est une personne ayant souscrit une licence dans un club. Une inscription est un contrat établi entre une catégorie et une embarcation composée d'un ou de plusieurs licenciés en fonction du nombre d'équipiers dans l'embarcation. Les caractéristiques de ces licenciés et de l'embarcation doivent être compatibles avec les caractéristiques de la catégorie (âge, sexe et nombre d'équipiers). Un licencié peut contracter plusieurs inscriptions dans des catégories différentes mais une seule

par catégorie.

La phase d'attribution des dossards permet au comité d'organisation, à l'issue de la phase d'inscription, d'attribuer à chaque inscription un numéro entier, appelé dossard, permettant de l'identifier. L'ordre des dossards détermine l'ordre de départ pour le concours. La liste de départ (liste des dossards) est communiquée aux athlètes et les dossards sont distribués.

Le déroulement du concours est supervisé par le comité d'organisation et se déroule en deux manches sur un même parcours pour tous les athlète. Le parcours d'environ 600 m est jalonné d'obstacles artificiels. Il est divisé en secteurs en fonction de la répartition de ces obstacles. Chaque secteur est contrôlé par un juge qui sanctionne les éventuelles pénalités réalisées par les coureurs sur les obstacles. La prestation de chaque inscription (ou embarcation) tient compte à la fois de sa rapidité (temps de parcours total) et de son adresse (total des pénalités réalisées sur chaque secteur). Pour chaque manche la prestation d'une embarcation est définie comme le total de son temps (converti en secondes) et de ses pénalités. Les pénalités sont jugées sur le lieu de déroulement de l'épreuve et transmises par téléphone au fur et à mesure de la progression de l'embarcation sur le parcours au CO qui centralise l'ensemble des résultats. Quand un coureur franchit la ligne d'arrivée, le chronométreur détermine son temps et le transmet au CO. Toutes les inscriptions réalisent une première prestation (première manche) dans l'ordre des dossards. A l'issue de cette première manche le CO établit et publie un classement provisoire des inscriptions (ou embarcations) réalisé sur le premier total. Tous les athlètes réalisent une seconde prestation dans l'ordre des dossards (seconde manche). A l'issue de cette seconde manche le CO publie un classement provisoire établi sur le meilleur des deux totaux réalisés par chaque inscription lors de ses deux prestations. Certains athlètes peuvent être autorisés à recourir une (ou les deux) manche(s) ; le nouveau résultat remplace celui de la manche recourue.

La phase de proclamation des résultats marque la clôture du concours (les manches litigieuses ayant été recourues), les classements officiels et définitifs sont publiés. Tous les classements sont établis par catégorie mais un classement général est quelquefois demandé.

1.2 Objectifs du projet

1.2.1 Motivations

Les remarques suivantes sont à l'origine de ce projet :

- la charge de travail est importante pour le comité d'organisation qui souhaite être assisté par un outil pour les travaux automatisables (affectation des dossards, édition des listes, classement des athlètes, etc.),
- l'organisation est trop rigide : les phases sont nécessairement séquentielles, on souhaite pouvoir l'assouplir (prise en compte des retardataires, corrections, etc.),
- des traitements actuellement réalisés en différé, comme les classements, seraient avantageusement remplacés par des traitements en temps réel, pour améliorer le retour d'informations vers les athlètes,

- il serait intéressant d'établir des classements intermédiaires et provisoires tout au long du déroulement du concours pour le rendre plus «vivant»,
- il existe une base de données contenant les licenciés, les clubs et les catégories de référence, il serait souhaitable d'exploiter ces informations.

1.2.2 Cohérence de l'enchaînement des étapes

Le déroulement de la manifestation impose un ordre général mais non strict d'enchaînement des différentes étapes :

initialisation → inscription → dossards → concours → résultats.

L'application doit assurer la cohérence des opérations proposées à l'utilisateur lors des différentes étapes. Notamment les opérations «non encore réalisables» (en avance) à une étape donnée ou les opérations «déjà plus réalisables» (en retard) ne doivent pas être activables. Par exemple, juste après l'étape d'affectation des dossards, les opérations liées à la proclamation des résultats ne doivent pas encore être accessibles et toute inscription doit être obligatoirement complétée par l'affectation d'un dossard à cette inscription.

L'ordre ne doit pas être rigide et un retour à une étape antérieure doit toujours être possible (l'ordre n'est pas strict). Par exemple les inscriptions d'une course peuvent être interrompue provisoirement pour permettre une simulation de la course (affectation dossard et concours simulés) à la suite de quoi, un retour à l'étape d'inscription pour compléter les inscriptions doit être possible.

1.2.3 Automatisation des tâches

On souhaite pouvoir gérer successivement plusieurs compétitions sans archivage des courses antérieures. La phase d'initialisation doit donc assurer en cas de nouvelle course, l'annulation de la précédente et l'initialisation de la nouvelle. L'annulation d'une course annule les inscriptions, les dossards et les résultats relatifs à la course courante. En revanche, les informations relatives aux licenciés, aux clubs et aux catégories de référence ne sont pas affectées.

Le système doit assurer l'édition des listes suivantes : licenciés, clubs, catégories, inscrits, dossards, classements partiels et résultats définitifs.

L'affectation (ou attribution) des dossards est réalisée pour l'ensemble des inscriptions de la course et doit pouvoir être effectuée automatiquement ou manuellement.

Le calcul du total d'une manche (temps + pénalités) pour un athlète et son classement dans sa catégorie pendant le déroulement du concours doit être effectué automatiquement au fur et à mesure de la saisie des performances des coureurs.

1.2.4 Assouplissement du déroulement

Les procédures de saisie d'informations doivent être souples. Par exemple, la saisie des inscriptions doit pouvoir se faire par individu ou par catégorie.

Comme on l'a mentionné ci-dessus le système doit autoriser des modifications concernant les opérations de phases antérieures (inscription d'un retardataire alors que la course est lancée ...) ou le retour à des phases antérieures (annulation des dossards ...).

Le système doit pouvoir être arrêté et redémarré sans perte d'informations. Par ailleurs le système peut être amené à fonctionner dans un environnement hostile avec notamment des coupures de courant fréquentes. Les pertes d'information doivent être aussi limitées que possible.

1.2.5 Informations pour les athlètes

Un aspect important du déroulement du concours est la rediffusion aux athlètes de leurs performances et de leurs classements, ces informations sont centralisées et détenues par le CO. Dans ce but, des écrans reproduisent l'écran de l'opérateur en cours de travail. Cet écran doit donc contenir des informations destinées aux athlètes, entre autre :

- les informations concernant le dossard en cours de saisie : nom, prénom et club, sa place actuelle puis les informations relatives à la manche en cours de saisie : manche, temps, pénalités, total, nouvelle place,
- les informations concernant les athlètes arrivés précédemment,
- l'affichage permanent du classement d'une catégorie au choix de l'opérateur,
- des impressions de classements provisoires doivent pouvoir être effectuées à tout moment.

La cohérence des informations affichées doit être maintenue en permanence.

1.3 Utilisateurs

L'application est destinée à être utilisée par des néophytes après une formation succinte, elle doit donc posséder des qualités de robustesse, de souplesse et de convivialité.

1.4 Différents états de la course

Le déroulement général de la course peut être décrit par un automate à cinq états :

Les flèches pleines indiquent les transitions normales (dans le sens du déroulement habituel), les flèches grises indiquent les retours arrières dans le déroulement. Ce schéma représente l'automate de l'organisation de la course et les flèches montrent les transitions possibles entre ces états mais ne préjugent en rien des opérations autorisées dans chacun des états.

- Re-initialisé : la course précédente a été annulée, les informations générales de la course courante ont été saisies (nom, date etc...) mais aucune inscription n'a encore été réalisée,
- Inscriptions réalisées : des inscriptions ont été effectuées mais les dossards n'ont pas été attribués,

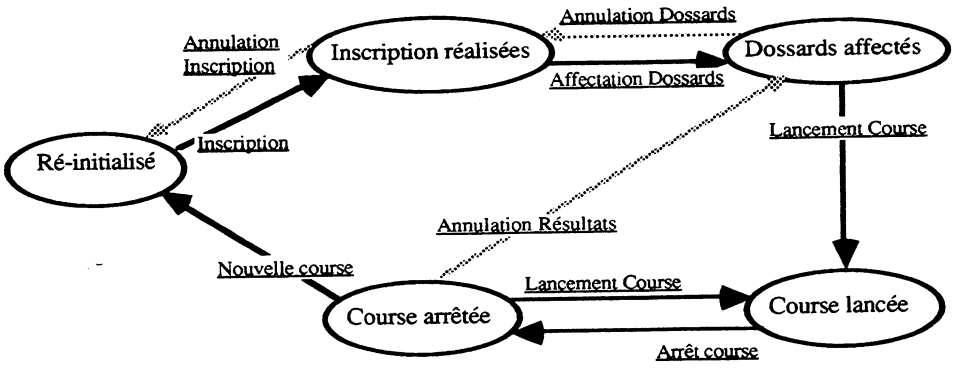


Figure C-1 : Automate de la course.

- Dossards affectés : les dossards ont été attribués mais aucun résultat n'a été saisi,
- Course lancée : la course est en train de se dérouler et des résultats ont été et sont saisis,
- Course arrêtée : des résultats ont été saisis mais la course est momentanément ou définitivement arrêtée.

2 Spécification des données

Dans cette section, nous fixons la terminologie du domaine et la description des données manipulées à l'aide du formalisme Z. On notera que ces données ne sont pas toutes persistantes notamment, les informations déduites d'autres informations peuvent être calculées ou stockées temporairement.

2.1 Domaines de base

Les données décrites dans cette section ne sont pas liées à une compétition particulière et persistent tout au long d'une saison de compétitions.

```
Age ⊂ Chaîne
Age = {Minime, Cadet, Junior, Senior}

Embarcation ⊂ Chaîne
Embarcation = {Kayak, Canoë}

Equipage ⊂ Chaîne
Equipage = {Mono, Bi} (*Mono = 1 personne, Bi = 2 personnes*)

Sexe ⊂ Chaîne
Sexe = {H, F}

Club ⊂ Chaîne
```

Un `Licencié` est une personne ayant adhéré à un club et pouvant à ce titre participer à une compétition.

```
                    numéro
Licencié          <-/---------------->      Entier
```

```
                    nom
                <<-/---------------->      Chaîne
                    prénom
                <<-/---------------->      Chaîne
                    sexe
                <<-/---------------->      Sexe
                    âge
                <<-/---------------->      Age
                    club
                <<-/---------------->      Club
```

Une `Catégorie_réf` (catégorie de référence) décrit une catégorie générique caractérisée par un nom et un ensemble de propriétés.

```
Catégorie_réf       libellé
                <-/---------------->       Chaîne
                    acronyme
                <-/---------------->       Chaîne
                    embarcation
                <<-/---------------->      Embarcation
                    équipage
                <<-/---------------->      Equipage
                    sexe
                <<-/---------------->      Sexe
                    âge
                <<-/---------------->      Age
```

On dispose au moins des catégories : {Kayak Homme, Canoë Mono Homme, Kayak Dame, Canoë Bi Homme} déclinées dans les catégories d'âge : {Senior, Junior, Cadet}.

2.2 Données propres à chaque compétition

Les informations décrites dans cette section sont propres à chaque compétition.

```
Compétition         nom
                <<-/---------------->      Chaîne
                    date
                <<-/---------------->      Date
                    lieu
                <<-/---------------->      Chaîne
                    catégories
                <<-/------------/->>       Catégorie
```

Une `Catégorie` regroupe un ensemble d'inscriptions concernant des licenciés ayant mêmes caractéristiques et souhaitant se mesurer durant la compétition. Les caractéristiques des inscriptions et des licenciés doivent être conformes aux caractéristiques de la catégorie définie par `Catégorie_ref` (`equipage`, `âge` et `sexe`).

```
Catégorie           caractéristique
                <-/---------------->       Catégorie_ref
                    inscrits
                <<-/------------/->>       Inscription
```

Un licencié peut contracter plusieurs inscriptions dans une compétition mais une seule inscription par catégorie. Les caractéristiques des licenciés ayant contracté une inscription dans une catégorie doivent être compatibles avec celles de la catégories décrites dans la catégorie de référence associée (âge, sexe et nombre d'équipiers dans l'inscription).

Une `Inscription` est une forme de contrat souscrit par un (`Inscription_Mono`) ou deux (`Inscription_Bi`) licenciés dans une catégorie. Ce contrat lui ou leur permet de participer à la compétition et de se mesurer aux autres compétiteurs de la catégorie. Les deux licenciés d'une `Inscription_Bi` doivent appartenir au même club.

`Inscription`	`numéro` <-/---------->	`Entier`
	`dossard` <-/----------/->	`Entier`
	`prestation` <<-/----------/->	`Prestation`

`Inscription = Inscription_Mono ∪ Inscription_Bi`

`Inscription_Mono`	`coureur` <<-/---------->	`Licencié`
`Inscription_Bi`	`coureurs` (2,2) <<-/---------->>	`Licencié`

Une `Prestation` représente le résultat d'un coureur (`Mono`) ou de deux coureurs (`Bi`). Le total d'une manche est égal à la somme du temps et des points de cette manche. La performance d'un coureur est constituée par le meilleur total des deux manches.

`Prestation`	`manche1` <----------/->	`Résultat`
	`manche2` <----------/->	`Résultat`
	`performance` <<-/---------->	`Réel`
`Résultat`	`statut` <<-/---------->	`{non-parti, arrivé, abandon}`
	`temps` <<-/---------->	`Réel`
	`points` <<-/---------->	`Entier`
	`total` <<-/---------->	`Réel`

Pendant l'étape de déroulement du concours, des données supplémentaires sont nécessaires. La liste des dossards permet d'accéder à une inscription connaissant son dossard et le classement général donne le classement de toutes les inscriptions indépendamment de leur catégorie. Pour chaque catégorie, les inscriptions sont classées selon leur performance.

`Compétition`	`liste dossards` <<-/---------->	`Entier × Inscription`
	`classement général` <<-/---------->	`Entier × Inscription`
`Catégorie`	`classement` <<-/---------->	`Entier × Inscription`

3 Spécifications fonctionnelles

Pour chaque étape du déroulement de la compétition nous présentons les services

attendus de l'application. Le système doit permettre des retours à des états antérieurs (cf. figure B-1) ainsi que l'édition de listes générales : liste des clubs, des licenciés, des catégories de référence, des catégories.

L'application peut être amenée à fonctionner dans un environnement précaire (alimentation par groupe électrogène, intempéries, etc.). Aussi toute information saisie par l'opérateur doit être dès que possible validée et transférée sur une mémoire permanente pour prévenir toute interruption d'alimentation.

3.1 Initialisation de la manifestation

Cette phase correspond à la mise en place d'une nouvelle compétition. Il faut annuler les informations concernant la course précédente (inscriptions, dossards, résultats, etc.) et saisir les informations concernant la nouvelle : nom, date, lieu et catégories autorisées à concourir.

L'application ne doit pas obligatoirement conserver l'historique de ces différentes compétitions. La compétition peut se dérouler en plusieurs séquences entre lesquelles le système doit pouvoir être arrêté sans perte d'informations.

3.2 Inscription des athlètes

Pendant cette étape, l'application doit permettre de saisir les inscriptions en vérifiant leur cohérence (âge, sexe, équipage, club, etc.) et en assurant l'identification de chaque inscription. Les inscriptions peuvent être saisies selon différentes procédures : soit athlète par athlète (équipage par équipage), soit par catégorie (plusieurs athlètes ou équipage à la suite dans la même catégorie). Par ailleurs, l'opérateur doit pouvoir vérifier une inscription en y accédant par le numéro d'inscription et modifier une inscription : modification de la catégorie, du coureur (par son numéro de licence).

Lorsque les dossards ont été attribués, les nouvelles inscriptions saisies sont complétées par un dossard dont l'attribution est à la charge de l'opérateur mais l'application doit vérifier la contrainte d'unicité des dossards.

On rappelle que tous les coureurs succeptibles de s'inscrire sont licenciés et donc identifiés par un numéro de licence mais que chaque licencié peut s'inscrire dans différentes catégories.

3.3 Attribution des dossards

Cette étape permet d'attribuer à chaque inscription un dossard permettant de l'identifier dans l'ensemble des inscriptions. L'opérateur fixe d'abord l'ordre des catégories puis le nombre de dossards «vides» placés au début de chaque catégorie et réservés au inscriptions tardives. Ensuite, l'attribution commence au dossard 1 et s'effectue en continu avec incrément de 1 en suivant l'ordre des catégories fixé au préalable. Pour chaque catégorie on réserve le nombre de dossards vides puis on attribue un dossard à chaque inscription et on passe ensuite à la catégorie suivante. Au sein de chaque catégorie, l'attribution des dossards aux inscriptions peut être automatique (déterminée

aléatoirement par l'application) ou manuellement (fixée explicitement par l'opérateur). L'application doit bien entendu assurer dans les deux cas, la continuité des dossards et le respect du nombre de dossards vides.

Après cette phase l'opérateur peut imprimer la liste des départs. Cette liste donne dans l'ordre des dossards, pour chaque dossard, le nom, le prénom et le club du coureur correspondant (deux coureurs pour les catégories bi). La modification d'un dossard est permise, pour cela l'opérateur indique l'ancien dossard et le nouveau dossard qui doit bien entendu être libre et dans la même catégorie que l'ancien.

Après cette phase (dans les états «Dossards affectés», «Course lancée» ou «Course arrêtée») l'affectation globale des dossards n'est plus autorisée.

3.4 Déroulement du concours

Pendant le déroulement du concours, l'application permet essentiellement de saisir, de modifier et d'afficher un résultat, d'établir et d'imprimer des classements provisoires.

A la fin du concours l'application doit permettre d'éditer de façon soignée et complète, l'ensemble des résultats de la compétition. Le format d'impression de chaque inscription est le suivant :

(place, dossard, nom, prénom, club, temps1, pénalité1, total1, temps2, pénalité2, total2, résultat) dont les champs (nom, prénom) sont à doubler pour les inscriptions bi.

4 Spécifications Externes

4.1 Généralités

Le système fonctionne sous les contraintes suivantes. Il est utilisé seulement par des utilisateurs non experts. Il fonctionne sur écran monochrome unique et l'interface est conviviale de type Macintosh. Toute fonction invalide ou interdite ne doit pas être activable et elle doit être affichée en grisé. L'application ne propose pas de fonctions d'aide. Dans les fenêtres proposant des boutons à l'utilisateur, il doit toujours exister un bouton privilégié activable par retour chariot (return) ou par souris. Si le temps d'exécution d'une commande est supérieur à une seconde on doit en informer l'utilisateur par un sablier, les traitements particulièrement longs doivent proposer un sablier indiquant la progression du traitement. Toute commande entraînant une perte d'information doit demander confirmation à l'utilisateur avec un texte indiquant les conséquences de l'action en cours. Pendant la gestion du concours, le retour d'information doit être rapide, notamment pour indiquer la place d'un coureur et pour mettre à jour les informations affichées.

Le tableau suivant présente la barre de menu principale et les différents menus déroulants :

L'évolution de ce menu principal selon l'état de la course est mis en évidence de la façon suivante :

Fichier	Inscriptions	Dossards	Course	Catégorie
Nouvelle Course --- Liste Licenciés Liste Clubs --- Quitter	Créer Par Catégorie Modifier Liste Inscrits --- Annuler Inscrits	Attribuer Modifier Liste Départs --- Annuler Dossards	Lancer Arrêter --- Résultat Afficher Imprimer --- Annuler Résultats	Créer Catégorie Modif. Catégorie Liste Catégories

- les opérations interdites sont notées en italique (ex : *annuler les résultats*),
- les opérations devenues autorisées sont soulignées (ex : attribuer),
- les opérations devenues interdites sont notées en italiques et barrées (ex : ~~*annuler inscrits*~~).

La présentation des fenêtres suit les conventions graphiques suivantes

(Inscrire) (Annuler) Boutons activables par l'utilisateur (bouton par défaut et bouton normal)

[ ] Zone de saisie d'une information utilisateur

INSCRIPTION
Libellé : Texte ou commentaire fixe

On notera que toutes les fenêtres ne figurent pas dans ce document, notamment certains formulaires simples sont décrits seulement textuellement.

4.2 Opérations élémentaires

Le lancement de l'application «Gestion de compétitions» provoque l'affichage d'une fenêtre qui persiste pendant toute la durée d'activation de l'application et qui présente le nom, la date et le lieu de déroulement de la compétition courante.

Compétition : Championnats de France
Date : 27 au 30 Juillet 1992
Lieu : Bourg Saint Maurice

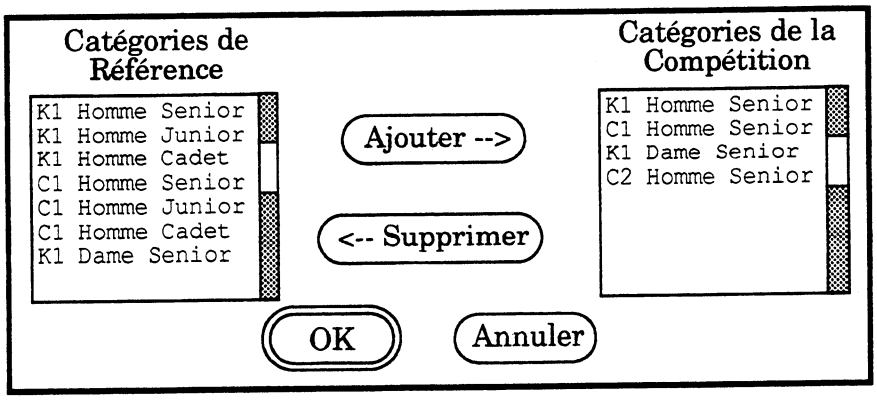
Le menu principal est ensuite affiché et l'utilisateur dispose de l'initiative du dialogue. Ce menu dépend de l'état de la compétition, par exemple le menu affiché quand les dossards n'ont pas encore été attribués et différent du menu affiché quand des résultats ont été saisis. On rappelle que l'opérateur peut interrompre l'application par l'opération quitter du menu **Fichier** puis relancer l'application ultérieurement.

L'opération **Nouvelle Course** du menu **Fichier** permet d'initialier une nouvelle com-

pétition. Après une demande de confirmation portant sur l'annulation de la compétition présente, l'opérateur saisit les caractéristiques de la nouvelle compétition dans un formulaire.

Les opérations **Liste Licenciés** et **Liste Club** permettent d'afficher ou d'imprimer la liste des licenciés et la liste des clubs.

L'opération **Créer Catégorie** du menu **Catégorie** permet d'ajouter une nouvelle catégorie de référence et l'opération **Liste Catégories** permet d'afficher ou d'imprimer la liste des catégories de références ou la liste des catégories de la compétition. L'opération **Modif. Catégorie** permet de définir les catégories autorisées à concourir dans la compétition parmi les catégories de référence.



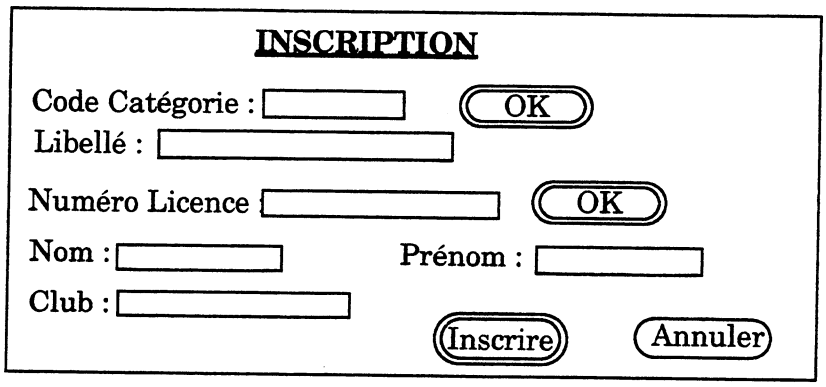
4.3 Inscription

Lorsqu'aucune inscription n'a été saisie le menu principal est le suivant :

Fichier	**Inscriptions**	***Dossards***	***Course***	**Catégorie**
Nouvelle Course	Créer	*Attribuer*	*Lancer*	Créer Catégorie
---	Par Catégorie	*Modifier*	*Arrêter*	Modif. Catégorie
Liste Licenciés	*Modifier*	*Liste Départs*	---	Liste Catégories
Liste Clubs	*Liste Inscrits*	---	*Résultat*	
---	---	*Annuler Dossards*	*Afficher*	
Quitter	*Annuler Inscrits*		*Imprimer*	

			Annuler Résultats	

L'opération **Créer** du menu **Inscription** permet de saisir dans un formulaire, une nouvelle insciption en indiquant tout d'abord la catégorie, l'application réagit alors en en présentant un formulaire adapté, selon la catégorie, pour recevoir un licencié ou deux licenciés introduits par les numéros de licence. Si les dossards ont été attribués le formulaire doit aussi présenter un champ pour saisir le dossard attribué par l'opérateur à cette nouvelle inscription.



L'opération **Par catégorie** permet de fixer une catégorie et d'effectuer une suite d'inscriptions dans cette catégorie. L'opération **Modifier** affiche une inscription dans un formulaire identique à celui de l'opération **Créer**. L'opération **Liste Inscrits** imprime la liste des inscrits soit par ordre d'inscription, soit par club, soit par catégorie, soit par ordre alphabétique.

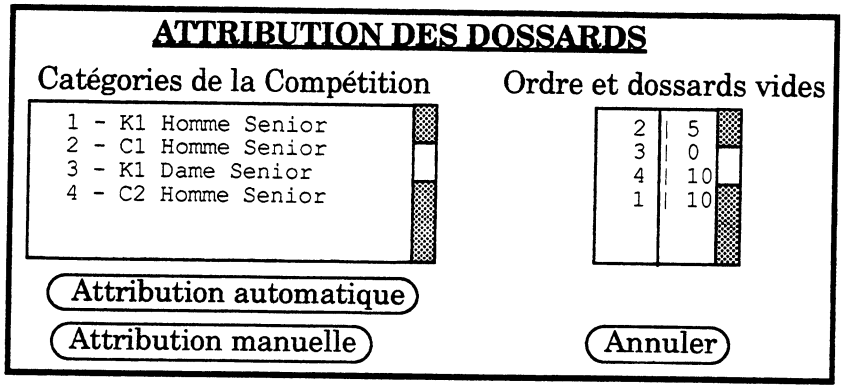
4.4 Attribution des dossards

Après la première inscription le menu principal permet d'affecter les dossards.

Fichier	**Inscriptions**	**Dossards**	***Course***	**Catégorie**
Nouvelle Course	Créer	Attribuer	*Lancer*	Créer Catégorie
----------	Par Catégorie	*Modifier*	*Arrêter*	Modif. Catégorie
Liste Licenciés	Modifier	*Liste Départs*	----------	Liste Catégories
Liste Clubs	Liste Inscrits	----------	*Résultat*	
----------	----------	*Annuler Dossards*	*Afficher*	
Quitter	Annuler Inscrits		*Imprimer*	

			Annuler Résultats	

L'opération d'affectation des dossards présente à l'opérateur une fenêtre permettant de fixer l'ordre des catégories et le nombre de dossards libres laissés au début de chaque catégorie. L'opérateur peut ensuite déclencher les opérations d'affectations.



Pour l'opération d'affectation manuelle, l'opérateur saisit successivement des couples (numéro de dossard, numéro d'inscription) et l'application doit assurer le respect des contraintes d'intégrité : unicité des dossards, nombre de dossards libres.

Après l'attribution des dossards, le menu principal est le suivant :

Fichier	Inscriptions	Dossards	Course	Catégorie
Nouvelle Course --- Liste Licenciés Liste Clubs --- Quitter	Créer Par Catégorie Modifier Liste Inscrits --- ~~*Annuler Inscrits*~~	~~*Attribuer*~~ Modifier Liste Départs --- Annuler Dossards	Lancer *Arrêter* --- *Résultat* *Afficher* *Imprimer* --- *Annuler Résultats*	Créer Catégorie Modif. Catégorie Liste Catégories

L'opération **Liste Départ** imprime les dossards en ordre croissant en indiquant les titres des catégories et pour chaque dossard, les nom(s), prénom(s) et club du ou des licenciés correspondant au dossard.

L'opération **Modifier** du menu **Dossards** permet de permuter les dossards de deux inscriptions dans la même catégorie ou de permuter une inscription et un dossard vide. Pour cela, l'application présente un formulaire simple contenant les deux dossards.

L'opération **Annuler Dossards** permet de revenir à l'étape précédente.

4.5 Lancement de la compétition

Une fois les dossards attribués, le concours peut effectivement commencer. La saisie des résultats doit être précédée par l'activation de l'opération **Lancer** dans le menu **Course**. Le menu principal est alors le suivant :

Fichier	Inscriptions	Dossards	Course	Catégorie
Nouvelle Course --- Liste Licenciés Liste Clubs --- Quitter	Créer Par Catégorie Modifier Liste Inscrits --- *Annuler Inscrits*	*Attribuer* Modifier Liste Départs --- ~~*Annuler Dossards*~~	~~*Lancer*~~ Arrêter --- Résultat Afficher Imprimer --- *Annuler Résultats*	Créer Catégorie Modif. Catégorie Liste Catégories

L'écran présenté à l'opérateur comporte trois fenêtres (cf. figure C-2) :

- La fenêtre F1 permet à l'opérateur de saisir un résultat en indiquant d'abord le dossard puis la manche et enfin la performance (temps et points), l'application répond alors en affichant le total, la place et en mettant à jour si nécessaire, les

autres fenêtres (F2 et F3).

- La fenêtre F2 présente une liste contenant les cinq derniers résultats saisis par l'opérateur.
- La fenêtre F3 est obtenue par l'activation de la commande **Afficher** du menu **Course** et affiche le classement d'une catégorie.

La partie gauche de l'écran constitue une zone de travail dans laquelle se déroulent les autres opérations activables par l'opérateur (inscriptions, listes, etc.).

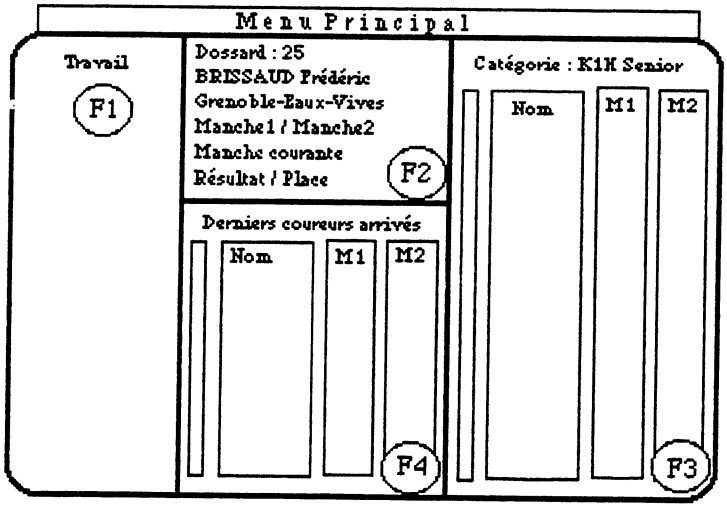


Figure C-2 : Ecran pour la gestion de la compétition.

La course peut être arrêtée par l'activation de la commande **Arrêter** du menu **Course**. Le menu présenté est alors le suivant :

Fichier	Inscriptions	Dossards	Course	Catégories
Nouvelle Course --- Liste Licenciés Liste Clubs --- Quitter	Créer Par Catégorie Modifier Liste Inscrits --- *Annuler Inscrits*	*Attribuer* Modifier Liste Départs --- *Annuler Dossards*	Lancer ~~*Arrêter*~~ --- ~~*Résultat*~~ ~~*Afficher*~~ ~~*Imprimer*~~ --- Annuler Résultats	Créer Catégorie Modif. Catégorie Liste Catégories

La course peut alors à nouveau être lancée.

5 Modélisation MOSAÏC

Cette section présente quelques éléments de la modélisation MOSAÏC de l'application «Gestion de compétitions sportives».

5.1 Classe projet

```
classe projet Gestion_Compétitions_Sportives
  racines de persistance
        compétitions : Ensemble(réf Compétition)
        compétition_courante : réf Compétition
        clubs : Ensemble(réf Club)
        licenciés : Ensemble(réf Licencié)
  racines d'application
        gestion_compétition : Classe_initiale_g_course
fin
```

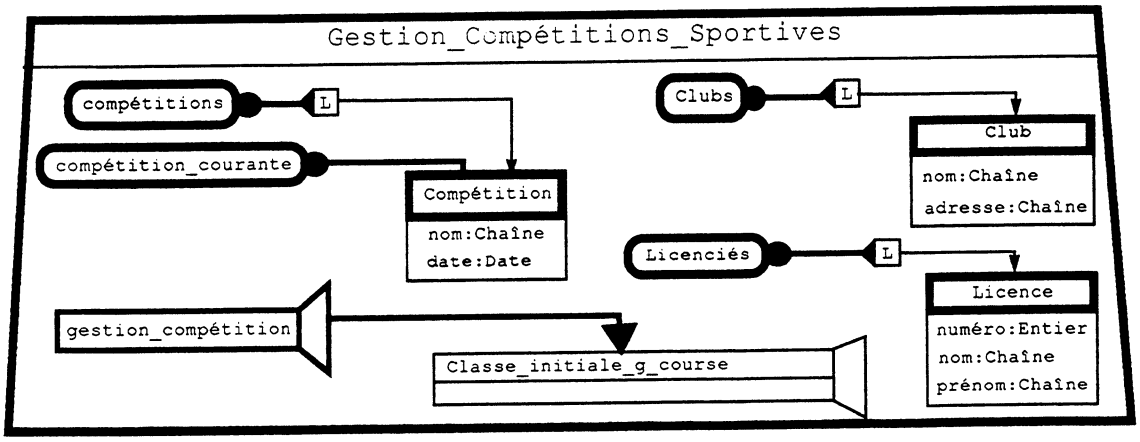


Figure C-3 : Classe projet principale

5.2 Schéma de la base d'objets

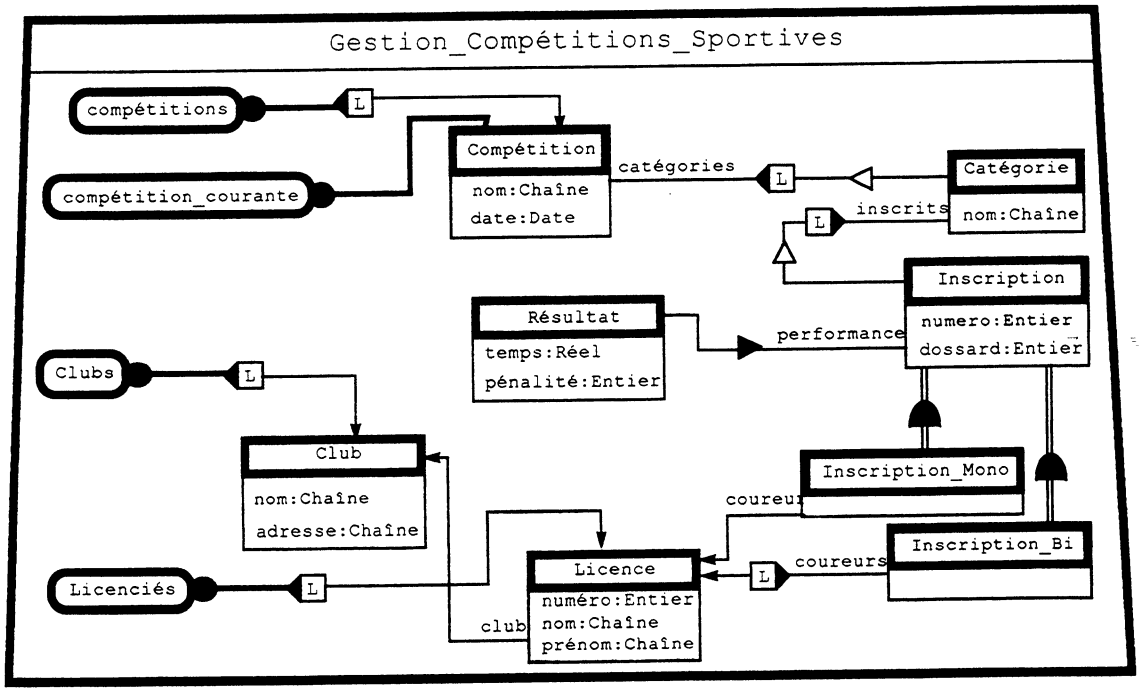


Figure C-4 : Schéma de la base d'objets.

Définition plus précise de quelques classes applicatives :

```
(********************* Classe Applicative Compétition **********************)

 classe applicative Compétition
   facette f_dossard : Catégorie_Dossard
   facette f_course : Catégorie_Course

   exporte        fonctions nom_compétition, date_compétition,
                            lieu_compétition, catégories

                  méthodes  initialiser, ajouter_catégorie, supprimer_catégorie,
                            numéro_inscription_suivant, lancer_course,
                            arrêter_course

                  états     initial, dossards_non_affectables,
                            dossards_affectables, dossards_non_affectés,
                            dossards_affectés

   attributs      nom_compétition : Chaîne
                  date_compétition : Chaîne
                  lieu_compétition : Chaîne
                  catégories : Liste (dép Catégorie )
                  compteur_inscription : Entier

   utilise Catégorie

   états    initial = (card(catégories) = 0)
            minimal = (nom_compétition ≠ "") et (card(catégorie) ≤ 50)

   invariant
      minimal
      dossards_affectables = (∃ c ∈ catégories / c.dossards_affectables)
      dossards_non_affectés = (∀ c ∈ catégories, c.dossards_non_affectés)
      dossards_affectés = (∀ c ∈ catégories, c.dossards_affectés)

   méthode initialiser(nom : Chaîne, date : Chaîne, lieu : Chaîne)
   méthode ajouter_catégorie (cat : Catégorie)
   méthode supprimer_catégorie (cat : Catégorie)
   méthode numéro_inscription_suivant : Entier
   méthode lancer_course
   méthode arrêter_course
 fin


 classe facette Catégorie_Dossard
   exporte        méthodes ordre_des_départs, affecter_les_dossards
   attributs      ordre_des_catégories : Table [1..50] (réf Catégorie)

   utilise Catégorie

   méthode ordre_des_départs (ordre : Table [1..50] (réf Catégorie))
   méthode affecter_les_dossards
      messages Catégorie • C_affecte_dossards
 fin


 classe facette Catégorie_Course
   exporte        méthodes initialiser, classer
   variables
      classement_général : Table [1..500] (réf Inscription)

   méthode initialiser
   méthode classer (i : Inscription)
 fin


(********************* Classe Applicative Catégorie **********************)

 classe applicative Catégorie
   exporte        fonctions nom, inscrits, classement
                  méthodes  inscrire, supprimer, classement_alphabétique,
```

```
                    C_affecte_dossards, inialiser_classement, classer
          états     initial, dossards_non_affectables,
                    dossards_affectables, dossards_non_affectés,
                    dossards_affectés

  attributs     nom : Chaîne
                inscrits : Liste (dép Inscription)

  variables
     class_alphab : Table [1..100] (réf Inscription),
     classement : Table [1..100] (réf Inscription)

  utilise    Inscription

  états    initial = (card(inscrits) = 0)
           minimal = (nom_catégorie ≠ "") et (card(inscrits) ≤ 100)
           dossards_non_affectables = (card(inscrits) < 3)
           dossards_affectables = (card(inscrits) ≥ 3)
           dossards_non_affectés = (∀ i ∈ inscrits, i.dossard = 0)
           dossards_affectés = (∀ i ∈ inscrits, i.dossard ≠ 0)

  méthode primaire inscrire (i : Inscription)
        précondition dossards_non_affectés et (card(inscrits) ≤ 99)
                     et (j ∈ inscrits ⇒ i.coureur ≠ j.coureur)
        postcondition inscrits = ancien (inscrits) ∪ {i}

  méthode primaire supprimer (i : Inscription)
        précondition dossards_non_affectés et (i ∈ inscrits)
        postcondition inscrits = ancien (inscrits) \ {i}

  méthode primaire classement_alphabétique
        précondition VRAI
        postcondition (i, j ∈ [1..card(inscrits)]) ⇒ (i ≤ j ⇔
                (class_alphab[i].coureur.nom ≤ class_alphab[j].coureur.nom))

  méthode secondaire C_affecte_dossards (dossard_courant : Entier) : Entier
     temporaire
     messages Inscription • I_affecte_dossard
     précondition dossards_affectables
     postcondition dossards_affectés

  méthode inialiser_classement
     temporaire

  méthode classer (i : Inscription) : Entier
     temporaire
fin


(************* Classe Applicative Inscription et sous-classes *************)

classe applicative Inscription (*classe virtuelle*)
  exporte       fonctions numéro_inscription, dossard, résultat

  attributs     numéro_inscription : Entier
                dossard : Entier
                résultat : réf Résultat

  méthode secondaire I_affecte_dossard (dossard_courant : Entier)
fin


classe applicative Inscription_Mono
  est-un Inscription

  exporte     fonctions coureur
              méthodes créer

  attributs
     coureur : réf Licence              (*Inscription_Mono = 1 Licencié*)
     performance : Réel
  méthode créer (l : Licence)
fin
```

```
classe applicative Inscription_Bi
  est-un Inscription
  exporte     fonctions coureurs
              méthodes créer

  attributs
     coureurs : Ensemble (réf Licence)
  invariant card (coureurs = 2)        (*Inscription_Bi = 2 Licenciés*)
  méthode créer (l : Ensemble (réf Licence))
fin
```

5.3 Interface de l'application

Le menu général de l'application est introduit par la classe initiale `Classe_initiale_g_course` (cf. figure C-5).

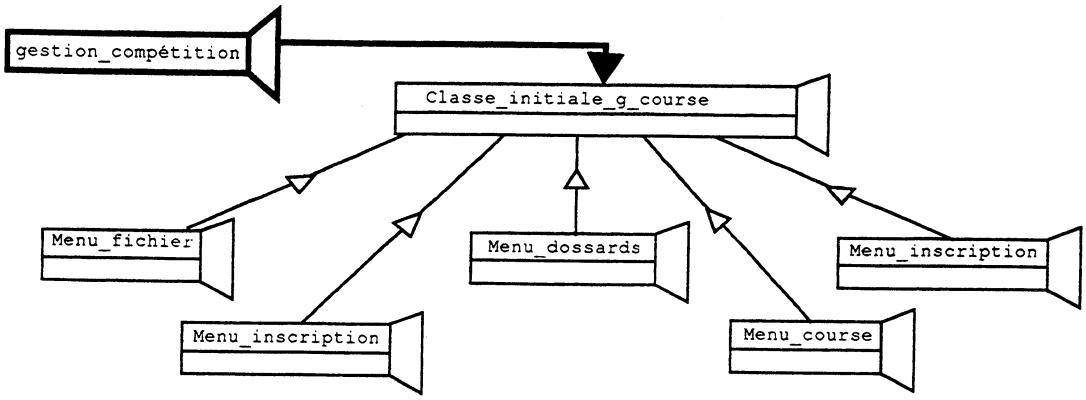


Figure C-5 : Définition structurelle de la classe `Init_compétition_et_menu_génér.`

```
classe interactive Classe_initiale_g_course
  (*pas de facette présentation pour cette classe*)

  attributs      (*contrôle*)
     menu_fichier : dép Menu_Fichier
     menu_inscription : dép Menu_Inscription
     menu_dossard : dép Menu_Dossard
     menu_course : dép Menu_Course
     menu_catégorie : dép Menu_Catégorie...

  méthode lancer_application (*méthode exécutée au lancement*)
     prologue
     (*transmet le contrôle du dialogue à l'utilisateur*)
     épilogue
  fin

  méthode prologue    (*détermine la course courante, initialise les classes*)
                      (*applicatives, instancie les classes interactives qui*)
                      (*gèrent les différents menus et affiche ces menus*)

  méthode épilogue (*ferme toutes les fenêtres*)
fin
```

La facette du sous-menu course décrit les commandes activables dans ce sous-menu.

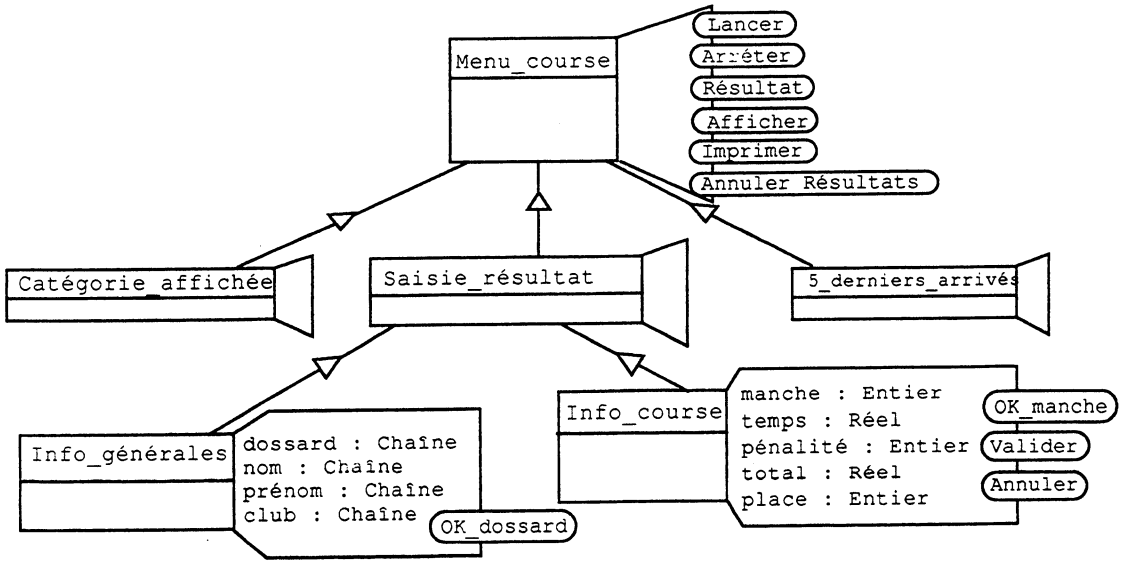


Figure C-6 : Définition structurelle du menu course et de la commande `saisie_résultat`.

```
classe interactive Menu_Course
  facette présentation
     méthodes Lancer, Arréter, Afficher, Imprimer, Annuler_résultats
  fin

  attributs      (*contrôle*)
     saisie_résultat : dép Saisie_Résultat
     affichage : dép Affichage
  méthodes initialiser
fin

classe interactive Saisie_Résultat
  attributs      (*contrôle*)
     info_générales : dép Info_Générales
     info_course : dép Info_Course
fin

classe interactive Info_Générales
  facetteprésentation :
     attributs
        dossard : Entier     commentaire "Numéro de dossard ?"
        nom : Chaîne         commentaire "Nom :"
        prénom : Chaîne
        club : Chaîne        commentaire "Club :"
     méthodes OK_dossard
  fin

  attributs      (*contrôle*)
     licencié : réf Licencié
  méthodes OK_dossard
fin

classe interactive Info_course
  facette présentation :
     attributs
        manche : Entier          commentaire "Numéro de la manche (1 ou 2) :"
        temps : Réel             commentaire "Temps ?"
        pénalités : Entier       commentaire "Points ?"
        total : Réel             commentaire "Total :"
        place : Entier           commentaire "Classement :"
     méthodes OK_manche, Valider, Annuler
  fin

  attributs      (*contrôle*)
     inscription_courante : réf Inscription
```

```
  méthode OK_manche
  méthode Valider
  méthode Annuler
fin
```

Quelques dépendances fonctionnelles entre les classes interactives de l'interface interactive sont présentées dans les figures C-7 et C-8.

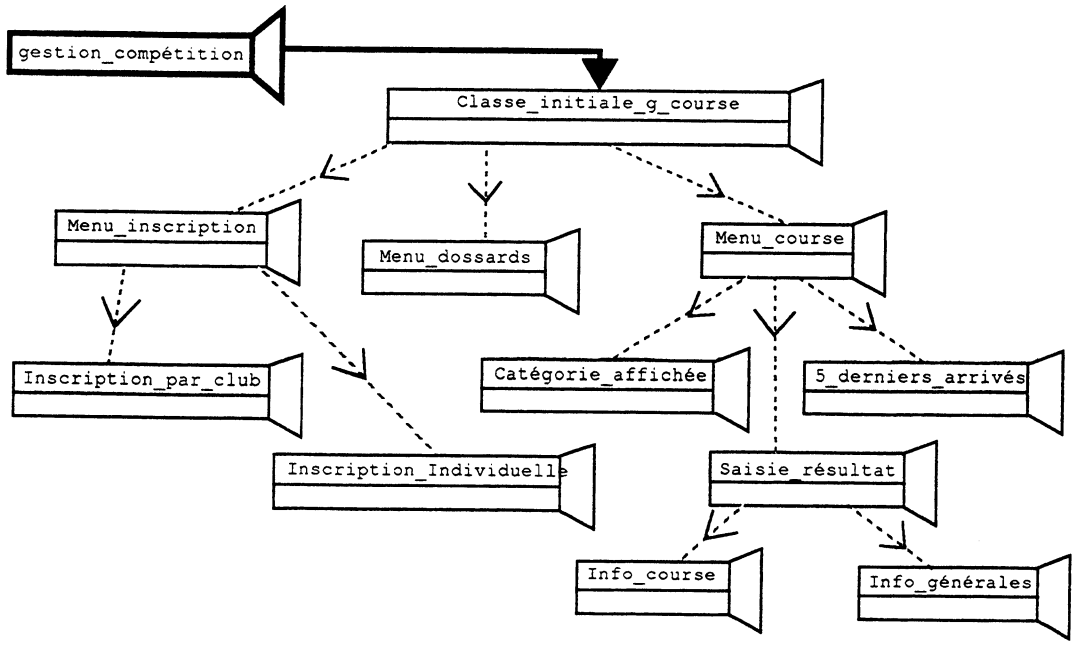


Figure C-7 : Vue partielle et fonctionnelle de l'interface interactive.

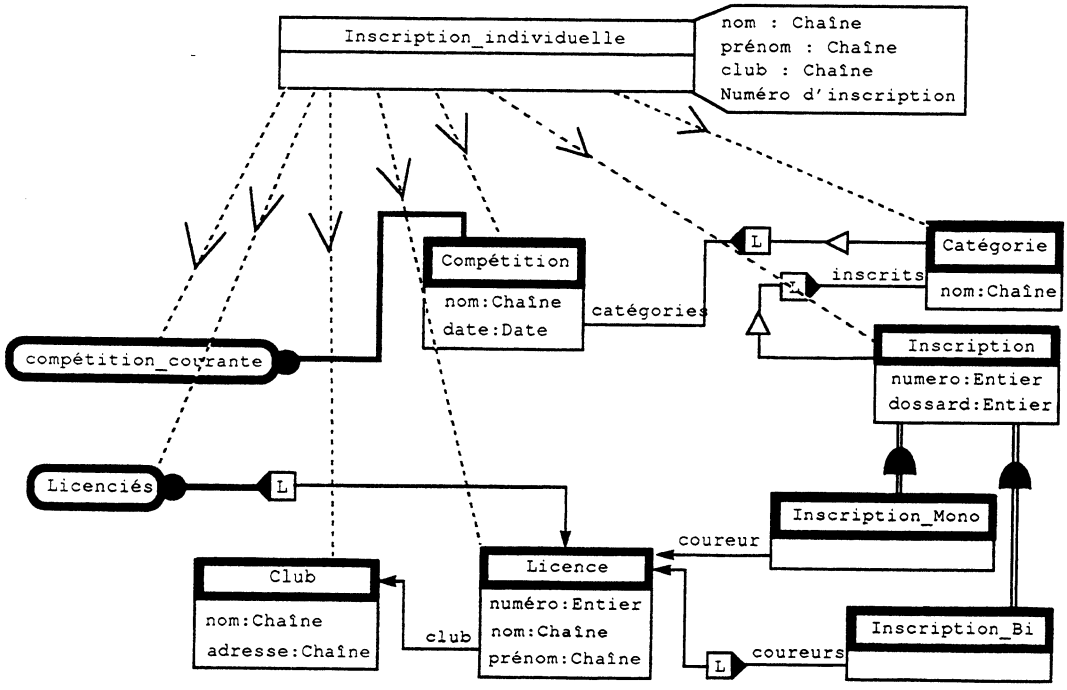


Figure C-8 : Dépendances fonctionnelles de la classe `Inscription_individuelle`.

6 Réalisation O2

L'application programmée en O2 est un sous-ensemble de l'application spécifiée dans les sections 1 à 4. Le texte O2 a été obtenu en appliquant les règles de traduction présentées dans la section 2 du chapitre 6 à la modélisation partielle décrite ci-dessus (section 5).

6.1 Introduction des classes O2

Les classes applicatives et les racines de persistances MOSAÏC (cf. Figure C-4) ont été traduite en classes et objets nommés O2 de la façon suivante :

Classes Applicatives MOSAÏC	Classes O2 aspects persistants	Classe O2 aspects temporaires
Competition Categorie Inscription	Competition Categorie Inscription	Competition_partie_temporaire Categorie_partie_temporaire

Racine de persist. MOSAÏC	Objet nommé O2	
Competition_courante	name competition_courante	: Competition

Nous avons proposé (cf. Chapitre 6 section 2.5) des classes génériques O2 et des objets nommés O2 pour gérer les classes applicatives partiellement temporaires :

```
O_partiellement_temporaire
Partie_temporaire
Ens_O_partiellement_temporaire

name Objets_partiellement_temporaires :Ens_O_partiellement_temporaire;
```

Les classes interactives et la racine d'application MOSAÏC ont été traduites par des classes O2 et une application O2 de la façon suivante :

Classes Interactives MOSAÏC	Classes O2 partie controle	Classes O2 partie presentation
CI_gest_course	Cl_initiale_applick	/*pas de présentation*/
Menu_fichier	Menu_fichier	/*Menu_fichier*/
Menu_inscription	Menu_inscriptions	/*Menu_inscriptions*/
Menu_dossards	Menu_dossards	/*Menu_dossards*/
Menu_courses	Menu_courses	/*Menu_courses*/
Menu_categories	Menu_categories	/*Menu_categories*/
Nouvelle_course	Controle_nouv_course	Present_nouv_course
Créer_inscription	Controle_creer_insc	Present_creer_insc
Liste_inscription	Controle_liste_insc	Present_liste_insc
Annuler_inscriptions	Controle_annul_insc	Present_annul_insc
Liste_dossards	Controle_liste_doss	Present_liste_doss
Annuler_dossards	Controle_annul_doss	Present_annul_doss
Saisie_résultat	Controle_saisie_resu	Present_saisie_resu Present_derniers_arrives
Créer_catégorie	Controle_creer_categ	Present_creer_categ

Racine d'application MOSAÏC	application O2	
gestion_competition	applick program dashboard	

Différentes classes prédéfinies O2 spécialisées dans la gestion du dialogue ont été utilisées :

```
Component, Prompt, Single_selection, Label, Dialog_box
```

De nouvelles classes génériques ont été ajoutées pour la gestion du dialogue :

```
Dial_2_boutons /* sous-classe de Dialog_box */
Dial_1_bouton /* sous-classe de Dialog_box */
Dial_sans_bouton /* sous-classe de Dialog_box */
Present_generic
Multiple_boutons_generic
Menu_generic
Controle_generic
```

6.2 Gestion des parties temporaires des classes applicatives

Cette section regroupe les mécanismes de gestion des parties temporaires des classes applicatives.

```
/* Classe O_partiellement_temporaire : racine de la hierarchie des */
/*              classes partiellement temporaires                  */

class O_partiellement_temporaire
read type tuple (     partie_temporaire : Partie_temporaire,
                      promu : boolean)
method      public creation,
            public promotion,
            public creer_partie_temporaire,         /*differee*/
            public supprimer_partie_temporaire
end;

/* Classe Partie_temporaire : racine de la hierarchie des parties */
/*              temporaires des classes partiellement temporaires */

class Partie_temporaire
read type tuple (partie_persistante : O_partiellement_temporaire)
end;

/* Classe Ens_O_partiellement_temporaire : classe de l'objet nomme charge*/
/*          de regrouper les objets en partie temporaires et persistants */

class Ens_O_partiellement_temporaire
read type tuple (ensemble : set(O_partiellement_temporaire))
method      public initialise,
            public debut_application,
            public fin_application,
            public ajoute(objet : O_partiellement_temporaire)
end;

/* Objet nomme Objets_partiellement_temporaires : objets nomme       */
/*              regroupant tous les objets partiellememt temporaires */

name Objets_partiellement_temporaires : Ens_O_partiellement_temporaire;


/** Classe O_partiellement_temporaire: les methodes **/
```

```
method body creation in class O_partiellement_temporaire {
      self -> promu = false; };

method body promotion in class O_partiellement_temporaire {
      if (self -> promu == false) {
            self -> promu = true;
            self -> creer_partie_temporaire;
            Objets_partiellement_temporaires -> ajoute(self);
};    };

method body creer_partie_temporaire in class O_partiellement_temporaire { };

method body supprimer_partie_temporaire in class O_partiellement_temporaire {
        self -> promu = false;
        self -> partie_temporaire = nil; };

method body initialise in class Ens_O_partiellement_temporaire {
      self -> ensemble = set(); };

/** Classe Ens_O_partiellement_temporaire : les methodes **/

/* debut_application: initialise l'ensemble des objets en partie temporaires  */

method body debut_application in class Ens_O_partiellement_temporaire {
        self -> ensemble = set(); };

/* fin_application: supprime les parties temporaires des obj. partiellement temp. */

method body fin_application in class Ens_O_partiellement_temporaire {
o2 O_partiellement_temporaire o;
        for (o in self -> ensemble) {o -> supprimer_partie_temporaire; }; };

method body ajoute(objet : O_partiellement_temporaire)
                in class Ens_O_partiellement_temporaire {
        self -> ensemble += set(objet); };

/* initialisation de l'objet nomme Objets_partiellement_temporaires */

run body {
   Objets_partiellement_temporaires = new Ens_O_partiellement_temporaire;
   Objets_partiellement_temporaires -> initialise; };
```

6.3 Traduction des classes applicatives

Cette section décrit les définitions et les principales méthodes des classes applicatives.

6.3.1 Définition, structure et signature des méthodes des classes applicatives

```
/**********************/
/* Classe Competition */
/**********************/

class Competition
inherit O_partiellement_temporaire
public type tuple (    nom : string, date : string, lieu : string,
                       categories : set(Categorie),
                       compteur_insc : integer,
                       etat : string)
method      public creer_partie_temporaire,
            public initialiser( nom :  string, date : string, lieu : string),
            public ajouter_categorie( cat : Categorie),
            public cat_de_insc( insc : Inscription) : Categorie,
            public cat_de_nom( nom_cat : string) : Categorie,
            public supprimer_cat( nom_cat : string),
            public inscrire( nouv_insc : Inscription, nom_cat : string),
            public num_insc_suiv : integer,
            public annuler_les_inscrits,
```

```
            public annuler_les_dossards,
            public annuler_les_resultats
end;

/* Classe Competition_partie_temporaire : partie temporaire de Competition  */

class Competition_partie_temporaire
inherit Partie_temporaire
read type tuple(     liste_dos : list(Inscription))
method      public initialiser (pp : Competition),
            public attribuer_les_dossards,
            public init_list_dos,
            public insc_de_dos (dos : integer) : Inscription,
            public dos_libre,
            public lancer_course
end;

/* redefinition des attributs partie_temporaire et partie_persistante */

attribute partie_persistante : Competition in class Competition_partie_temporaire;
attribute partie_temporaire : Competition_partie_temporaire in class Competition;

/* Objet nomme competition_courante */

name competition_courante : Competition;

/*********************/
/* Classe Categorie */
/*********************/

class Categorie
inherit O_partiellement_temporaire
public type tuple(     nom : string, acronyme : string,
                       inscrits : set(Inscription))
method      public creer_partie_temporaire,
            public creer (nom : string, acronyme : string),
            public inscrire (i : Inscription),
            public supprimer (i : Inscription),
            public annuler_inscrits,
            public annuler_les_dossards,
            public annuler_les_resultats
end;

/* Classe Categorie_partie_temporaire */

class Categorie_partie_temporaire
inherit Partie_temporaire
read type tuple (      classement : list (Inscription),
                       nb_insc : integer)
method      public initialiser (pp : Categorie),
            public affecter_dossards (dos_courant : integer) : integer,
            public init_course,
            public classer (insc : Inscription) : integer,
            private etablir_classement
end;

/* redefinition des attributs partie_temporaire et partie_persistante */

attribute partie_persistante : Categorie in class Categorie_partie_temporaire;
attribute partie_temporaire : Categorie_partie_temporaire in class Categorie;

/***********************/
/* Classe Inscription */
/***********************/

class Inscription
read type tuple (      num_insc : integer,
                       nom : string, prenom : string,
                       dossard : integer,
                       couru : boolean, resultat : integer)
method      public initialiser( n : string, p : string),
            public affecte_num_insc( num : integer),
            public affecte_dossard( dos : integer),
```

```
            public affecte_resultat( resu : integer),
            public annuler_le_dossard
end;
```

6.3.2 Méthodes de la classe Compétition

```
method body creer_partie_temporaire in class Competition {
      self -> partie_temporaire = new Competition_partie_temporaire;
      self -> partie_temporaire -> initialiser(self); };

method body initialiser (nom :  string, date : string, lieu : string)
            in class Competition {
      self -> creation;
      self -> nom = nom; self -> date = date; self -> lieu = lieu;
      self -> categories = set(); self -> compteur_insc = 0;
};

method body ajouter_categorie (cat : Categorie) in class Competition {
      self -> categories += set(cat); };

/* num_insc_suiv : retourne un nouveau numero d'inscription */

method body num_insc_suiv : integer in class Competition {
      self -> compteur_insc += 1;
      return self -> compteur_insc; };

method body inscrire(nouv_insc:Inscription,nom_cat:string) in class Competition {
o2 Categorie cat;
o2 integer num;
      cat = self -> cat_de_nom(nom_cat);
      if (cat != nil) {      num = self -> num_insc_suiv;
                             nouv_insc -> affecte_num_insc(num);
                             cat -> inscrire( nouv_insc);
};    };


/**** Methodes de la Classe Competition_partie_temporaire ****/

/* methode initialiser : relie la partie temporaire a la partie persistante  */

method body initialiser(pp:Competition) in class Competition_partie_temporaire {
      self -> partie_persistante = pp; };

method body attribuer_les_dossards in class Competition_partie_temporaire {
o2 Categorie cat;
o2 integer compteur;
      compteur = 0;
      for (cat in self -> partie_persistante -> categories) {
            cat -> promotion;
            compteur = cat -> partie_temporaire -> affecter_dossards(compteur);
};    };

/* init_list_dos : initialise la liste des dossards */

method body init_list_dos in class Competition_partie_temporaire {
o2 Inscription insc;
o2 Categorie cat;
o2 integer i;
      self -> liste_dos = list(); /* initialisation de la liste */
      for (i=0; i<500; i++) { self -> liste_dos += list((o2 Inscription) nil); };
            /* construction de la liste des dossards */
      for (cat in self -> partie_persistante -> categories) {
            for (insc in cat -> inscrits) {
                  self -> liste_dos[insc -> dossard] = insc; };
};    };

/* lancer_course : initialise les donnees temp. utilisees pendant la competition */

method body lancer_course in class Competition_partie_temporaire {
o2 Categorie cat;
      self -> init_list_dos;
```

```
                                        /* initialisation des categories */
      for (cat in self -> partie_persistante -> categories) {
            cat -> promotion;
            cat -> partie_temporaire -> init_course;
};    };
```

6.3.3 Méthodes de la classe Catégorie

```
method body creer_partie_temporaire in class Categorie {
      self -> partie_temporaire = new Categorie_partie_temporaire;
      self -> partie_temporaire -> initialiser(self); };

method body creer (nom : string, acronyme : string) in class Categorie {
      self -> nom = nom; self -> acronyme = acronyme;
      self -> inscrits = set();
      self -> promu = false; self -> partie_temporaire = nil; };

method body inscrire (i : Inscription) in class Categorie {
      self -> inscrits += set(i); };

method body annuler_les_dossards in class Categorie {
o2 Inscription insc;
      for (insc in self -> inscrits) { insc -> annuler_le_dossard; }; };


/**** Méthodes de la Classe Categorie_partie_temporaire ****/

method body initialiser (pp : Categorie) in class Categorie_partie_temporaire {
      self -> partie_persistante = pp; };

/* methode affecter_dossards : attribue les dossards */

method body affecter_dossards (dos_courant : integer) : integer
           in class Categorie_partie_temporaire {
o2 Inscription i;
      for (i in self -> partie_persistante -> inscrits) {
            dos_courant += 1;
            i -> affecte_dossard (dos_courant);
      };
      return dos_courant;
};

method body init_course in class Categorie_partie_temporaire {
o2 Inscription i;
      self -> classement = list();
      for (i in self -> partie_persistante -> inscrits) {
            self -> classement += list(i); };
      self -> nb_insc = count(self -> partie_persistante -> inscrits);
      self -> etablir_classement;
};

method body classer (insc : Inscription) : integer
           in class Categorie_partie_temporaire {
o2 integer i;
      self -> etablir_classement;
      for (i=0; i < self->nb_insc; i++) {
            if (self -> classement[i] == insc) { return (i+1); };
};    };

method body etablir_classement in class Categorie_partie_temporaire {
o2 Inscription insc;
o2 boolean permute;
o2 integer i,r1,r2;
      permute = true;
      while (permute) {
            permute = false;
            for (i=0; i<self->nb_insc-1; i++) {
                  r1 = self -> classement[i] -> resultat;
                  r2 = self -> classement[i+1] -> resultat;
                  if (r1>r2) {      insc = self -> classement[i+1];
                                    self -> classement[i+1] = self -> classement[i];
```

```
                                        self -> classement[i] = insc;
                                        permute = true;
};    };    };    };
```

6.3.4 Méthodes de la classe Inscription

```
method body initialiser (n : string, p : string) in class Inscription {
      self -> nom = n; self -> prenom = p;
      self -> dossard = 0;
      self -> couru = false; self -> resultat = 10000;};

method body affecte_num_insc ( num : integer) in class Inscription {
      self -> num_insc = num; };

method body affecte_dossard ( dos : integer) in class Inscription {
      self -> dossard = dos;
      self -> couru = false; self -> resultat = 10000; };

method body affecte_resultat ( resu : integer) in class Inscription {
      self -> resultat = resu; self -> couru = true; };
```

6.4 Classes génériques utilisées pour le dialogue homme-machine

Cette section décrit les classes génériques utilisées pour gérer le dialogue homme-machine.

6.4.1 Présentation, Controle et Menu génériques

```
/*****************************************************************************/
/* Present_generic : facette presentation generique des classes interactives */
/*****************************************************************************/

class Present_generic
type tuple (f_controle : Controle_generic)
end;

/*****************************************************************************/
/* Controle_generic : facette controle generique des classes interactives */
/*****************************************************************************/

class Controle_generic
type tuple (    pere : Controle_generic,  /*l'interface est une arborescence*/
            f_presentation : Present_generic)
method      public maj_pere( pere : Controle_generic)
end;

method body maj_pere( pere : Controle_generic) in class Controle_generic {
      self -> pere = pere; };

/*****************************************************************************/
/* Multiple_boutons_generic : routines pour gerer des fenetres a plusieurs boutons */
/*****************************************************************************/

class Multiple_boutons_generic
inherit Present_generic
read type tuple(liste_boutons : list(tuple(nom : string, activable : boolean)),
            p_presentation : integer,
            x : integer,
            y : integer )
method      public maj_f_controle ( f_controle : Controle_generic ),
            public affiche( x : integer, y : integer ),
            public efface,
            public reprend_le_controle,
            public passe_controle,
            public menu : list(string),
            public init_liste_boutons,      /* methode differee */
            public met_a_jour( modif :
```

```
                list( tuple( nom:string,activable:boolean))),/* differee */
            public desactive_tous_boutons,
            public reactive_liste_boutons
end;

method body maj_f_controle (f_controle : Controle_generic)
            in class Multiple_boutons_generic {
      self -> f_controle = f_controle; };

/** affiche : affiche les boutons ************************/

method body affiche ( x : integer, y : integer ) in class Multiple_boutons_generic {
Lk_resource bouton[2];
            /*supprime les boutons predefinis dans LOOKS*/
      bouton[0].name = "pen"; bouton[0].value = "INVISIBLE";
      bouton[1].name = "eraser"; bouton[1].value = "INVISIBLE";

      self -> x = x; self -> y = y;
      self -> p_presentation = lk_present(self, 0, "", 2, bouton);
      lk_map(self -> p_presentation, COORDINATE,0,0,self->x,self->y);
};

method body efface in class Multiple_boutons_generic {
      lk_delete_presentation(self -> p_presentation); };

/** passe_controle_a_user : se place en attente des actions de l'utilisateur */

method body passe_controle in class Multiple_boutons_generic {
      lk_wait(self -> p_presentation); };

/** reprend_le_controle : le programme reprend le controle */

method body reprend_le_controle in class Multiple_boutons_generic {
            lk_free(self -> p_presentation, ERASE); };

/** met_a_jour : mise a jour des boutons activables */

method body met_a_jour( modif : list( tuple( nom:string, activable:boolean)))
            in class Multiple_boutons_generic {
o2 tuple(nom : string, activable : boolean) n1,n2;
      for (n1 in modif) {
            for (n2 in self -> liste_boutons) {
                  if (n1.nom == n2.nom) {
                        n2.activable = n1.activable;
                        if (n2.activable) { self -> enable_method (n2.nom);
                        } else {self -> disable_method (n2.nom); };
};    };    };    };

/** menu : héritée de la classe Object (construction de la liste des boutons) */

method body menu : list(string) in class Multiple_boutons_generic {
o2 list(string) s;
o2 tuple( nom : string, activable : boolean ) b;
      s = list();
      for (b in self -> liste_boutons) { s += list(b.nom); };
      return s; };

method body desactive_tous_boutons in class Multiple_boutons_generic {
o2 tuple(nom : string, activable : boolean) n1;
      for (n1 in self -> liste_boutons) { self -> disable_method (n1.nom); }; };

method body reactive_liste_boutons in class Multiple_boutons_generic {
o2 tuple(nom : string, activable : boolean) n1;
      for (n1 in self -> liste_boutons) {
            if (n1.activable == true) { self -> enable_method (n1.nom);
            } else {self -> disable_method (n1.nom);};
};    };

/*****************************************/
/* Menu_generic : menu generique */
/*****************************************/

class Menu_generic
```

```
inherit Multiple_boutons_generic
type tuple (    pere : Controle_generic,/*l'interface est une arborescence*/
           f_presentation : Present_generic)
method     public init_menu ( pere : Controle_generic, x : integer, y : integer),
           public maj_menu ( etat : string, com_cour : list(string)), /* differee */
           public init_liste_boutons,
           private menu_actif,
           private menu_inactif
end;

/* maj_menu : m-a-j les commandes activables en fonctions des commandes lancees */

method body init_menu( pere : Controle_generic, x : integer, y : integer)
       in class Menu_generic {
     self -> pere = pere;
     self -> init_liste_boutons;
     self -> affiche (x, y);
     self -> reactive_liste_boutons; };

/** menu_actif : rend les commandes inactivables ************************/

method body menu_inactif in class Menu_generic {
     self -> desactive_tous_boutons; };

/** menu_inactif : rend les commandes activables ************************/

method body menu_actif in class Menu_generic {
     self -> reactive_liste_boutons; };

/** init_liste_boutons : fixe le nom des boutons affichés *****/

method body init_liste_boutons in class Menu_generic { }; /*différée*/
```

6.4.2 Boîtes de dialogue particulières

```
/*****************************************/
/* Contours des fenetres de dialogue     */
/*****************************************/

class Dial_2_boutons
inherit Dialog_box
type tuple (nom_bouton1 : string, nom_bouton2 : string)
method     public init (    s : string, llab : list(list(Component)),
                            nom_bouton1 : string, nom_bouton2 : string),
           public create_presentation : integer
end;

class Dial_1_bouton
inherit Dialog_box
type tuple (nom_bouton : string)
method     public init(s:string,llab:list(list(Component)),nom_bouton:string),
           public create_presentation : integer
end;

class Dial_sans_bouton
inherit Dialog_box
method     public init (s : string, llab : list(list(Component))),
           public create_presentation : integer
end;

/*****************************************/
/**** Dial_2_boutons : les methodes ****/

method body init (s : string, llab : list(list(Component)), nom_bouton1 : string,
                  nom_bouton2 : string) in class Dial_2_boutons {
     self -> nom_bouton1 = nom_bouton1; self -> nom_bouton2 = nom_bouton2;
     self -> Dialog_box@init(s,llab); };

method body create_presentation : integer in class Dial_2_boutons {
Lk_resource res[8];
o2_string ch;
```

```
o2 integer p;
Mask masks;
char nomC1[20], nomC2[20];          /* redefinition des deux boutons */
      res[0].name = "penType"; res[0].value = "STRING";
      strcpy(nomC1, self -> nom_bouton1);
      res[1].name = "penString"; res[1].value = nomC1;
      res[2].name = "lockedPenType"; res[2].value = "STRING";
      res[3].name = "lockedPenString"; res[3].value = nomC1;
      res[4].name = "eraserType"; res[4].value = "STRING";
      strcpy(nomC2, self -> nom_bouton2);
      res[5].name = "eraserString"; res[5].value = nomC2;
      res[6].name = "lockedEraserType"; res[6].value = "STRING";
      res[7].name = "lockedEraserString"; res[7].value = nomC2;

      masks = lk_method ("get_mask");
      ch = self -> ed_name;
      p=lk_present(self,lk_specific(ch,0,0,"dialog",1,&masks),"dialog",8,res);
      return p;
};

/*************************************/
/**** Dial_1_bouton : les methodes ****/

method body init (s : string, llab : list(list(Component)),
                  nom_bouton : string) in class Dial_1_bouton {
      self -> nom_bouton = nom_bouton;
      self -> Dialog_box@init(s,llab); };

method body create_presentation : integer in class Dial_1_bouton {
Lk_resource res[5];
o2 string ch;
o2 integer p;
Mask masks;
char nomC[20];          /* efface un bouton et redéfini l'autre */
      res[0].name = "penType"; res[0].value = "STRING";
      strcpy(nomC, self -> nom_bouton);
      res[1].name = "penString"; res[1].value = nomC;
      res[2].name = "lockedPenType"; res[2].value = "STRING";
      res[3].name = "lockedPenString"; res[3].value = nomC;
      res[4].name = "eraser"; res[4].value = "INVISIBLE";

      masks = lk_method ("get_mask");
      ch = self -> ed_name;
      p=lk_present(self,lk_specific(ch,0,0,"dialog",1,&masks),"dialog",5,res);
      return p;
};

/*****************************************/
/**** Dial_sans_bouton : les methodes ****/

method body init (s : string, llab : list(list(Component)))
           in class Dial_sans_bouton {
      self -> Dialog_box@init(s,llab); };

method body create_presentation : integer in class Dial_sans_bouton {
Lk_resource res[2];
o2 string ch;
o2 integer p;
Mask masks;             /* efface les deux boutons */
      res[0].name = "pen"; res[0].value = "INVISIBLE";
      res[1].name = "eraser"; res[1].value = "INVISIBLE";

      masks = lk_method ("get_mask");
      ch = self -> ed_name;
      p=lk_present(self,lk_specific(ch,0,0,"dialog",1,&masks),"dialog",2,res);
      return p;
};
```

6.5 Traduction des classes interactives «Menus»

Cette section décrit les classes O2 représentant la classe initiale et les menus de l'application.

6.5.1 Définition de la classe initiale, des menus et de l'application

```
/* Classe_initiale_applick : classe initiale de l'application applick */

class Classe_initiale_applick
inherit Controle_generic
type tuple (     f : Menu_fichier,
                 i : Menu_inscriptions,
                 d : Menu_dossards,
                 c : Menu_courses,
                 r : Menu_categories)
method public lancer_programme
end;

class Menu_fichier
inherit Menu_generic
method      public init_liste_boutons,
            public nouvelle_course,
            public quitter
end;

class Menu_inscriptions
inherit Menu_generic
method      public init_liste_boutons,
            public creer,
            public par_categorie,
            public liste_des_inscrits,
            public annuler_les_inscrits
end;

class Menu_dossards
inherit Menu_generic
method      public init_liste_boutons,
            public attribuer,
            public liste_des_dossards,
            public annuler_dossards
end;

class Menu_courses
inherit Menu_generic
type tuple (    traite_resultat : Controle_saisie_resu,
            affichage : Controle_affichage)
method      public init_liste_boutons,
            public lancer,
            public arreter,
            public resultat,
            public fin_resultat,
            public afficher,
            public maj_affichage (cat : Categorie),
            public fin_afficher,
            public annuler_les_resultats
end;

class Menu_categories
inherit Menu_generic
method      public init_liste_boutons,
            public creer,
            public liste_des_categories
end;

/* Application applick : application gestion de competitions sportives */

application applick
     program dashboard
end;

program body dashboard in application applick {
```

```
o2 Classe_initiale_applick racine_interface;
      racine_interface = new Classe_initiale_applick;
      racine_interface -> lancer_programme; };
```

6.5.2 Classe Initiale

```
/*************************************************************************/
/* Class Classe_initiale_applick : Racine de l'interface, coordonne les menus*/
/*************************************************************************/

method body lancer_programme in class Classe_initiale_applick {
      lk_prologue(0,"AppliCK","AppliCK",0,0); /* Initialisation de O2_LOOKS */

      self -> f = new Menu_fichier; /* Creation des menus */
      self -> i = new Menu_inscriptions;
      self -> d = new Menu_dossards;
      self -> c = new Menu_courses;
      self -> r = new Menu_categories;

      self -> f -> init_menu (self,10,10); /* Initialisation des menus */
      self -> i -> init_menu (self,120,10);
      self -> d -> init_menu (self,260,10);
      self -> c -> init_menu (self,375,10);
      self -> r -> init_menu (self,485,10);

                  /* initialisation de l'objet nomme charge de */
      transaction;      /* regrouper les objets partiellement temporaires */
            Objets_partiellement_temporaires = new Ens_O_partiellement_temporaire;
            Objets_partiellement_temporaires -> initialise;
      validate;

      /* Transmet controle a l'utilisateur : les actions de l'util. sont traitees */
      /* par les methodes des menus et des fenetres qui lui sont presentes */
      self -> f -> passe_controle;

      /* Seule la commande QUITTER du menu FICHIER "debloque" l'application */
      /* et rend la main au programme principal. La suite de cette methode  */
      /* doit donc effectuer les operations de fin d'application */

      self -> f -> efface; /* Effacement des menus */
      self -> i -> efface;
      self -> d -> efface;
      self -> c -> efface;
      self -> r -> efface;

      transaction; /* Effacement des objets temporaires */
            Objets_partiellement_temporaires -> fin_application;
      validate;

      lk_epilogue(); /* Cloture de O2_LOOKS */
};
```

6.5.3 Corps des méthodes des menus

```
/* Class Menu_fichier : body des commandes du menu Fichier */

method body init_liste_boutons in class Menu_fichier {
o2 tuple(nom: string, activable: boolean) b;
      b.nom = "nouvelle_course" ; b.activable = true;
      self -> liste_boutons = list(b);
      b.nom = "quitter" ; b.activable = true;
      self -> liste_boutons += list(b); };

method body nouvelle_course in class Menu_fichier {
o2 Controle_nouvelle_course ac;
      ac = new Controle_nouvelle_course;
      ac -> traite_commande; };

method body quitter in class Menu_fichier {
      self -> reprend_le_controle; };
```

```
/* Classe Menu_categories : body des commandes du sous-menu */

method body init_liste_boutons in class Menu_categories {
o2 tuple(nom: string, activable: boolean) b;
      b.nom = "creer" ; b.activable = true;
      self -> liste_boutons = list(b);
      b.nom = "liste_des_categories" ; b.activable = true;
      self -> liste_boutons += list(b); };

method body creer in class Menu_categories {
o2 Controle_creer_categorie a;
      a = new Controle_creer_categorie;
      a->traite_commande; };


/* Class Menu_inscriptions : body des commandes du sous-menu */

method body init_liste_boutons in class Menu_inscriptions {
o2 tuple(nom: string, activable: boolean) b;
      b.nom = "creer" ; b.activable = true;
      self -> liste_boutons = list(b);
      b.nom = "liste_des_inscrits" ; b.activable = true;
      self -> liste_boutons += list(b);
      b.nom = "annuler_les_inscrits" ; b.activable = true;
      self -> liste_boutons += list(b); };

method body creer in class Menu_inscriptions {
o2 Controle_creer_insc a;
      a = new Controle_creer_insc;
      a->traite_commande; };

method body liste_des_inscrits in class Menu_inscriptions {
o2 Controle_liste_insc a;
      a = new Controle_liste_insc;
      a->traite_commande; };

method body annuler_les_inscrits in class Menu_inscriptions {
o2 Controle_annul_insc a;
      a = new Controle_annul_insc;
      a->traite_commande; };


/* Dossard : body des methodes des commandes du sous-menu */

method body init_liste_boutons in class Menu_dossards {
o2 tuple(nom: string, activable: boolean) b;
      b.nom = "attribuer" ; b.activable = true;
      self -> liste_boutons = list(b);
      b.nom = "liste_des_dossards" ; b.activable = true;
      self -> liste_boutons += list(b);
      b.nom = "annuler_dossards" ; b.activable = true;
      self -> liste_boutons += list(b); };

method body attribuer in class Menu_dossards {
      transaction;
            competition_courante -> promotion;
            competition_courante->partie_temporaire->attribuer_les_dossards;
      validate; };

method body liste_des_dossards in class Menu_dossards {
o2 Controle_liste_doss a;
      a = new Controle_liste_doss;
      a->traite_commande; };


/* Menu_course : body des commandes du sous-menu */

method body init_liste_boutons in class Menu_courses {
o2 tuple(nom: string, activable: boolean) b;
      b.nom = "lancer" ; b.activable = true;
      self -> liste_boutons = list(b);
```

```
        b.nom = "arreter" ; b.activable = false;
        self -> liste_boutons += list(b);
        b.nom = "resultat" ; b.activable = false;
        self -> liste_boutons += list(b);
        b.nom = "afficher" ; b.activable = false;
        self -> liste_boutons += list(b);
        b.nom = "annuler_les_resultats" ; b.activable = true;
        self -> liste_boutons += list(b); };

method body lancer in class Menu_courses {
o2 tuple(nom: string, activable: boolean) b;
      self -> disable_method("lancer");
      self -> disable_method("annuler_les_resultats");
      self -> enable_method("arreter");
      self -> enable_method("resultat");
      self -> enable_method("afficher");
      self -> affichage = nil;
      transaction;
            competition_courante -> promotion;
            competition_courante -> partie_temporaire -> lancer_course;
      validate; };

method body arreter in class Menu_courses {
o2 tuple(nom: string, activable: boolean) b;
      if (self->traite_resultat != nil) { self->traite_resultat->effacer_tout; };
      if (self -> affichage != nil) { self -> affichage -> effacer_tout; };
      self -> fin_afficher;
      self -> disable_method("resultat");
      self -> disable_method("arreter");
      self -> disable_method("afficher");
      self -> enable_method("lancer");
      self -> enable_method("annuler_les_resultats"); };

method body resultat in class Menu_courses {
o2 tuple(nom: string, activable: boolean) b;
o2 Controle_saisie_resu ac;
      self -> disable_method("resultat");
      ac = new Controle_saisie_resu;
      ac -> traite_commande(self);
      self -> traite_resultat = ac; };

method body fin_resultat in class Menu_courses {
      self -> enable_method ("resultat");
      self -> traite_resultat = nil; };

method body afficher in class Menu_courses {
      if (self -> affichage == nil) {   self -> affichage = new Controle_affichage;
                                        self -> affichage -> initialise(self); };
      self -> disable_method("afficher");
      self -> affichage -> traite_commande; };

method body maj_affichage (cat : Categorie) in class Menu_courses {
      if (self->affichage != nil) { self->affichage->maj_affichage (cat); }; };

method body fin_afficher in class Menu_courses {
      self -> enable_method("afficher");
      self -> affichage = nil; };
```

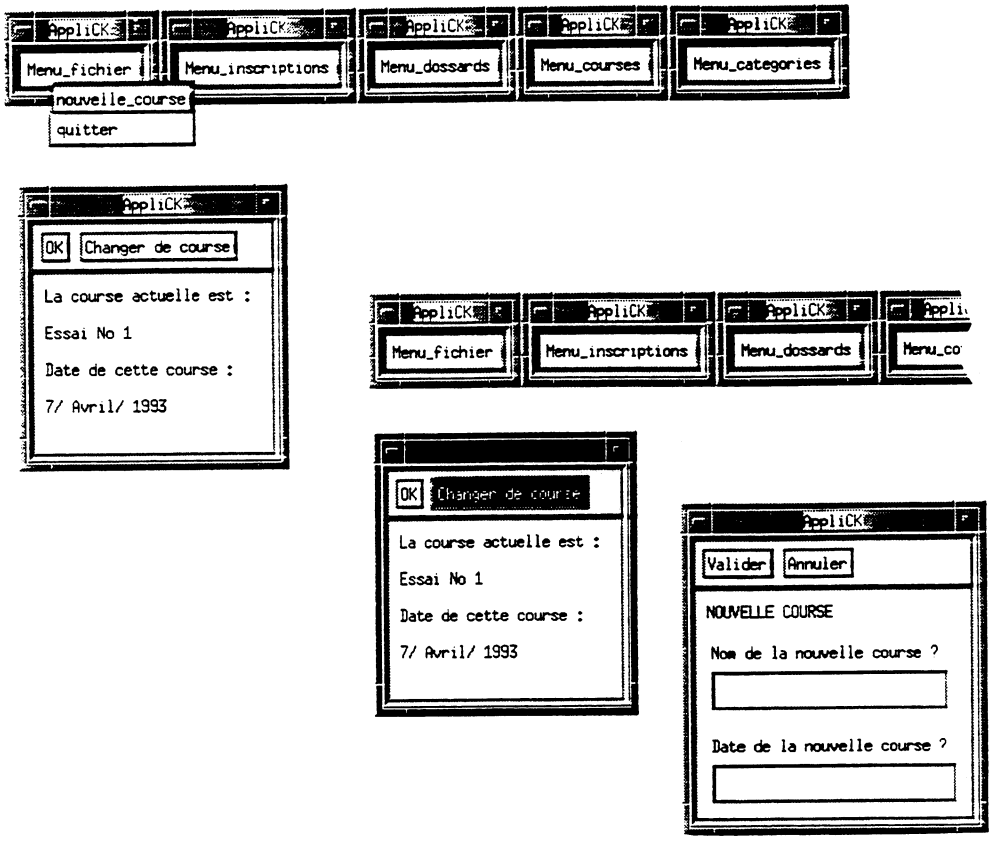
6.6 Traduction de quelques commandes

Cette section présente la réalisation de quelques commandes ainsi que l'image partielle de l'écran obtenu avec chacune de ces commandes.

6.6.1 Commande nouvelle course

La figure ci-dessous présente les fenêtres affichées lors de l'activation de la commande «nouvelle course», suivie par l'activation du bouton «Changer de course» dans la

première fenêtre.

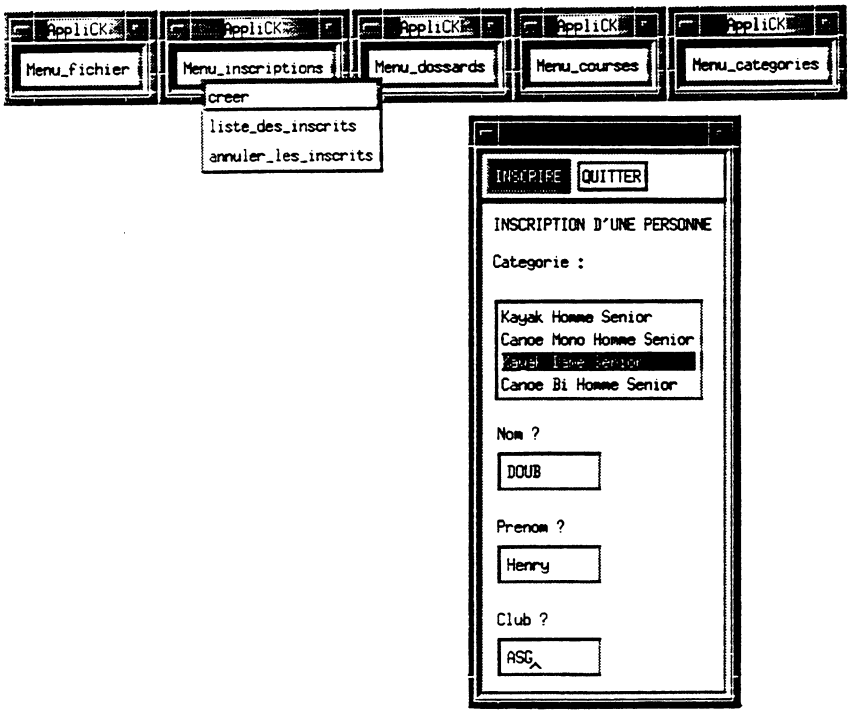



```
/* Class Controle_nouvelle_course : Controle de la commande "nouvelle_course" */

class Controle_nouvelle_course
inherit Controle_generic
method      public traite_commande
end;

/* Classe Present_nouvelle_course : Presentation de la commande "nouvelle_course" */

class Present_nouvelle_course
inherit Present_generic
method      public course_valide(c : tuple (nom : string, date : string)) : boolean,
            public nouvelle_course:tuple(valide:boolean,nom:string,date:string)
end;


/**** Controle_nouvelle_course : les methodes ****/

method body traite_commande in class Controle_nouvelle_course {
o2 Present_nouvelle_course p;
o2 boolean ok, confirme;
o2 tuple ( nom : string, date : string) course;
o2 tuple ( valide : boolean,nom : string, date : string) reponse;
      p = new Present_nouvelle_course; /* creation de la facette presentation */

      if (competition_courante == nil) {      /* recherche de la course actuelle */
            transaction;
                  competition_courante = new Competition;
                  competition_courante -> initialiser ("","","");
            validate;
      };
      course.nom = competition_courante -> nom;
```

```
        course.date = competition_courante -> date;
        ok  = p -> course_valide (course); /* est-ce la bonne course ? */
        if (ok == false) { /* l'utilisateur veut changer de competition */
              reponse = p -> nouvelle_course;
              if ( reponse.valide == true) {
                    transaction;
                    competition_courante->initialiser (reponse.nom,reponse.date,"");
                    validate;
  };    };    };


  /**** Present_nouvelle_course : les methodes ****/

  method body course_valide (c : tuple (nom : string, date : string)) : boolean
          in class Present_nouvelle_course {
  o2 Label l1,l2,l3,l4,l5;
  o2 Dial_2_boutons dial;
  int p, reponse;
  o2 list(list(Label)) llab;
              /* creation des composants de la boite de dialogue*/
        l1 = new Label("l1","La course actuelle est : ");
        l2 = new Label("l2",c.nom);
        l3 = new Label("l3","Date de cette course : ");
        l4 = new Label("l4",c.date);
        l5 = new Label("l5","");
        llab = list(list(l1),list(l2),list(l3),list(l4),list(l5));

              /* creation de la fenetre proprement dite */
        dial = new Dial_2_boutons("Initialisation",llab,"OK","Changer de course");

        p = dial -> create_presentation; /* affichage de la boite */
        lk_map(p, COORDINATE, 0, 0,200,200);

        reponse = lk_wait(p); /* attente et consultation de la reponse */
        lk_consult(p, dial);
        lk_delete_presentation(p);

        if (reponse == SAVE){return (true);
        } else {          return (false); };
  };

  method body nouvelle_course : tuple (valide : boolean, nom : string, date : string)
          in class Present_nouvelle_course {
  o2 Label lab1;
  o2 Prompt pr1, pr2;
  o2 Dial_2_boutons dial;
  o2 list(list(Component)) llab;
  o2 tuple ( valide : boolean, nom : string, date : string) resu;
  int p, reponse;
              /* creation des composants de la boite et de la boite */
        lab1 = new Label("lab1","NOUVELLE COURSE");
        pr1 = new Prompt ("prompt1","Nom de la nouvelle course ? ","");
        pr2 = new Prompt ("prompt2","Date de la nouvelle course ? ","");
        llab = list( (o2 list(Component)) list(lab1));
        llab += list( (o2 list(Component)) list(pr1));
        llab += list( (o2 list(Component)) list(pr2));
        dial=new Dial_2_boutons("Init3",llab,"Valider","Annuler");

        p = dial -> create_presentation; /* affichage et consultation de la boite */
        lk_map(p, COORDINATE, 0, 0,200,200);
        reponse = lk_wait(p);
        lk_consult(p, dial);
        lk_delete_presentation(p);

        if (reponse == SAVE){       resu.valide = true;
                                    resu.nom = pr1 -> answer;
                                    resu.date = pr2 -> answer;
        } else {resu.valide = false; };
        return resu;
  };
```

6.6.2 Commande «créer une inscription»

La figure suivante présente les fenêtres affichées après l'activation de la commande «creer» de la fenêtre «Menu_inscriptions».




```
/* Class Controle_creer_insc : Controle de la commande "creer inscription" */

class Controle_creer_insc
inherit Controle_generic
method      public traite_commande
end;

/* Present_creer_insc : Presentation de la commande "creer inscription" */

class Present_creer_insc
inherit Present_generic
type tuple (cat_courante : integer,
            categories : list(string))
method      public init_presentation (list_cat : list(string)),
            public demande_insc : tuple (termine : boolean, categ : string,
                        nom : string, prenom : string, club : string)
end;

attribute f_presentation : Present_creer_insc in class Controle_creer_insc;
attribute f_controle : Controle_creer_insc in class Present_creer_insc;


/**** Controle_creer_insc : les methodes ****/

method body traite_commande in class Controle_creer_insc {
o2 tuple (termine : boolean, categ : string, nom : string,
            prenom : string, club : string) reponse;
o2 list(string) list_cat;
o2 Categorie cat;
o2 Inscription insc;
      list_cat = list(); /* recherche des categories de la course */
      for (cat in competition_courante -> categories) {
```

```
            list_cat += list(cat -> nom);};
            /* creation de la facette presentation */
      self -> f_presentation = new Present_creer_insc;
      self -> f_presentation -> init_presentation (list_cat);

      reponse.termine = false; /* boucles permettant plusieurs inscriptions */
      while (reponse.termine == false) {
            reponse = self -> f_presentation -> demande_insc;
            if (reponse.termine == false) { /* inscription du coureur */
                  insc = new Inscription;
                  insc -> initialiser ( reponse.nom, reponse.prenom);
                  transaction;
                        competition_courante -> inscrire (insc, reponse.categ);
                  validate;
};    };    };


/**** Present_creer_insc : les methodes ****/

method body init_presentation (list_cat : list(string)) in class Present_creer_insc {
      self -> cat_courante = 0;
      self -> categories = list_cat; };

method body demande_insc : tuple(termine : boolean, categ : string, nom : string,
                  prenom : string, club : string) in class Present_creer_insc {
o2 Label lab1, lab2;
o2 Single_selection sel;
o2 Prompt pr1, pr2, pr3;
o2 Dial_2_boutons dial;
o2 list(list(Component)) llab;
o2 tuple ( termine : boolean, categ : string, nom : string,
            prenom : string, club : string) resu;
o2 integer p, reponse;
            /* creation des composants de la boite et de la boite */
      lab1 = new Label("lab1","INSCRIPTION D'UNE PERSONNE");
      lab2 = new Label("lab2","Categorie :");
      sel = new Single_selection ("sel","", self -> categories, self -> cat_courante);
      pr1 = new Prompt ("prompt1","Nom ? ","");
      pr2 = new Prompt ("prompt2","Prenom ? ","");
      pr3 = new Prompt ("prompt3","Club ? ","");
      llab = list( (o2 list(Component)) list(lab1));
      llab += list( (o2 list(Component)) list(lab2));
      llab += list( (o2 list(Component)) list(sel));
      llab += list( (o2 list(Component)) list(pr1));
      llab += list( (o2 list(Component)) list(pr2));
      llab += list( (o2 list(Component)) list(pr3));

      dial = new Dial_2_boutons("Insc",llab, "INSCRIRE", "QUITTER");

      p = dial -> create_presentation;
      lk_map(p, COORDINATE, 0, 0,200,200);
      reponse = lk_wait(p);
      lk_consult(p, dial); /* recuperation de la reponse */
      lk_delete_presentation(p);

      if (reponse == SAVE){  /*l'user a active le premier bouton */
            resu.termine = false;
            resu.categ = self -> categories [sel -> answer];
            self -> cat_courante = sel -> answer;
            resu.nom = pr1 -> answer;
            resu.prenom = pr2 -> answer;
            resu.club = pr3 -> answer;
      } else {resu.termine = true; };
      return resu;
};
```

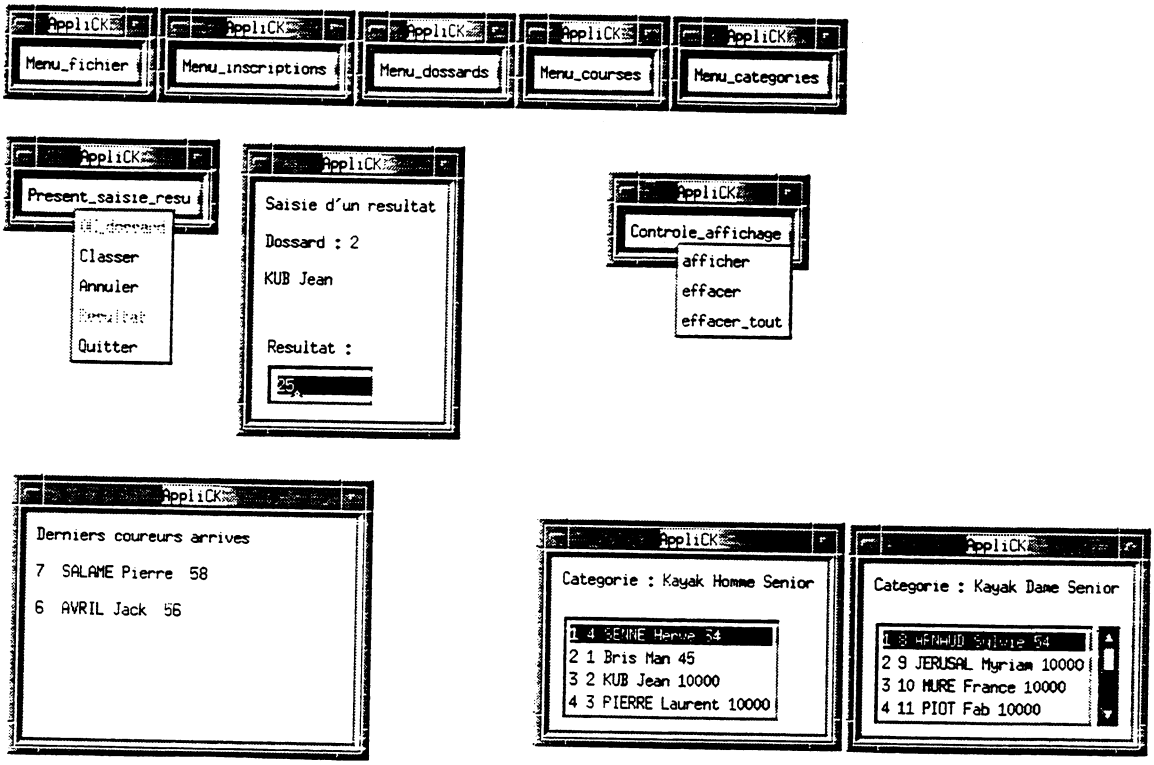
6.6.3 Commande «liste de départ»

```
/* Class Controle_liste_insc : Controle de la commande "annuler inscription" */

class Controle_liste_doss
```

```
inherit Controle_generic
method      public traite_commande
end;

/* Present_liste_doss : Presentation de la commande "annuler inscription" */

class Present_liste_doss
inherit Present_generic
method      public presente_liste( list_insc : list(string))
end;

attribute f_presentation : Present_liste_doss in class Controle_liste_doss;
attribute f_controle : Controle_liste_doss in class Present_liste_doss;


/**** Controle_liste_doss : les methodes ****/

method body traite_commande in class Controle_liste_doss {
o2 Inscription i;
o2 string s, sdos;
o2 list(string) ls;
      transaction;
            competition_courante -> promotion;
            competition_courante -> partie_temporaire -> init_list_dos;
      validate;
      ls = list();
      for (i in competition_courante -> partie_temporaire -> liste_dos) {
            if (i != nil ) {       sdos = int_to_str(i -> dossard);
                                   s = sdos + " " + i -> nom + " " + i -> prenom;
                                   ls += list(s); };
      };
      self -> f_presentation = new Present_liste_doss;
      self -> f_presentation -> presente_liste(ls);
};


/**** Present_liste_doss : les methodes ****/

method body presente_liste( list_insc:  list(string)) in class Present_liste_doss {
o2 Label l1;
o2 Single_selection sel;
o2 Dial_1_bouton dial;
int p, reponse;
o2 list(list(Component)) llab;
      l1 = new Label("l1","liste de depart");
      sel = new Single_selection ("sel","", list_insc, 0);
      llab = list( (o2 list(Component)) list(l1));
      llab += list( (o2 list(Component)) list(sel));

      dial=new Dial_1_bouton ("Initialisation", llab, "QUITTER");
      p = dial -> create_presentation;
      lk_map(p, COORDINATE, 0, 0,200,200);
      reponse = lk_wait(p);
      lk_delete_presentation(p);
};
```

6.6.4 Commande «saisie d'un résultat»

La figure suivante présente les fenêtres affichées après l'activation des commandes «lancer» puis «resultat» et «afficher» de la fenêtre «Menu_courses». Les deux fenêtres «Present_saisie_resu» et «Saisie d'un resultat» représentent les boutons et les informations utilisées lors de la saisie d'un résultat. La fenêtre «Derniers coureurs arrivés» présente le résultat des derniers coureurs arrivés. Les trois fenêtres en bas à droite montrent les classements de différentes catégories.




```
/* Class Controle_saisie_resu : Controle de la commande "resultat" */

class Controle_saisie_resu
inherit Controle_generic
type tuple (     menu_pere : Menu_courses,
                 derniers_arrives : Present_derniers_arrives,
                 insc : Inscription)
method      public traite_commande ( pere : Menu_courses),
            public insc_de_dos(dos:integer):tuple(valide:boolean, nom:string,
                                    prenom:string, club:string, resu:integer),
            public resu_et_place ( resu : integer ) : integer,
            public quitter,
            public effacer_tout
end;

/* Present_saisie_resu : Presentation de la commande resultat */

class Present_saisie_resu
inherit Multiple_boutons_generic
type tuple (     info : Dial_sans_bouton,
                 p_info : integer,
                 coureur : tuple(dos : integer, nom : string, prenom : string,
                                 club : string, resu : integer))
method      public init_presentation (controle : Controle_saisie_resu),
            public init_liste_boutons,
            public OK_dossard,
            public Classer,
            public Resultat,
            public Annuler,
            public Quitter,
            public effacer_tout,
            private affiche_dial (llab : list(list(Component))),
            private dial_doss_seul : list(list(Component)),
            private dial_n_p_c : list(list(Component)),
            private dial_definitif (place : integer) : list(list(Component))
end;
```

```
attribute f_presentation : Present_saisie_resu in class Controle_saisie_resu;
attribute f_controle : Controle_saisie_resu in class Present_saisie_resu;

/* Present_derniers_arrives  : Presentation des derniers coureurs classes */

class Present_derniers_arrives
type tuple(p_info : integer,
           derniers : list(tuple(dos : string, n_p_c : string)))
method     public init_presentation,
           private affiche,
           public maj_derniers ( dos : integer, n_p_c : string, resu : integer),
           public effacer
end;


/**** Present_saisie_resu : les methodes ****/

method body init_presentation(controle : Controle_saisie_resu)
       in class Present_saisie_resu {
o2 list(list(Component)) llab;
     self -> f_controle = controle;
     self -> init_liste_boutons; /* afficher les boutons */
     self -> affiche(50, 150);
     self -> reactive_liste_boutons;
     llab = self -> dial_doss_seul; /* afficher les informations */
     self -> affiche_dial(llab); };

method body init_liste_boutons in class Present_saisie_resu {
o2 tuple(nom: string, activable: boolean) b;
     b.nom = "OK_dossard" ; b.activable = true;
     self -> liste_boutons = list(b);
     b.nom = "Classer" ; b.activable = false;
     self -> liste_boutons += list(b);
     b.nom = "Annuler" ; b.activable = false;
     self -> liste_boutons += list(b);
     b.nom = "Resultat" ; b.activable = false;
     self -> liste_boutons += list(b);
     b.nom = "Quitter" ; b.activable = true;
     self -> liste_boutons += list(b); };

method body affiche_dial (llab : list(list(Component)))
          in class Present_saisie_resu {
o2 Dial_sans_bouton dial;
     dial = new Dial_sans_bouton ("Categories",llab); /*creation de la fenetr*/
     self -> info = dial;
     self -> p_info = dial -> create_presentation; /*affichage de la fenetre*/
     lk_map(self -> p_info, COORDINATE,0,0,self-> x + 160, self -> y); };

method body dial_doss_seul : list(list(Component)) in  class Present_saisie_resu {
o2 Label l1;
o2 Prompt pr1;
o2 list(list(Component)) llab;
          /* creation des composants de la fenetre affichee */
     l1 = new Label("l1","Saisie d'un resultat");
     pr1 = new Prompt ("prompt1","Dossard du coureur : ","");
     llab = list( (o2 list(Component)) list(l1));
     llab += list( (o2 list(Component)) list(pr1));
     return llab; };

method body OK_dossard in class Present_saisie_resu {
o2 string dos_string;
o2 tuple ( valide : boolean, valeur : integer) dos_saisi;
o2 tuple(valide : boolean,nom : string, prenom : string,
          club : string, resu : integer) coureur;
o2 list(list(Component)) llab;
o2 Prompt pr;
     lk_consult(self -> p_info, self -> info); /* recuperation des parametres */
     pr = ((o2 Prompt) self -> info -> element (1,0));
     dos_string = pr -> answer;
     dos_saisi = str_to_int (dos_string);
     if (dos_saisi.valide == true) { /* recherche du coureur*/
          coureur = self -> f_controle -> insc_de_dos(dos_saisi.valeur);
          if (coureur.valide) {
```

```
                    self -> coureur.dos = dos_saisi.valeur;
                    self -> coureur.nom = coureur.nom;
                    self -> coureur.prenom = coureur.prenom;
                    self -> coureur.club = coureur.club;
                    self -> coureur.resu = coureur.resu;
                    llab = self -> dial_n_p_c;
                    lk_delete_presentation (self -> p_info);
                    self -> affiche_dial(llab);
                    self -> enable_method("Classer");
                    self -> enable_method("Annuler");
                    self -> disable_method("OK_dossard");
};    };    };

method body dial_n_p_c : list(list(Component)) in  class Present_saisie_resu {
o2 string s;
o2 Label l1,l2,l3,l4;
o2 Prompt pr;
o2 list(list(Component)) llab;
          /* creation des composants de la fenetre affichee */
      l1 = new Label("l1","Saisie d'un resultat");
      s = int_to_str (self -> coureur.dos);
      l2 = new Label("l2","Dossard : " + s);
      l3 = new Label("l3",self -> coureur.nom + " " + self -> coureur.prenom);
      l4 = new Label("l4",self -> coureur.club);
      s = int_to_str (self -> coureur.resu);
      pr = new Prompt ("pr","Resultat : ",s);

      llab = list( (o2 list(Component)) list(l1));
      llab += list( (o2 list(Component)) list(l2));
      llab += list( (o2 list(Component)) list(l3));
      llab += list( (o2 list(Component)) list(l4));
      llab += list( (o2 list(Component)) list(pr));
      return llab;
};

method body Classer in class Present_saisie_resu {
o2 string resu_string;
o2 tuple ( valide : boolean, valeur : integer) resu_saisi;
o2 integer place;
o2 list(list(Component)) llab;
o2 Prompt pr;
      lk_consult(self -> p_info, self -> info); /* recuperation des parametres */
      pr = ((o2 Prompt) self -> info -> element (4,0));
      resu_string = pr -> answer;
      resu_saisi = str_to_int (resu_string);
      if (resu_saisi.valide == true) { /* maj du resultat et demande de la place */
            place = self -> f_controle -> resu_et_place(resu_saisi.valeur);
            self -> coureur.resu = resu_saisi.valeur;
            llab = self -> dial_definitif(place);
            lk_delete_presentation (self -> p_info);
            self -> affiche_dial(llab);
      };
      self -> enable_method("Resultat");
      self -> disable_method("Classer");
};

method body dial_definitif (place : integer) : list(list(Component))
          in class Present_saisie_resu {
o2 string s;
o2 Label l1,l2,l3,l4,l5,l6;
o2 list(list(Component)) llab;
      l1 = new Label("l1","Saisie d'un resultat"); /* creation des composants*/
      s = int_to_str (self -> coureur.dos);
      l2 = new Label("l2","Dossard : " + s);
      l3 = new Label("l3",self -> coureur.nom + " " + self -> coureur.prenom);
      l4 = new Label("l4",self -> coureur.club);
      s = int_to_str (self -> coureur.resu);
      l5 = new Label ("l5","Resultat : "+ s);
      s = int_to_str (place);
      l6 = new Label ("l5","Place : "+ s);
      llab = list( (o2 list(Component)) list(l1));
      llab += list( (o2 list(Component)) list(l2));
      llab += list( (o2 list(Component)) list(l3));
```

```
      llab += list( (o2 list(Component)) list(l4));
      llab += list( (o2 list(Component)) list(l5));
      llab += list( (o2 list(Component)) list(l6));
      return llab;
};

method body Annuler in class Present_saisie_resu {
o2 list(list(Component)) llab;
      lk_delete_presentation (self -> p_info);
      llab = self -> dial_doss_seul;
      self -> affiche_dial(llab);
      self -> reactive_liste_boutons; };

method body Resultat in class Present_saisie_resu {
o2 list(list(Component)) llab;
      lk_delete_presentation (self -> p_info);
      llab = self -> dial_doss_seul;
      self -> affiche_dial(llab);
      self -> reactive_liste_boutons; };

method body Quitter in class Present_saisie_resu {
      self -> effacer_tout;
      self -> f_controle -> quitter; };

method body effacer_tout in class Present_saisie_resu {
      self -> efface;
      lk_delete_presentation (self -> p_info); };


/**** Present_derniers_arrives : les methodes ****/

method body init_presentation in class Present_derniers_arrives {
int i;
o2 tuple(dos : string, n_p_c : string) c;
      c.dos = ""; c.n_p_c ="                                          ";
      self -> derniers = list();
      for (i=0;i<5;i++) { self -> derniers += list(c);};
      self -> affiche; };

method body maj_derniers ( dos : integer, n_p_c : string, resu : integer)
            in class Present_derniers_arrives {
o2 string s, res;
o2 tuple(dos : string, n_p_c : string) c;
o2 integer i;
o2 boolean trouve;
      s = int_to_str (dos);
      res = int_to_str(resu);
      trouve = false;
      for (i=0; i<5; i++) {
            if (self -> derniers[i].dos == s) {
                  trouve = true;
                  self -> derniers[i].n_p_c = n_p_c + "  " + res; };
      };
      if (trouve == false) {
            for (i=4; i>0; i--) { self -> derniers [i] = self -> derniers [i-1]; };
            self -> derniers[0].dos = s;
            self -> derniers[0].n_p_c = n_p_c + "  " + res;
      };
      lk_delete_presentation (self -> p_info);
      self -> affiche;
};

method body affiche in class Present_derniers_arrives {
o2 tuple(dos : string, n_p_c : string) c;
o2 Label l;
o2 list(list(Component)) llab;
o2 Dial_sans_bouton dial;
            /* creation des composants de la fenetre affichee */
      l = new Label("l1","Derniers coureurs arrives");
      llab = list( (o2 list(Component)) list(l));
      for (c in self -> derniers) {     l = new Label("l",c.dos + "  " + c.n_p_c);
                                        llab += list( (o2 list(Component)) list(l));
      };
```

```
      dial = new Dial_sans_bouton ("Categories",llab); /* creation de la fenetre */
      self -> p_info = dial -> create_presentation; /* affichage de la fenetre */
      lk_map(self -> p_info, COORDINATE,0,0,50,450);
};

method body effacer in class Present_derniers_arrives {
           lk_delete_presentation (self -> p_info); };


/**** Controle_saisie_resu : les methodes ****/

/** traite_commande : met en place les elements du dialogue charge de gerer la **/
/**         commande de creation d'une categorie */

method body traite_commande (pere : Menu_courses) in class Controle_saisie_resu {
      self -> menu_pere = pere;
      self -> f_presentation = new Present_saisie_resu;
      self -> f_presentation -> init_presentation (self);
      self -> derniers_arrives = new Present_derniers_arrives;
      self -> derniers_arrives -> init_presentation; };

method body insc_de_dos (dos : integer) :
      tuple(valide : boolean, nom : string, prenom : string, club : string,
           resu : integer) in class Controle_saisie_resu {
o2 tuple(  valide : boolean, nom : string,
           prenom : string, club : string, resu : integer) resu;

      self->insc=competition_courante->partie_temporaire->insc_de_dos(dos);
      if (self -> insc == nil) {   resu.valide = false;
      } else {    resu.valide = true;
                  resu.nom = self -> insc -> nom;
                  resu.prenom = self -> insc -> prenom;
                  resu.club = "";
                  resu.resu = self -> insc -> resultat;
      };
      return resu;
};

method body resu_et_place ( resu : integer ) : integer
           in class Controle_saisie_resu {
o2 Categorie cat;
o2 integer place;
o2 string s;
o2 integer i1, i2;
      cat = competition_courante -> cat_de_insc(self -> insc);
      transaction;
           self -> insc -> affecte_resultat(resu);
           place = cat -> partie_temporaire -> classer(self -> insc);
      validate;

      self -> menu_pere -> maj_affichage(cat);
      s = self -> insc -> nom + " " + self -> insc -> prenom;
      i1 = self->insc->dossard; i2 = self->insc->resultat;
      self->derniers_arrives->maj_derniers(i1, s, i2);
      return place;
};

method body quitter in class Controle_saisie_resu {
      self -> derniers_arrives -> effacer;
      self -> menu_pere -> fin_resultat; };

method body effacer_tout in class Controle_saisie_resu {
      self -> derniers_arrives -> effacer;
      self -> f_presentation -> effacer_tout; };
```

6.6.5 Commande «Affichage d'une catégorie»

```
/* Class Controle_affichage : Controle de la commande "afficher" */

class Controle_affichage
inherit Multiple_boutons_generic
```

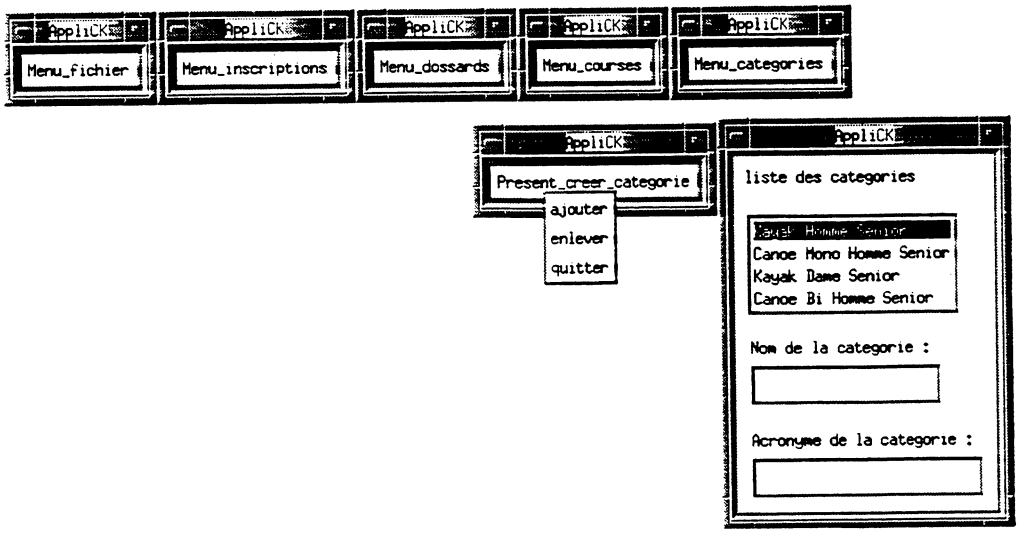
```
type tuple (      menu_pere : Menu_courses,
                  ff_controle : Present_affichage,
                  num_place : set(integer),
                  aff : set(tuple(      cat : Categorie, num_place : integer,
                                        resent : Present_affichage)))
method      public initialise(pere : Menu_courses),
            public init_liste_boutons,
            public traite_commande,
            public afficher,
            public effacer,
            public effacer_tout,
            public quelle_cat (message : string, liste : list(string)) : string,
            public affiche_cat (cat : Categorie),
            public maj_affichage (cat : Categorie)
end;

/* Present_affichage : Presentation de la commande "afficher" */

class Present_affichage
inherit Present_generic
type tuple (p_info : integer)
method      public affiche_cat(nom : string, num_place : integer, cl : list(
                  tuple ( p : integer, d : integer, npc : string, r : integer))),
            public effacer
end;


/**** Controle_affichage : les methodes ****/

method body initialise (pere : Menu_courses) in class Controle_affichage {
      self -> menu_pere = pere;
      self -> aff = set();
      self -> num_place = set();
      self -> init_liste_boutons;
      self -> affiche(500, 150);
      self -> reactive_liste_boutons; };

method body init_liste_boutons in class Controle_affichage {
o2 tuple(nom: string, activable: boolean) b;
      b.nom = "afficher" ; b.activable = true;
      self -> liste_boutons = list(b);
      b.nom = "effacer" ; b.activable = true;
      self -> liste_boutons += list(b);
      b.nom = "effacer_tout" ; b.activable = true;
      self -> liste_boutons += list(b); };

method body traite_commande in class Controle_affichage {
o2 tuple (cat : Categorie, num_place : integer, present : Present_affichage) i;
o2 string selection, nc;
o2 list(string) list_cat;
o2 set(string) list_cat_aff;
o2 Categorie cat;
      list_cat_aff = set();
      for (i in self -> aff) { list_cat_aff += set(i.cat -> nom);};
      list_cat = list();/* recherche des categories */
      for (cat in competition_courante -> categories) {
            nc = cat -> nom;
            if ((nc in list_cat_aff) == false) {list_cat += list(nc); };
      };
      selection = self -> quelle_cat ("Categorie a afficher : ", list_cat);
      if (selection != "annuler") {
            self -> affiche_cat (competition_courante -> cat_de_nom(selection));
};    };

method body quelle_cat (message : string,liste : list(string)) : string
            in class Controle_affichage {
o2 Label lab;
o2 Single_selection sel;
o2 Dial_2_boutons dial;
o2 list(list(Component)) llab;
o2 integer reponse, p;
      lab = new Label("lab",message);/* creation du dialogue */
      sel = new Single_selection ("sel","", liste, 0);
```

```
        llab = list( (o2 list(Component)) list(lab));
        llab += list( (o2 list(Component)) list(sel));
        dial = new Dial_2_boutons("Cat",llab, "VALIDER", "ANNULER");

        p = dial -> create_presentation; /* affichage et consultation du dial */
        lk_map(p, COORDINATE, 0, 0,500,250);
        reponse = lk_wait(p);
        lk_consult(p, dial); /* recuperation de la reponse */
        lk_delete_presentation(p);
        if (reponse == SAVE){        /*l'user a active le premier bouton */
                if (liste != list()) {return liste [sel -> answer]; };
        } else {return "annuler";};
};

method body afficher in class Controle_affichage {
      self -> traite_commande; };

method body effacer in class Controle_affichage {
o2 tuple (cat : Categorie, num_place : integer, present : Present_affichage) i;
o2 list(string) list_cat;
o2 string selection;
      list_cat = list();
      for (i in self -> aff) { list_cat += list(i.cat -> nom); };
      selection = self -> quelle_cat ("Categorie a afficher : ", list_cat);
      if (selection != "annuler") {
            for (i in self -> aff) {
                  if (i.cat -> nom == selection) {
                                          i.present -> effacer;
                                          self -> num_place -= set(i.num_place);
                                          self -> aff -= set(i); };
};    };    };

method body affiche_cat (cat : Categorie) in class Controle_affichage {
o2 tuple ( p : integer, d : integer, npc : string, r : integer) insc;
o2 Inscription i;
o2 list(tuple ( p : integer, d : integer, npc : string, r : integer)) ls;
o2 tuple(cat : Categorie, num_place : integer, present : Present_affichage) nouv_cat;
o2 integer place, j, numero;
      numero = count(self -> num_place); /*recherche d'une place pour afficher*/
      for (j = 0; j<count(self -> num_place); j++) {
            if ((j in self -> num_place) == false) { numero = j; };
      };
      self -> num_place += set(numero);
      ls = list();/*creation de la liste a afficher*/
      place = 1;
      for (i in cat -> partie_temporaire -> classement) {
            insc.p = place; place += 1;
            insc.d = i -> dossard;
            insc.npc = i -> nom + " " + i -> prenom;
            insc.r = i -> resultat;
            ls += list(insc);
      };
      nouv_cat.cat = cat;
      nouv_cat.num_place = numero;
      nouv_cat.present = new Present_affichage;
      nouv_cat.present -> affiche_cat(cat -> nom, numero, ls);
      self -> aff += set(nouv_cat);
};

method body effacer_tout in class Controle_affichage {
o2 tuple(cat : Categorie,present : Present_affichage) i;
      for (i in self -> aff) { i.present -> effacer; };
      self -> efface;
      self -> menu_pere -> fin_afficher; };

method body maj_affichage (cat : Categorie) in class Controle_affichage {
o2 tuple (cat : Categorie,num_place : integer, present : Present_affichage) i;
      for (i in self -> aff) {
            if (cat == i.cat) {i.present -> effacer;
                               self -> aff -= set(i);
                               self -> num_place -= set(i.num_place);
                               self -> affiche_cat(i.cat); };
};    };
```

```
/**** Present_affichage : les methodes ****/

method body affiche_cat(nom : string, num_place : integer, cl : list(
          tuple ( p : integer, d : integer, npc : string, r : integer)))
          in class Present_affichage {
o2 tuple ( p : integer, d : integer, npc : string, r : integer) c;
o2 list (string) ll;
o2 string s1, s2, s3;
o2 Label l;
o2 Single_selection sel;
o2 list(list(Component)) llab;
o2 Dial_sans_bouton dial;
o2 integer rang;
     ll = list(); /* creation des composants de la fenetre affichee */
     for (c in cl) {
          s1 = int_to_str(c.p);
          s2 = int_to_str(c.d);
          s3 = int_to_str(c.r);
          ll += list(s1 + " " + s2+ " " + c.npc + " " + s3);
     };
     l = new Label("l1","Categorie : " + nom); /* creation de la fenetre */
     llab = list( (o2 list(Component)) list(l));
     sel = new Single_selection ("sel","", ll, 0);
     llab += list( (o2 list(Component)) list(sel));
     dial = new Dial_sans_bouton ("Categories",llab);
     rang = num_place / 4; /* affichage de la fenetre */
     num_place = num_place % 4;
     self -> p_info = dial -> create_presentation;
     lk_map(self -> p_info, COORDINATE,0,0,450 + num_place*200,400 + rang*250);
};

method body effacer in class Present_affichage {
     lk_delete_presentation(self -> p_info); };
```

6.6.6 Ecrans de création d'une catégorie

La figure suivante présente les fenêtres affichées après l'activation de la commande «creer» de la fenêtre «Menu_categories». Les informations sont affichées dans la fenêtre de droite («liste des catégories») etles boutons sont activables dans la fenêtre «Present_-creer_categorie».



paramètre
<<-/————————> Classe_présentation_gén

Définition des méthodes des classes interactives :

Méthode_applicative

nom
<-/————————> Chaîne

arguments
<<-/————————/->> {réf, dép, prop} × Classe_interactive_gén

résultat
<<-/————————/-> Classe_interactive_gén

précondition
<<-/————————/-> Proposition_logique

postcondition
<<-/————————/-> Proposition_logique

messages
<<-/————————/->> (Classe_applicative × Caractéristique_applicative) ∪ (Classe_interactive × Caractéristique_interactive)

algorithme
<<-/————————/-> Texte

Variable_interactive

nom
<-/————————> Chaîne

nature
<<-/————————> {réf, dép, prop}

classe
<<-/————————> Classe_interactive_gén

Méthode_présentation

nom
<-/————————> Chaîne

postcondition
<<-/————————/-> Proposition_logique

algorithme
<<-/————————/-> Texte