MASTER EN SCIENCES INFORMATIQUES

Méthodes de conception orientées objets : analyse critique et proposition

Vandeloise, Marc-André

Award date:
1992

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix.

Institut d'Informatique.

Année Académique 1991-1992.

MÉTHODES DE CONCEPTION ORIENTÉES OBJETS: ANALYSE CRITIQUE ET PROPOSITION

PAR MARC-ANDRÉ VANDELOISE

Mémoire présenté en vue de l'obtention du diplôme de
Licencié et Maître en Informatique.

REMERCIEMENTS

Je tiens à remercier l'ensemble des personnes qui m'ont aidé et soutenu durant la réalisation de ce travail, et en particulier

Monsieur Jean-Luc Hainaut , promoteur,

Monsieur Nicholas Hills, superviseur durant mon stage chez Digital Equipment Corporation, Ferney-Voltaire (France) ainsi que toute l'équipe au sein de laquelle j'ai été accueilli,

Monsieur Eric Dubois, pour la documentation qu'il m'a procurée,

de même que ma famille pour l'aide apportée durant la rédaction de ce travail.

RÉSUMÉ

Ce travail est consacré aux nouvelles méthodes de conception orientées objets.

Dans un premier temps, il propose de faire le point sur l'approche orientée objets elle-même, tant au niveau des concepts qu'au niveau des modifications qu'elle entraîne dans les cycles de développement d'applications informatiques. Une synthèse de différentes méthodes y est également inclue.

Dans un second temps, il analyse et compare ces différentes méthodes en fonction des concepts utilisés et de la couverture du cycle de développement. Une critique que l'on peut émettre vis-à-vis de ces méthodes est leur inaptitude à être utilisées avec les outils disponibles actuellement.

Suite à cette analyse critique, ce travail propose une méthode qui se veut opérationnelle, ceci au prix d'une introduction d'éléments non conformes à l'approche orientée objets mais permettant le passage vers des applications existantes.

Une étude de cas permet d'appliquer cette nouvelle méthode et d'en montrer le caractère opérationnel.

ABSTRACT

This work is about the new object oriented design methods.

First, a state of the art of the object oriented approach is proposed at the concept level as well as at the modification of the software applications development life cycle. A synthesis of several methods is proposed.

In a second time, a review and an analysis of these object oriented methods are given with respect to the concepts and the coverage of the development process. One of the best aspects is the improvement of the software quality but one of the worst is the method's inability to be used by existing tools during the implementation phase.

Following the analysis, this work defines a method which wants to be operational, this by introducing non-object oriented elements that make easier the use of existing software applications.

Finally, this new method is used in a case study showing its operational nature.

TABLE DES MATIÈRES

TABLE DES FIGURES

INTRODUCTION

Au début des années '80 une petite révolution a eu lieu dans le monde des méthodes de conception. L'approche orientée objets y a fait son apparition.

Différents auteurs, connus pour leurs méthodes de conception structurées, alors utilisées, nous ont proposé soit une adaptation de leur méthode à cette nouvelle approche, soit une méthode inédite.

Le but de ce travail est d'une part de dresser un état de l'art dans le monde de ces nouvelles méthodes de conception et d'autre part de proposer une méthode qui voudrait pallier les défauts des unes et reprendre les avantages des autres.

Nous commencerons par faire un petit historique de cette nouvelle approche dans le monde des méthodes de conception. Nous définirons également ses différents concepts ainsi que les modifications qu'elle apporte dans le cycle de développement d'applications informatiques. Cette première approche nous permettra ensuite de proposer une synthèse de méthodes existantes.

Dans une deuxième partie, nous analyserons et comparerons ces différentes méthodes. Commençant par leurs concepts de base, nous poursuivrons par les modèles utilisés et par les démarches de conception préconisées par leurs auteurs.

Suite à cette analyse, nous proposerons une méthode qui se veut opérationnelle immédiatement. Elle reprendra les avantages des méthodes analysées tout en essayant d'en éviter les défauts.

Nous appliquerons ensuite notre méthode à un cas relativement connu au sein de l'Institut d'Informatique, l'informatisation de la gestion d'un hôpital. Cette application se veut illustratrice et ne couvre que les phases d'analyse et de conception. Nous donnerons néanmoins quelques pistes en ce qui concerne la phase d'implémentation.

Nous terminerons en tirant les conclusions qui s'imposent suite à l'analyse comparative et à la proposition de méthode.

1ÈRE PARTIE:

LES CONCEPTIONS ORIENTÉES OBJETS

Introduction

Dans cette première partie, nous présenterons les méthodes de conception orientées objets d'une manière générale et théorique.

Nous considèrerons tout d'abord l'approche orientée objets. Relativement récente, elle offre une nouvelle vision du monde réel. Cette approche comprend un certain nombre de concepts et de notions assez éloignés des approches que l'on pourrait qualifier de traditionnelles. Nous présenterons un petit historique et donnerons les définitions des concepts de cette nouvelle approche.

Le second chapitre sera consacré aux méthodes de conception. Ces méthodes guident le développeur tout au long du cycle de développement d'une application. Elles fournissent un certain nombre d'étapes par lesquelles doit passer le système que l'on étudie. Ce cycle de développement est différent suivant l'approche choisie. Nous rappelerons donc ce qu'est le cycle de développement traditionnel avant de présenter celui introduit par l'approche orientée objets.

Dans un troisième temps, nous présenterons des méthodes de conception orientées objets proposées par un certain nombre d'auteurs. Ceux-ci sont relativement connus dans le monde des méthodes de conception.

Chapitre 1 :

Historique et définition des concepts.

Introduction

L'utilisation des Systèmes d'Information (SI) est devenue une réalité quotidienne. La gestion des SI, ainsi que leur conception, représentent aujourd'hui un problème majeur pour les organisations.

Nous allons, dans un premier temps, faire un petit historique des méthodes de conception des SI. Nous commencerons par les méthodes cartésiennes apparues au début des années 60. Viendront ensuite les méthodes systémiques des années 70 et enfin l'émergence, durant le seconde moitié des années 80, des méthodes orientées objets.

Dans un second temps, nous donnerons les définitions des concepts de bases de l'approche orientée objets.

Ce chapitre est basé essentiellement sur une communication de Colette Roland et André Flory lors du 20ème Anniversaire des MIAGE, au congrès INFORSID'90 [ROL90].

1.1. Historique

1.1.1. Les méthodes cartésiennes

Au début des années 60 apparaissent les premières méthodes de conception de SI. Ces méthodes, essentiellement basées sur les décompositions hiérarchiques des processus et des flux de données, restent d'application aujourd'hui dans la plupart des équipes de conception. Elles associent une approche fonctionnelle de conception à l'approche Top-Down prônée par le paradigme cartésien. Les fonctions du SI sont décomposées de leur aspect le plus général, l'aspect gestion, à l'aspect le plus spécifique, les opérations indécomposables, élémentaires. La méthode Structured Analysis and Design Technique (SADT) est un exemple de méthode cartésienne.

Ces méthodes analysent et conçoivent le système d'information en se centrant sur ses fonctions. Le SI est donc vu comme un système de traitement de l'information qui fournit des résultats en fonction des données qu'il reçoit. De plus, elles mettent

l'accent sur la modélisation des processus. On éclate un de ceux-ci en plusieurs sous-processus plus facilement analysables et spécifiques. Cette décomposition met également en évidence les relations existant entre eux au sein même du SI.

Ces méthodes sont utilisées intensivement dans le monde entier, même si l'on a déjà souvent montré qu'elles étaient surtout adaptées à la description d'un système existant ou déjà conçu. Elles sont donc essentiellement des instruments d'analyse plutôt que de conception.

1.1.2. Les méthodes systémiques

Les méthodes de conception de la deuxième génération se sont centrées sur la modélisation des données. Le SI est un modèle de la réalité organisationnelle qui apporte aux acteurs et décideurs la connaissance dont ils ont besoin pour agir et décider.

Le processus de conception est alors semblable à un processus de modélisation centré sur les données. L'information est l'incrément de connaissance que l'on peut inférer d'une donnée, tandis que le modèle de données est un outil qui permet d'interpréter les données et leurs relations. La définition d'un modèle de données facilitant l'interprétation de la sémantique et permettant la spécification du résultat de la modélisation a été le principal axe de recherche de ces quinze dernières années.

La méthode IDA proposée par François BODART est un très bon exemple de méthode systémique [BOD89].

Les modèles binaires sont également des méthodes systémiques, mais ils restreignent les relations entre les entités à de simples associations binaires. La méthode NIAM est l'exemple type.

De même, les réseaux sémantiques entrent dans ces méthodes. Ils permettent de structurer les objets selon le principe de l'abstraction, qu'elle soit de classification, d'agrégation ou de généralisation.

La *classification* permet de distinguer le niveau des instances (éléments de la classe) du niveau des classes (les objets). Une classe regroupe un ensemble d'objets de même nature.

L'*agrégation* est une abstraction qui permet de visualiser une relation entre plusieurs objets comme un seul objet. L'agrégation est très utile pour montrer la structure d'un objet en montrant les relations existant entre les objets qui le composent. Lorsque l'on étudie une telle structure, l'approche ascendante permet de comprendre l'objet, tandis que l'approche descendante permet de concevoir l'objet complexe.

La *généralisation* permet l'introduction d'objets génériques composés d'éléments spécialisés. Elle permet de classer des objets distincts dans d'autres objets plus généraux.

1.1.3. L'approche orientée objets

L'intérêt des approches objets est reconnu par tous. Ces approches, initialement développées pour les langages de programmation, sont rapidement arrivées dans le monde des Bases de Données, des interfaces Homme-Machine, et donc tout logiquement dans les Systèmes d'Informations. Ceci est dû aux nouveaux types de données que l'utilisateur doit manipuler (image, son, texte, etc.), ou, pour les informaticiens, à la vue unifiée des données et des traitements.

Cette approche définit un certain nombre de concepts de base que nous allons passer en revue dans le point suivant (1.2).

1.1.4. Les outils intelligents, les CASE's

Les outils CASE's (Computer Aided Software Engineering) actuellement disponibles permettent non seulement de fournir un support aisé et graphique à l'utilisateur, mais également une certaine aide méthodologique, allant jusqu'à automatiser certaines étapes du cycle de développement. Cette nouvelle génération provient de l'apport de l'intelligence artificielle et du génie logiciel, ainsi que de la réalisation de bases de connaissances. Celles-ci proviennent, tant de l'expérience présente ou passée, que des connaissances liées directement à la méthode et aux différentes transformations nécessaires dans chaque étape de processus de développement.

1.2. Concepts de base de l'approche orientée objets

Nous allons maintenant passer en revue les différents concepts de base de cette approche. Ces concepts ont été définis dans un article de Korson et McGregor dans un numéro consacré à l'orienté objets du journal des ACM [KOR90]. Ils appartiennent tous deux au département d'informatique de l'université de Clemson (Caroline du Sud).

1.2.1. L'objet: classe ou occurrence

Un *objet* est une entité qui combine une structure de données bien définie ainsi qu'une série d'opérations (procédures) qui caractérisent son comportement. Cet objet contient donc à la fois des données et des opérations qui sont, au niveau de la conceptualisation, en étroite relation. Il existe deux catégories d'objets: les objets classes et les objets occurrences.

Une *classe* peut être définie comme la description générale d'un ensemble d'objets particuliers. Tandis que l'*occurrence* est un objet particulier dont la structure et le comportement sont déterminés par la classe à laquelle il appartient.

Dans la description d'une classe, nous trouvons la description de variables et de méthodes.

La *variable* est une structure qui permet de stocker des données, les attributs d'un objet. Les variables sont de deux types: les variables classes qui permettent de stocker de l'information partagée par toutes les occurrences de la classe considérée, et

les variables occurrences qui contiennent de l'information spécifique à une occurrence particulière d'une classe.

La *méthode* est utilisée pour modéliser les opérations effectuées sur les objets. Ce sont des morceaux de codes décrivant comment les objets réagissent aux messages reçus. Les méthodes comprennent une partie spécification et une partie implémentation qui reste cachée de l'utilisateur. De nouveau, les méthodes peuvent être de deux types: les méthodes de classe qui modélisent la manière dont la classe (donc toutes les occurrences de celle-ci) peut répondre aux messages reçus, et les méthodes d'occurrence qui modélisent la manière dont l'occurrence elle-même peut répondre aux messages reçus.

1.2.2. L'encapsulation

L'*encapsulation* permet de modéliser, au sein d'une même entité, les données et les méthodes de cette entité. De plus, l'objet a une partie interface et une partie implémentation. La partie *interface* est constituée des spécifications des opérations pouvant être exécutées sur l'objet. C'est la partie visible de l'objet. La partie *implémentation* contient elle-même une partie donnée, c'est-à-dire la mémoire de l'objet et une partie opération qui, elle, décrit l'implémentation de chaque opération dans un certain langage de programmation.

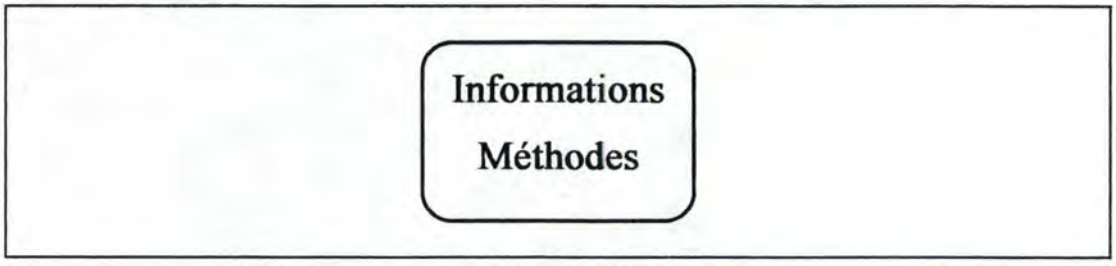


Figure 1-1: L'encapsulation.

Ce concept crée une barrière logique autour d'un ensemble d'éléments, dans notre cas les données et les opérations, et aide à la conceptualisation et à l'abstraction. Son but premier est donc de transformer un ensemble de choses disparates mais liées entre elles, en une unité manipulable. De plus, cette encapsulation permet de travailler à un niveau supérieur puisque l'on va parler d'objets et non plus de données et d'opérations.

1.2.3. Le masquage d'information

Un autre concept important au niveau des objets est le *masquage d'information* (Information Hiding). Les objets possèdent une *interface publique* et une *représentation privée*, invisible à l'utilisateur de l'objet. Ces deux facettes de l'objet sont relativement séparées. De cette manière, on n'a pas besoin de savoir comment l'objet est construit, on ne voit qu'une partie de ce qui a été encapsulé dans l'objet. De ce fait, nous nous plaçons à un degré d'abstraction plus important. Le masquage d'information permet donc de distinguer la possibilité de réaliser une opération et la manière dont celle-ci sera effectivement réalisée.

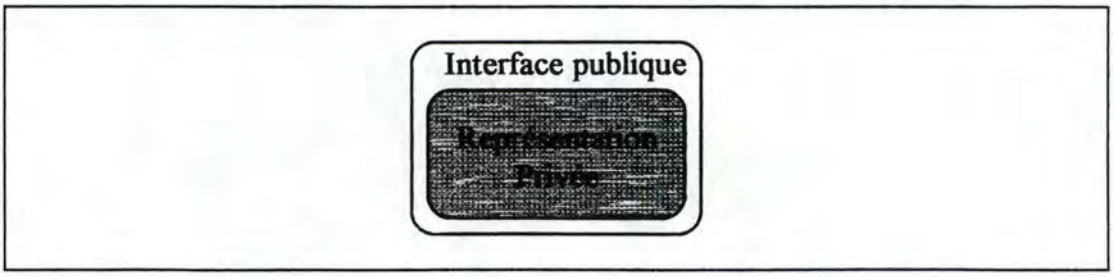


Figure 1-2: Le masquage d'information.

Les objets ne connaissent que les opérations disponibles, réalisables sur les autres objets. Dans le cycle de vie du système, nous pourrons changer l'implémentation ou la représentation interne d'un objet sans qu'il soit nécessaire de modifier le système, tout au moins si on ne modifie pas le nom des opérations disponibles.

L'encapsulation et le masquage d'information fonctionnent dans le même sens. Ils isolent une partie du système, permettent de modifier une partie du code ou de l'étendre et de corriger des erreurs sans risque d'introduire des effets "secondaires" inattendus et non nécessaires.

Les objets communiquent entre eux par le biais des messages.

1.2.4. Les messages

Un objet peut demander à tout autre d'exécuter une de ses opérations et ce, grâce à l'envoi d'un message. Un *message* doit spécifier le nom de l'opération désirée (*sélecteur*), l'objet auquel est destiné le message (*récepteur*), ainsi que les paramètres éventuels. A la réception d'un message, l'objet récepteur vérifie s'il est apte à répondre à l'opération demandée, c'est-à-dire qu'il vérifie si cette opération fait partie de son interface. Dans la négative, il transmet la demande à la classe qui va elle-même vérifier sa capacité à répondre.

On ne peut manipuler la *mémoire* d'un objet, les valeurs de ces données, que via les méthodes. De cette manière, les représentations internes sont totalement cachées.

1.2.5. L'héritage

Un premier mécanisme apporté par l'approche objets est l'*héritage*. Grâce à lui, toutes les caractéristiques d'un sur-type caractérisent également tous les sous-types. Nous avons essentiellement deux types d'héritage: l'héritage simple, où le sous-type n'est subordonné qu'à un seul sur-type, et l'héritage multiple, où le sous-type est subordonné à plusieurs sur-types immédiats.

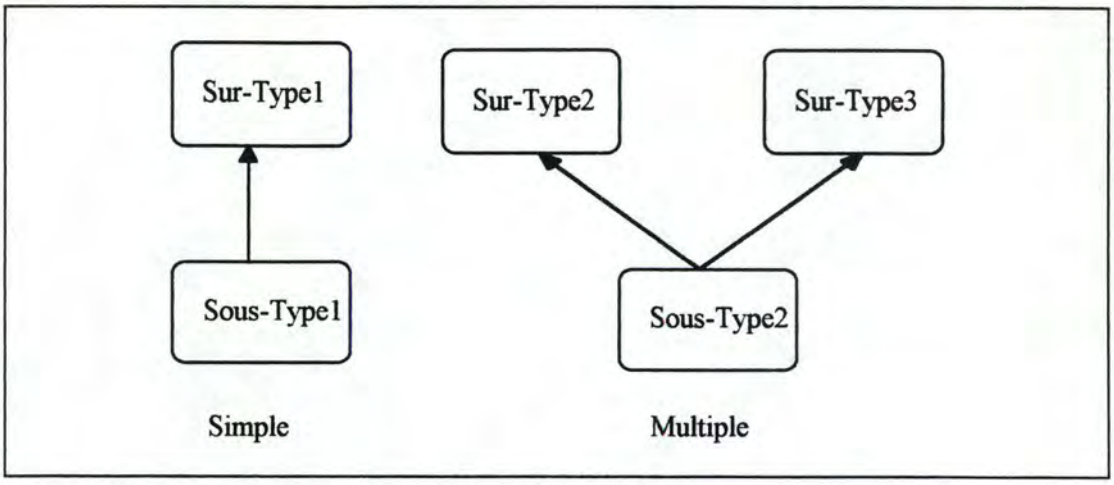


Figure 1-3: L'héritage.

1.2.6. L'overriding

Lorsqu'une opération est définie dans un type générique du système, tous les sous-types de celui-ci héritent de cette opération. Cette opération peut être redéfinie au niveau d'un objet afin de l'adapter à cet objet particulier. Le mécanisme qui permet de réaliser ces redéfinitions est l'*overriding* (le **prédominant**). L'opération exécutée sera celle héritée et redéfinie sur l'objet à qui on demande de l'exécuter. Dans le cas contraire, si elle n'est pas redéfinie, ce sera l'opération héritée qui sera exécutée.

1.2.7. La surcharge

La *surcharge* (overloading) consiste en l'utilisation d'un même nom d'opération pour des types d'objets différents. Nous pouvons ainsi faire une certaine économie d'écriture. Par exemple, l'action de création de l'objet peut avoir le même nom dans tous les objets du système.

Remarque: si l'on a de l'overriding, on a nécessairement de la surcharge, mais l'inverse n'est pas vrai.

1.2.8. Le lien dynamique

Une des conséquences de la surcharge est qu'au moment de l'exécution d'une opération, il faut sélectionner la bonne version à exécuter en fonction du type de l'objet récepteur. Il s'agit de remplacer, lors de l'exécution de l'opération, l'appel par un saut à l'adresse du début de la version correcte. C'est le mécanisme du *lien dynamique* (dynamic binding).

1.2.9. Les références polymorphiques et les liens dynamiques

Littéralement, le *polymorphisme* est l'aptitude à prendre plusieurs formes distinctes.

Dans l'approche orientée objets, on parle de *référence polymorphique*. Une telle référence est celle qui peut, à des moments différents, indiquer des instances d'objets de types différents.

Une référence polymorphique a en fait deux types: un statique et un dynamique. La référence statique est définie dans sa déclaration et peut donc être déterminée à la compilation. La référence dynamique se rapporte au type de l'objet qu'elle référence à un instant donné. Ce type ne peut donc être connu qu'a l'exécution. Le type statique d'une référence polymorphique détermine l'ensemble des types dynamiques qu'elle peut accepter au cours de l'exécution.

Chapitre 2 :

Une méthode orientée objets?

Lors de la conception d'un Système d'Information (SI), nous devons prendre en compte un grand nombre de facteurs déterminants. Dès lors, une méthode est nécessaire pour répondre aux divers points de vue que l'on doit adopter. De plus, la taille des SI devient telle que le travail en équipe, ainsi qu'un certain partage des tâches, est nécessaire. Nous avons donc besoin d'outils d'intégration des résultats provenant de ces différentes équipes. La maintenance des réalisations est souvent confiée à d'autres personnes que celles qui les ont réalisées. Il convient donc d'adopter une certaine standardisation dans la réalisation ainsi que dans la création d'une documentation la plus complète possible.

Nous allons, pour présenter le cycle de vie des systèmes orientés objets, rappeler ce qu'est une approche traditionnelle de développement et évoquer ce que l'on entend par approche de développement orientée objets. Dans un deuxième temps, nous analyserons ce que pourrait être une méthode de conception orientée objets. Nous terminerons en donnant une représentation graphique du cycle de développement orienté objets.

Ce chapitre est essentiellement basé sur un article de Brian Henderson-Sellers et Julian M. Edwards, professeur et assistant au département d'informatique de l'université de South Wales et paru dans la revue "Communications of the ACM" [HEN90].

2.1. Le cycle de développement "traditionnel"

La description traditionnelle du cycle de développement est basée sur le modèle de la "*Chute d'Eau*" représenté ci-dessous. La figure 2-1 est tirée du livre de Grady Booch [BOO91, p.198].

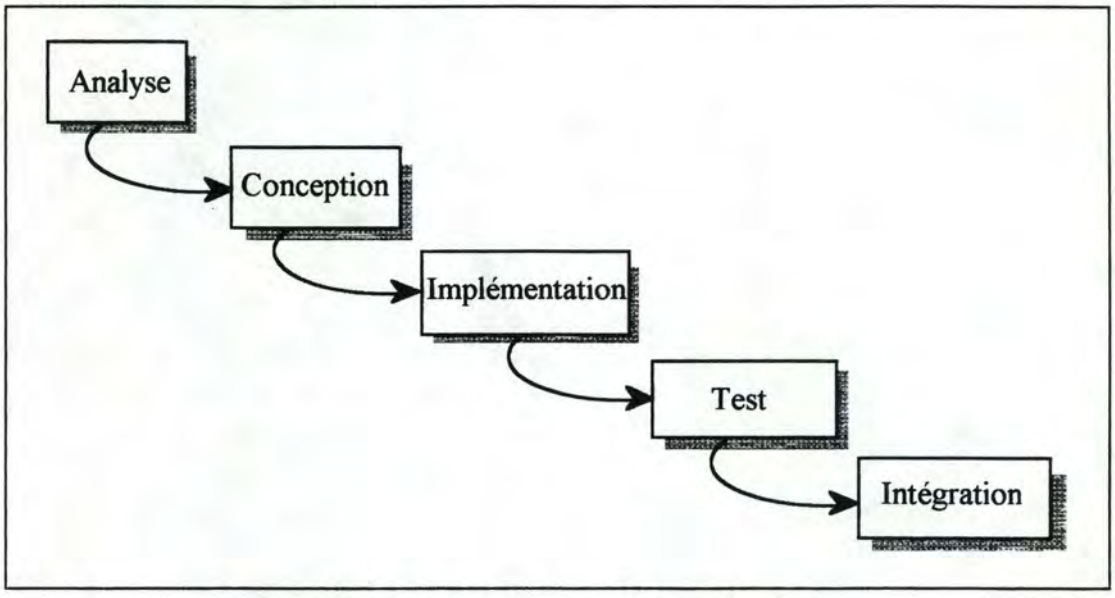


Figure 2-1: Le modèle de la Chute d'Eau.

Ce modèle sépare totalement les différentes étapes du cycle de développement: on ne passe à l'étape suivante que lorsque la précédente est terminée. De plus, il n'y a pas de retour en arrière. Une fois que l'on a quitté la phase de conception, on ne peut plus y revenir.

A un niveau plus général, ce cycle de développement peut être décomposé en trois phases: l'analyse, la conception et l'implémentation vue dans le sens le plus large, c'est-à-dire implémentation, tests et intégration.

La phase d'analyse couvre les étapes se succédant depuis la décision de développer le projet jusqu'à la spécification des attentes des utilisateurs. Cette phase doit également contenir l'analyse des exigences des utilisateurs et une étude de faisabilité. Généralement, le problème est d'abord défini et une analyse des exigences des utilisateurs est réalisée par le biais de discussions. Les spécifications sont ensuite écrites dans le langage des utilisateurs.

La phase de conception couvre les différents types de conception dont on a besoin : la conception logique et globale, puis détaillée du système et enfin, la conception physique du système. La première étape est la traduction des exigences des utilisateurs dans le langage des concepteurs et des programmeurs. Une fois cette étape réalisée, la détermination du QUOI est alors terminée. Nous pouvons passer au COMMENT. C'est à cette question que le reste de la phase de conception va répondre. Nous aurons ainsi une étape de conception logique et une étape de conception physique du système.

La phase d'implémentation couvre le reste du processus de développement: l'écriture du programme, les tests, l'intégration et la maintenance. Cette implémentation est généralement basée sur une interprétation du monde en termes de décomposition fonctionnelle. C'est pourquoi nous commencerons par demander quelles fonctions le système doit remplir.

La phase de conception fonctionnelle est donc basée sur l'interprétation de l'espace du problème et de la transition dans l'espace de la solution en tant qu'ensemble de fonctions interdépendantes. Le système final est vu comme un ensemble de procédures où les données sur lesquelles elles travaillent apparaissent comme secondaires.

La décomposition fonctionnelle est liée à l'approche Top-Down. Mais la modélisation Top-Down est beaucoup plus restrictive. Bertrand Meyer a résumé les défauts de cette méthode de conception:

- "La méthode ne tient pas compte de la nature évolutive des systèmes."
- "La notion de système caractérisé par une fonction unique est discutable."
- "L'utilisation de fonctions comme base signifie souvent que l'aspect structure de données est négligé."
- "Travailler de manière Top-Down ne facilite pas la réutilisabilité."

[MEY88, p.44]

La décomposition fonctionnelle est moins restrictive. En effet, le système est initialement vu à un niveau très élevé à propos de ce qu'il est supposé faire. Ensuite, on l'affine par différentes étapes de conception en déterminant comment il va réaliser la fonction initiale. Différents outils peuvent être utilisés pour cette décomposition. Le plus utilisé est le modèle du flux des données, qui comprend les diagrammes de flux de données (DFD), les dictionnaires de données et les schémas de structuration des traitements.

La décomposition fonctionnelle est bien adaptée aux langages de programmation traditionnels, mais ce type de décomposition conduit à un "château de cartes". Les effets secondaires sont fréquents; si on veut changer une fonction ou la modifier, un problème apparaît souvent dans une partie du programme qui ne lui semblait pas liée. De plus, ce type de système considéré comme une fonction unique ne peut pas tenir compte de nouvelles structures de données ou de nouvelles fonctions. En effet, l'ajout de certains éléments ne permet plus de garantir la correction du programme dans son ensemble.

2.2. Le cycle de développement orienté objets.

Le paradigme orienté objets utilise les mêmes éléments du système. Il n'insiste plus sur les procédures mais plutôt sur l'encapsulation des données et des procédures. Dans cette approche, le système est vu comme une collection d'objets qui sont appelés entités, objets ou classes selon les auteurs.

L'analyse et la conception de haut niveau sont réalisées non seulement en termes d'objets, mais également en termes de services, opérations, méthodes fournis à travers des relations existant entre les différents objets selon le modèle Client-Serveur.

Au sein de ces relations, les objets vont échanger des messages qui transmettront des informations, déclencheront des procédures, etc.

La conception détaillée des objets, incluant l'implémentation des procédures et des structures de données, sera reportée à plus tard dans le cycle de développement du système et sera réservée à l'objet. De cette manière, les autres objets ne sauront pas comment cet objet particulier est implémenté réellement. C'est une manière de réaliser le principe du masquage d'information.

Un système basé sur cette approche sera, de ce fait, beaucoup plus flexible. En effet, on pourra changer l'implémentation interne de l'objet sans que cela n'interfère avec les autres objets ou le bon fonctionnement du système. Nous avons donc une certaine abstraction des données, ou du moins, de leurs types.

Cette approche combine donc à la fois des aspects Top-Down et Bottom-Up. Nous entrons dans le système par le milieu, puis, d'une part, nous créons un certain nombre de classes génériques, et d'autre part, nous détaillons les classes et implémentons les éléments internes à la classe.

Nous allons faire une certaine distinction entre le haut niveau d'abstraction nécessaire pour l'analyse du système dans son ensemble et l'implémentation des classes dans un langage de programmation orienté objets. De cette manière, nous avons une analyse Top-Down et une conception des classes Bottom-Up. Ces deux éléments forment ensemble l'analyse et la conception du système d'information qui est étudié.

Un certain nombre d'étapes peuvent être identifiées lors de ces analyses et conceptions. Chacune d'elles peut être réalisée plusieurs fois, dans un processus itératif. De plus, comme les séparations entre ces étapes ne sont pas toujours bien nettes, nous nous trouvons devant un processus continu plutôt qu'un processus discret. Ces étapes sont données par les auteurs de l'article [HEN90, pp. 149-150].

- "*Entreprendre les spécifications des exigences du système.*" Cette étape est une analyse de haut niveau en termes d'objets et de leurs services. Il s'agit donc de réaliser un inventaire des objets réels.
- " *Identifier les objets.*" L'identification et la description des objets avec leurs attributs et les services qu'ils fournissent sera l'étape suivante. Un dictionnaire des objets sera créé de manière analogue au dictionnaire des données de l'approche traditionnelle. Cette identification ne doit pas porter uniquement sur les objets trouvés dans les spécifications, mais aussi sur tous les objets nécessaires et plus particulièrement les objets génériques qui seront réutilisés plus tard. L'identification des objets et des classes va également définir les opérations et les services disponibles pour ces objets. De ce fait, elle définira son interface.
- "*Etablir les interrelations entre les objets*" et ce, en termes de services requis sur les autres objets et les services qu'il rend. Pour cette étape, la création d'un "Diagramme des Flux d'Informations" (DFI) peut être très utile. Ce diagramme est la version objet du Diagramme des Flux de Données. L'information qui est manipulée par le DFI représente les messages. En effet, les données sont conte-

nues dans un objet et n'en sortent pas. Ces flux ont donc une signification différente par rapport aux flux de données. Il s'agit donc de décrire l'organisation globale des objets et de leurs comportements

- "*Création des DFI à un niveau plus bas.*" Lorsque la phase d'analyse rejoint la phase de conception, la création de ces DFI peut montrer les détails des objets. L'identification des éléments réutilisables est également importante pour leur utilisation future. Cette étape est donc réalisée sur la structure interne de l'objet, de manière locale.
- "*Les aspects Bottom-Up sont réalisés en temps qu'utilisation des DFI pour décrire la structure interne la plus détaillée.*" Les objets sont construits à partir d'objets plus primitifs selon le modèle Client-Serveur. De ce fait, une description de l'organisation globale des objets et de leur structure est obtenue.
- "*Introduire les relations hiérarchiques d'héritage.*" Les nombreux objets identifiés aux phases précédentes demandent certaines nouvelles super-classes (parents) ou sous-classes (enfants). Pour ces dernières, il faudra repartir d'une étape précédente. La création de diagrammes d'héritage peut apporter une information précieuse quant à la structure interne d'une super-classe. De plus, il permet de réutiliser une classe de ce graphe tout en connaissant exactement les super-classes et les sous-classes de l'objet réutilisé.
- "*Agrégation et/ou généralisation des classes.*" Cette étape, comme la précédente, peut demander des retours en arrière pour redéfinir les DFI qui auront été modifiés par elle. Grâce aux deux dernières étapes, un schéma objet est obtenu, schéma décrivant les objets en termes d'héritage et de généralisation - agrégation.

De ce canevas de méthode, une modélisation du système, sous forme de schémas, pouvant alors être implémentée est obtenue. Néanmoins, il faut rappeler que la conception d'un système d'information va influencer la conception des classes et leur implémentation. De même, la conception des classes et leur disponibilité va influencer l'analyse du système et la conception de sa structure.

2.3. Représentation graphique du cycle de vie orientée objets

Comme nous venons de le voir, le cycle de vie de développement n'est plus basé sur le modèle de la Chute d'Eau. Il serait plutôt basé sur le modèle de la Fontaine introduit par les auteurs de l'article.

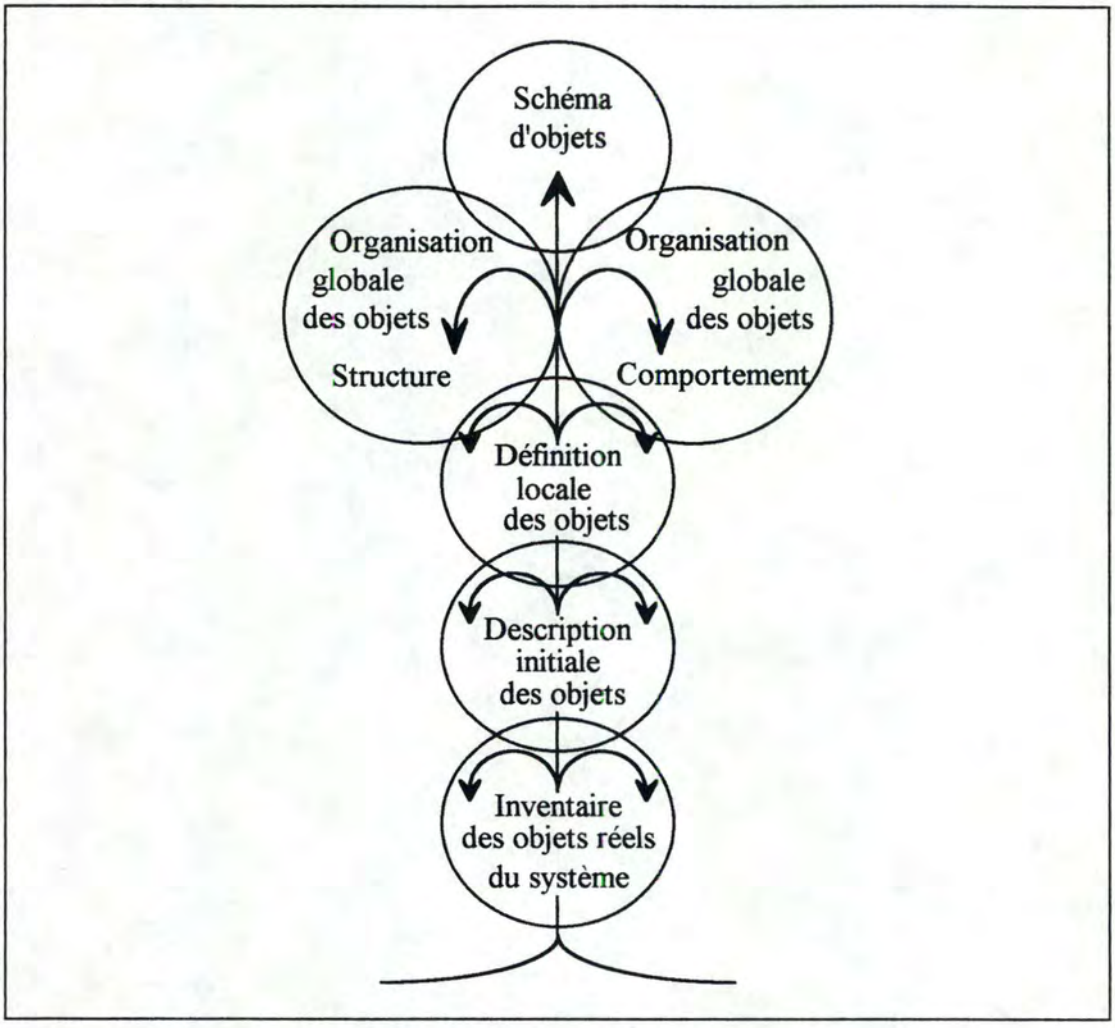


Figure 2-2: Le modèle de la Fontaine.

Ce modèle représente bien ce que la méthode orientée objets veut faire. Il montre d'abord les différentes étapes du cycle de développement et du processus d'itération et de continuité. Nous n'avons pas de coupures dans ce cycle, comme c'était le cas dans le modèle de la Chute d'Eau. De plus, l'ordre dans lequel sont réalisées les différentes étapes est bien représenté.

2.4. Conclusions

Actuellement, il n'y a pas de modèle de cycle de vie de développement accepté par tous. La tendance va cependant vers l'acceptation du modèle de la Fontaine. Néanmoins, le canevas de développement présenté semble bien adapté à celui-ci.

L'utilisation des approches Top-Down et Bottom-Up crée des interactions qui rendent la conception plus solide et en même temps, plus flexible. L'utilisation de l'approche orientée objets, depuis les spécifications jusqu'à l'implémentation, grâce aux langages de programmation orientés objets, rend le cycle de développement cohérent d'un bout à l'autre. Nous utilisons les mêmes termes, les mêmes concepts, les mêmes éléments tout au long du processus.

Les résultats devraient donc être une maintenance du système plus aisée, un système plus flexible et extensible, plus près des exigences des utilisateurs et qui requiert au total un temps de production et de maintenance plus court.

Chapitre 3 :

Présentation de méthodes actuelles

Introduction

Dans ce chapitre, nous allons présenter très brièvement différentes méthodes actuellement disponibles. Notre choix s'est porté sur les méthodes proposées par Grady BOOCH, Peter COAD et Edward YOURDON, Edward COLBERT, Sally SHLAER et Stephen MELLOR, Rebecca WIRFS-BROCK, et enfin, la méthode HOOD. Ce choix arbitraire est essentiellement dû à la documentation disponible et à la connaissance de l'existence de ces méthodes.

En effet, si la méthode de Booch et celle de Coad/Yourdon sont relativement célèbres, les autres ont nécessité une recherche plus importante. Pour la méthode de Colbert, par exemple, nous avons dû nous adresser directement à l'auteur.

Pour faciliter la lecture, nous avons joint, en Annexe 1, une synthèse de chacune de ces méthodes. Le lecteur pourra, de ce fait, consulter la ou les méthodes qui l'intéresseraient. De plus, nous avons tenté de présenter ces méthodes de manière totalement indépendantes, de façon à ce que le lecteur puisse choisir librement les méthodes qu'il voudrait consulter, sans devoir consulter les autres.

3.1. La méthode de Booch

Cette méthode, créée par Grady BOOCH, est générale et fait référence à de nombreux auteurs. Elle est présentée dans le livre "*Object Oriented Design with applications*" [BOO91].

La Conception Orientée Objets est la méthode proposée. Celle-ci définit une notation riche ainsi qu'un ensemble de modèles logiques ayant pour but de modéliser un système dont la complexité est relativement grande.

Le premier modèle est le Modèle Objet qui fournit un cadre conceptuel à la méthode. Ce modèle introduit les différents principes habituellement définis dans une approche orientée objets. Les principes d'abstraction, d'encapsulation et de modularité travaillent en synergie. L'objet propose une limite nette autour d'une abstraction simple, tandis que l'encapsulation et la modularité proposent des barrières autour de cette abstraction.

A coté de ce modèle, nous en avons d'autres : le modèle des classes, le modèle d'état-transition, le modèle des modules, le modèle des processus, et le modèle du temps qui indique les enchaînements des opérations sur les différents objets.

Dans une conception, il est important de regarder le problème sous plusieurs angles. Nous avons ainsi des structures logiques et physiques, des sémantiques statiques et dynamiques qui permettent d'observer le problème sous toutes ses faces.

Le processus de conception orienté objets n'est ni une approche Top-Down, ni une approche Bottom-Up mais plutôt une "Round-Trip Gestalt" [BOO91, p188]. Ce style de conception amplifie le développement incrémental et itératif d'un système à travers les raffinements successifs d'éléments logiques ou physiques déjà identifiés dans le système.

La conception orientée objets est généralement composée de quatre étapes :

- identification des classes et des objets à un niveau d'abstraction donné,
- identification de la sémantique de ces classes et objets,
- identification des relations existant entre ces classes et objets,
- implémentation de ces classes et objets.

Ce processus est incrémental car l'identification de nouvelles classes ou objets cause généralement une révision de la sémantique et des relations entre les classes et les objets existants.

Dans ce cycle de développement :

- *la phase d'analyse* permet aux développeurs et aux utilisateurs de déterminer un vocabulaire commun,
- *la phase de conception* affine le problème en sous-problèmes, ceux-ci étant à nouveau décomposés,
- *la phase d'évolution* combine les aspects de codage, de tests, et d'intégration,

- *les modifications* ont essentiellement lieu lors de la maintenance et sont donc des ajouts aux spécifications de départ.

Selon l'auteur, la gestion d'un projet appliquant cette méthode est généralement plus facile, le temps de réalisation est plus court et les logiciels produits sont de meilleure qualité et respectent mieux les spécifications.

3.2. La méthode de Coad/Yourdon

Cette méthode, l'Analyse Orientée Objets (AOO), est proposée par Peter COAD et Edward YOURDON.[COA92]

L'approche globale est constituée de cinq activités principales:

- l'identification des classes et des classe&objets. Les classes ne sont pas instanciables et les classe&objets sont à la fois des objets ou des classes instanciables.
- l'identification des structures. Elles représentent les différentes relations existant dans le modèle. Ces relations sont la Généralisation-Spécialisation et l'Agrégation.
- l'identification des sujets. Ils permettent de regrouper un certain nombre d'éléments afin de les cacher ou de les afficher selon le désir du lecteur ou du concepteur.
- la définition des attributs. Ils permettent de mémoriser les informations contenues dans l'objet.
- la définition des Services. Ils sont les opérations qu'un objet peut effectuer.

Ces activités ne sont pas séquentielles. En effet, l'identification des classe&objets nécessite parfois de nouvelles structures, de nouveaux attributs, ... Nous avons donc différents niveaux d'abstraction.

Le modèle lui-même est constitué de cinq couches:

- la couche sujet,
- la couche classe&objets,
- la couche structure,
- la couche attribut,
- la couche service.

Chacune de ces couches présente un certain nombre de détails, à la manière de transparents superposables. Le modèle est complet lorsque les cinq couches apparaissent simultanément.

L'Analyse Orientée Objets identifie et spécifie les classes et les classe&objets qui représentent directement le domaine du problème. Par contre, la Conception Orientée Objets (COO) identifie et spécifie les classes et classe&objets supplémentaires représentant l'implémentation des besoins. Le passage de l'une à l'autre se situe donc dans le cycle de développement.

Durant la COO, nous allons identifier quatre composantes principales:

- l'interaction homme-machine,
- le domaine du problème qui comprend les résultats de l'analyse,
- la gestion des tâches,
- la gestion des données.

Nous voyons donc clairement apparaître le continuum entre l'AOO et la COO. Il se poursuivra avec la Programmation Orientée Objets, l'ensemble étant essentiellement un processus d'enrichissement du problème. Pour plus de détails sur la COO, il est intéressant de consulter le livre des auteurs sur le sujet [COA91].

3.3. La méthode de Colbert

Edward COLBERT nous propose "The Object-Oriented Software Development Method" (OOSD) [COL89].

Dans la phase d'analyse, le modèle présenté va chercher à montrer le QUOI, c'est-à-dire les objets, classes, fonctions, comportement et attributs du problème. La phase d'analyse va être composée de quatre activités permettant de créer le modèle de l'application:

- la spécification des interactions entre les objets. Cette activité va identifier les objets, leurs interactions et relations hiérarchiques requis. Une interaction implique une opération et des flux d'informations.
- la spécification des classes d'objets. L'activité principale de cette phase consiste en l'identification des classes d'objets et des relations entre elles. Lorsque les classes d'un objet sont identifiées, le comportement et les structures de celles-ci sont également définis.
- la spécification du comportement. Le comportement dynamique du système et de chaque objet du système seront identifiés.
- la spécification des attributs. Cette activité identifie les mesures (caractéristiques) quantitatives et qualitatives et les ressources nécessaires à chaque objet du système.

Ces activités pourront être réalisées dans n'importe quel ordre dès que la première aura été effectuée. Après toutes ces étapes, nous obtenons un modèle complet du problème.

L'approche selon l'OOSD de la conception est essentiellement un raffinement du modèle produit dans la phase d'analyse vers une architecture logicielle. Elle définit une représentation spécifique au langage de cette architecture.

Pour ce faire, la première étape sera une approche de la conception. Celle-ci va créer une description de la solution logicielle du problème de manière à rester indépendante de tout langage.

Viendra ensuite une conception plus détaillée. Elle va créer une représentation tenant compte des éléments nécessaires du langage d'implémentation qui aura été choisi et les intégrant.

Dans ces deux dernières étapes, nous aurons également les mêmes activités que lors de l'analyse. Ceci permettra de déterminer tous les détails de ce système.

Selon l'auteur, si la plupart des tâches peuvent être automatisées, l'identification des objets reste entièrement de la compétence du développeur. De plus, cette méthode s'adapte bien pour tous les domaines de problèmes du monde réel.

3.4. La méthode HOOD

La méthode HOOD (Hierarchical Object Oriented Design) est une méthode développée par CISI-Ingenierie, CRI et Matra-Espace pour l'European Space Agency. Cette méthode est intéressante dans le sens où elle décompose, de manière hiérarchisée, le domaine du problème en identifiant des objets et des opérations [HOO91].

Cette méthode s'appuie donc sur deux types de hiérarchisation importants:

- *la hiérarchie de supériorité*, où les objets se trouvant en haut de la hiérarchie contrôlent et utilisent les objets se trouvant en dessous,
- *la hiérarchie parent-enfant*, permettant à un objet d'être la composition d'autres objets.

La conception d'un système peut être modélisée selon trois vues différentes:

- une *vue logique*, qui comprend un ensemble d'arbres de conception (Design Tree) structurés par des objets, et qui dépend d'une bibliothèque commune d'éléments pour les objets,
- une *vue distribution*, qui traite de l'identification des unités de distribution,
- une *vue physique*, qui permet d'attribuer les unités de distribution aux noeuds physiques.

HOOD travaille sur la vue logique et sur la vue de distribution.

Cette méthode propose un mécanisme qui permet de dépasser les limitations de la COO dans les grands systèmes complexes:

- définition d'un processus de conception par étapes (Top-Down),
- définition de procédures de vérification et de validation qui ont conduit à une formalisation du concept objet.

Cette formalisation est basée sur une notation graphique et textuelle, soutenue par le squelette de description de l'objet.

Le processus de conception de HOOD est basé sur le principe de décomposition successive de l'objet HOOD représentant le système à conceptualiser. Nous aurons une hiérarchie parent-enfant avec, comme racine, cet objet. Cette hiérarchie représente l'arbre de conception HOOD

L'utilisation d'une notation formalisée permet de supprimer les ambiguïtés, de faciliter l'échange d'informations et de faciliter l'étape finale qu'est l'implémentation.

Cette formalisation n'est pas un processus figé mais est continuellement remise à jour au cours du développement du modèle et de l'identification de nouveaux éléments, relations, etc.

Cette méthode est essentiellement liée au langage de programmation ADA. Elle intègre, dès les premières phases de conception, des aspects purement orientés vers ce langage. Elle n'est donc pas aussi générale que les autres.

3.5. La méthode de Shlaer/Mellor

Cette méthode a été développée par Sally SHLAER et Stephen J. MELLOR dans le livre "*Object-Oriented Systems Analysis, Modeling the World in Data*" [SHL88].

Un *modèle d'information* (Information Model) consiste en une organisation et une notation graphique adaptées pour décrire et définir à la fois le vocabulaire et la modélisation du domaine du problème. Le modèle identifie, classifie et permet une certaine abstraction de ce qui est dans le système étudié. Il organise l'information au sein d'une structure formalisée.

Les auteurs recommandent de commencer un projet de développement de logiciel par la construction du modèle d'information de l'application. Ensuite, la phase d'analyse se déroulera beaucoup plus facilement étant donné que le problème sera parfaitement connu et spécifié. La modélisation de l'information peut être utilisée lors de la formalisation d'une situation qui exige quelque chose de systématique. Dans le monde de la conception, ce type de modèle est évidemment à sa place.

Le processus de développement peut être décomposé en quatre étapes:

- l'analyse du problème. La première chose à faire est de construire le modèle d'information afin de déterminer sur quoi le travail va porter. Pour chaque objet, un cycle de vie sera déterminé. A chaque objet possédant un cycle de vie dans le modèle d'information, un attribut indiquant le statut, l'état de l'objet sera ajouté, son domaine sera la liste des états possibles.
- les spécifications externes.
- la conception du système. A ce point du développement, nous avons une définition claire de ce qui sera mémorisé dans le système et des opérations nécessaires pour agir sur cette mémoire. Dans cette étape, nous aurons aussi quatre tâches principales:
  - la conception de l'architecture du logiciel. Pour cela, les règles d'organisation et d'accès aux données seront définies, de même que la manière dont les opérations vont être exécutées ainsi que toutes les conventions requises par le programme lui-même, c'est-à-dire, les interfaces entre les programmes,

- la conception du contenu du système d'information, les informations qui doivent être mémorisées, disponibles et ce, d'après le diagramme des flux réalisé lors de l'analyse,
- la conception de la structure de données. Celle-ci va transformer le résultat de l'étape précédente en structures de données que l'on utilisera lors de l'implémentation. Cette structure va tenir compte des moyens disponibles et d'éventuels programmes de gestion de bases de données,
- la partition de l'application en programmes. L'application est divisée en programmes de manière rationnelle. Cette division peut conduire à du code possédant une forte réutilisabilité, du code clair et facile à lire et donc de maintenance facile.

- l'implémentation et l'intégration.

Nous détaillerons seulement les phases d'analyse et de conception. La phase de spécification externe vise essentiellement à prendre des décisions sur ce que l'application doit et ne doit pas faire. La phase d'implémentation et d'intégration transforme les résultats des phases précédentes en un programme correct et accepté par les utilisateurs comme étant ce qu'ils avaient demandé.

3.6. La méthode de Wirfs-Brock

Cette méthode a été développée par Rebecca WIRFS-BROCK, Brian WILKERSON, Laureen WIENER, dans un livre paru en 1990 : "*Designing Object-Oriented Software*" [WIR90].

Les auteurs détaillent d'abord les différents éléments de la méthode puis reprennent les principales étapes de cette conception de manière plus générale. C'est ce que nous allons exposer ici.

La conception orientée objets est un processus par lequel les spécifications du problème sont transformées en spécifications d'objets. Celles-ci comprennent une description complète des rôles et des responsabilités de chacun des objets ainsi que la manière dont ils communiquent entre eux.

Dès le départ, le processus de conception orientée objets est un processus d'exploration, de découverte. Le concepteur va rechercher les classes, essayer de les organiser selon un certain ordre et définir les différents niveaux d'abstraction du système. La conception consiste donc en différentes étapes:

- identifier les classes du système étudié,
- déterminer les opérations, les méthodes pour lesquelles chaque classe est responsable de la réalisation et les informations que chacune doit conserver,
- déterminer la manière dont les objets collaborent entre eux en vue de se décharger de la responsabilité de certaines opérations.

A partir des informations ainsi recueillies, nous pouvons commencer l'étape d'analyse du processus de conception.

Les relations d'héritage entre les classes ainsi que les différents comportements de chaque classe seront d'abord examinés. Ensuite viendront les relations entre les classes en vue de les rationaliser.

Il sera, à ce moment du processus, possible de spécifier complètement les responsabilités de chaque classe et donc de fournir un ensemble de spécifications de classes qui pourra être implémenté.

Cette méthode est donc plus ou moins générale et peut s'appliquer à tous les problèmes que l'on pourrait rencontrer.

2ÈME PARTIE:

ANALYSE ET COMPARAISON

DE CES MÉTHODES

Introduction

Suite à la présentation générale de l'approche objet et des méthodes de conception orientées objets, nous allons présenter une analyse comparative des méthodes proposées par les différents auteurs repris dans cette première partie.

Cette analyse est essentiellement basée sur la présentation d'Edward COLBERT lors du Seminar and Conference on the Object Oriented Programming Europe 91 (SCOOP91), qui s'est déroulé à Londres durant le mois de novembre 1991[COL91]. L'analyse menée par l'auteur est une extension du travail de recherche effectué par P.ARNOLD, S. BODROFF, et al. [ARN91] des laboratoires Hewlett-Packard.

L'analyse que nous allons présenter ici consiste en une comparaison des méthodes d'analyse et de conception que nous avons présentées dans le chapitre précédent. Pour chaque concept, nous donnerons une table reprenant les points de comparaison, ainsi qu'un commentaire sur ceux-ci. L'utilisation de tables peut paraître ennuyeux, mais il nous semble que celles-ci peuvent faciliter la compréhension.

Cette analyse portera sur différents critères qui peuvent être classés en trois groupes. Dans le premier, nous aborderons la manière dont les auteurs de ces méthodes utilisent les concepts de base de l'approche orientée objets. Cette première partie va donc faire correspondre les différentes terminologies utilisées par les auteurs. Nous nous efforcerons de montrer les synonymies et les homonymies. Et ce, de façon à pouvoir mettre en rapport les éléments comparables quels que soient leurs noms.

Pour ce faire, nous allons utiliser un vocabulaire commun. C'est pourquoi chaque point sera précédé d'une explication ou d'une définition des différents termes utilisés. Néanmoins, nous recommandons au lecteur de se référer à l'Annexe 1 pour les termes et la signification exacte qu'en donnent les différents auteurs.

Nous continuerons par la mise en relation des modèles compris dans ces différentes méthodes et par les différents éléments qui les composent.

Nous terminerons par la mise en oeuvre de la méthode en évoquant les différentes étapes qui conduisent à la réalisation d'une modélisation correcte du problème posé.

Chapitre 4 : Les concepts

Introduction

Dans cette première partie d'analyse, nous allons passer en revue les différents concepts de l'approche orientée objets et voir comment les auteurs les ont intégrés dans leur méthode.

Ces concepts sont :

- les classes et les objets,
- les relations,
- l'héritage,
- la visibilité des différents objets,
- le cycle de vie des objets,
- la concurrence,
- la communication.

4.1. Les classes et les objets

Les critères de comparaison sont la présence ou l'absence des concepts suivants :

- l'objet,
- le package est une routine d'un type abstrait, le type des données est considéré comme un paramètre,
- la classe,
- la méta-classe est la possibilité d'avoir des Classes de Classes, et donc d'obtenir un certain nombre de niveaux dans la modélisation.

Méthode	*Objet*	*Package*	*Classe*	*Méta-Classe*
Booch	✓	✓	✓	✓
Coad/Yourdon	✓		✓	✓
Colbert	✓	✓	✓	✓
HOOD	✓	✓		☑
Shlaer/Mellor	Instance		Objet	✓
Wirfs-Brock	✓		✓	✓

Table 4-1: Présence des concepts dans les méthodes.

Les divers éléments sont généralement différenciés par des représentations graphiques différentes. Dans la méthode de Shlaer/Mellor, la définition de l'objet correspond à ce que l'on considère généralement comme une classe et les instances comme étant les objets.

Le package, utilisé dans la méthode de Booch, Colbert et dans la méthode HOOD, est très fortement lié à ADA. Ces méthodes ont intégré les concepts des premiers langages orientés objets, dont ADA.

Dans la méthode HOOD, la notion de classe n'apparaît pas mais des objets peuvent être des objets parents, qui contiennent d'autres objets, remplissant de ce fait la fonction de la classe.

Comme HOOD ne possède pas de classes, il ne peut posséder de méta-classes mais les objets parents peuvent contenir également d'autres objets parents. Ils jouent donc le rôle de la méta-classe. Nous avons indiqué ce fait par le symbole ☑. Néanmoins, seul Colbert propose une représentation particulière de la méta-classe, Booch, se contente d'ajouter un symbole sur la notation de la classe qui joue ce rôle.

Les définitions des objets dans les méthodes ne sont pas toutes identiques quant à la structure de l'objet et aux responsabilités qu'il doit assurer.

Méthode	*Structure*		*Responsabilités*			
	Partie	*Attributs*	*Etat*	*Compor-tement*	*Opéra-tions offertes*	*Opéra-tions requises*
Booch	✓		✓	✓	✓	✓
Coad/Yourdon	✓	✓	✓	✓	✓	✓
Colbert	✓	✓	✓	✓	✓	✓
HOOD	✓		✓	✓	✓	✓
Shlaer/Mellor	✓	✓	✓	✓		
Wirfs-Brock	✓	✓	✓	✓	✓	✓

Table 4-2: Définition de l'objet dans les différentes méthodes.

Dans cette table, nous pouvons remarquer que tous les objets repris dans les différentes méthodes assurent les mêmes responsabilités. Ce sont l'état dans lequel ils se trouvent à un certain moment, le comportement géré, les opérations définies sur eux et supposées se réaliser correctement et les opérations demandées aux autres objets. Shlaer/Mellor n'introduisent les opérations que par après, dans le modèle des actions qui apparaît lorsque la modélisation des données est terminée.

En ce qui concerne la structure, elle comprend chaque fois plusieurs parties. Nous voyons apparaître une grosse différence au sujet des attributs, Booch et HOOD, ne possèdent pas d'attributs. Dans le cas de Booch, les états sont définis en fonction de propriétés qui jouent le rôle des attributs, mais de façon implicite, et ne possédant pas de représentation graphique. Dans HOOD, les constantes et les données circulant dans la partie interne de l'objet jouent ce rôle d'attribut. Ceci de nouveau, d'une manière essentiellement cachée et sans réelle représentation. Dans le cas de Colbert, les attributs n'apparaissent pas sur les représentations graphiques mais dans un document de spécification d'attributs qui en reprend les caractéristiques.

Nous pouvons observer exactement les mêmes éléments de comparaison et les mêmes commentaires pour les classes.

4.2. Les relations

Nous allons maintenant aborder les relations existant dans les différents modèles. Il en existe essentiellement de cinq types :

- la décomposition,
- l'appartenance à une collection ou à un ensemble,
- la relation de contenant et contenu,
- la communication (demande d'exécution d'une opération sur un autre objet),
- les relations définies par l'utilisateur lui-même.

Ces différentes relations peuvent exister dans un même modèle. Elles permettent de représenter un certain nombre de notions telles que : l'héritage, l'instanciation ou l'utilisation de l'objet mis en relation. Cette dernière est une relation entre deux classes ou deux méta-classes, indiquant que la classe (méta-classe) utilisatrice ne peut exister sans la définition de la classe (méta-classe) utilisée. La classe utilisatrice possède des éléments de la classe utilisée, a un accès indirect aux objets de celle-ci ou requiert des services définis par la classe utilisée.

4.2.1. Les relations entre un objet et un autre objet

Méthode	*Décompo-sition*	*Membre d'une collection*	*Contenant-Contenu*	*Communi-cation*	*Définie par l'utilisateur*
Booch	✓	✓	✓	✓	✓
Coad/Yourdon					✓
Colbert	✓	✓	✓	✓	✓
HOOD	✓	✓	✓	✓	
Shlaer/Mellor					✓
Wirfs-Brock					✓

Table 4-3: Relation entre un objet et un autre objet.

Si toutes les méthodes proposent les relations définies par les utilisateurs, c'est pour laisser le choix à ceux-ci de définir eux-mêmes les relations existant entre les objets. En ce qui concerne la méthode de Booch et celle de Colbert, les auteurs ajoutent en plus certaines relations particulières et donnent une représentation graphique spéciale pour elles. Par contre, HOOD donne des représentations différentes selon le type de relation, que ce soit par des flèches ou par le placement des objets sur les diagrammes.

4.2.2. Les relations entre un objet et une classe

Les relations existant entre les objets et les classes sont essentiellement des relations d'instanciation. Ces relations sont toujours définies dans l'ensemble des méthodes.

4.2.3. Les relations entre classes

Méthode	*Décom-position*	*Membre d'une collection*	*Conte-nant-Contenu*	*Commu-nication*	*Définie par l'utilisa-teur*	*Héritage*	*Utilisa-tion*
Booch	✓	✓	✓	✓	✓	✓	✓
Coad/Yourdon	✓	✓	✓	✓		✓	
Colbert	✓	✓	✓	✓	✓	✓	✓
HOOD	✓	✓	✓			☑	
Shlaer/Mellor				✓		✓	
Wirfs-Brock				✓		✓	

Table 4-4: Relation entre une classe et une autre classe.

Les relations indiquées ci-dessus montrent les différentes relations existant entre les classes.

Nous pouvons remarquer que HOOD n'offre pas la relation d'héritage, mais par les principes hiérarchiques et de décomposition, il offre une relation ayant le même effet. C'est pourquoi nous avons placé un symbole différent dans la table.

D'autre part, les relations chez Shlaer/Mellor et Wirfs-Brock sont réduites au maximum. En effet, nous avons uniquement des relations de communication et donc de demande d'exécution d'opérations ou d'actions et des relations d'héritage. Ce sont les relations minimales que l'on peut exiger entre les classes.

4.3. L'héritage

Nous allons maintenant nous pencher sur la relation d'héritage de manière plus précise. En effet, nous avons diverses formes d'héritage:

- l'héritage simple, entre une méta-classe et une classe,
- l'héritage multiple, entre une classe et plusieurs méta-classes,
- l'héritage mutuellement exclusif, partition des méta-classes et donc toute classe appartient à une seule méta-classe,
- l'héritage restreint, la classe a exactement le même comportement que sa méta-classe,

- l'héritage non restreint, la classe peut changer le comportement des opérations héritées de sa méta-classe,
- les classes abstraites, classes qui ne définissent que partiellement les propriétés des classes et pas celles des objets.

Nous pouvons remarquer que l'héritage mutuellement exclusif et l'héritage multiple sont incompatibles entre eux. C'est pourquoi, nous n'avons que l'un ou l'autre et jamais les deux simultanément dans une même méthode.

Méthode	*Héritage simple*	*Héritage multiple*	*Héritage mutuellement exclusif*	*Héritage restreint*	*Héritage non restreint*	*Classes abstraites*
Booch	✓	✓		✓	✓	✓
Coad/Yourdon	✓	✓				
Colbert	✓	✓		✓	✓	✓
HOOD	☑					
Shlaer/Mellor	✓		✓			
Wirfs-Brock	✓	✓		✓		✓

Table 4-5: Relation d'héritage.

Au niveau de cette table, nous pouvons remarquer que Booch et Colbert sont les méthodes qui tirent le plus d'éléments de ce concept d'héritage. Par contre, toutes les méthodes à l'exception de HOOD, ont intégré plusieurs types d'héritage, ce qui montre l'importance de ce concept. Nous tenons à rappeler la remarque faite à propos de HOOD au point 4.2.3.

4.4. La visibilité

La visibilité est la possibilité pour un objet client de voir un objet serveur afin d'utiliser ses opérations. C'est donc l'application d'un autre concept important qu'est le masquage d'informations (Information Hiding).

Nous avons différentes sortes de "visibilité":

- l'agrégation, un objet est un ensemble d'éléments, ayant le même temps de vie, les éléments étant visibles pour l'ensemble,
- la vue globale, pas de restriction de visibilité sur les objets où qu'ils soient,
- la vue restreinte, pas de restriction pour les objets faisant partie de l'agrégat, mais visibilité restreinte pour les objets n'en faisant pas partie.

Cette notion de vue restreinte va donc de pair avec la notion d'agrégat telle que définie ci-dessus. De plus, elle peut être de deux natures différentes, soit statique, fixée dès le début ou dès la compilation, soit dynamique, tenant compte des modifications des objets au fur et à mesure que l'application s'exécute.

Méthode	*Agrégation*	*Vue d'ensemble*	*Vue restreinte*	
			Statique	Dynamique
Booch	✓	✓	✓	✓
Coad/Yourdon	✓	✓		
Colbert	✓	✓	✓	✓
HOOD	✓	✓	✓	
Shlaer/Mellor		✓		
Wirfs-Brock		✓		

Table 4-6: La visibilité des objets.

Comme nous pouvons le remarquer, ce sont les méthodes de Booch et de Colbert qui offrent le plus de possibilités dans la visibilité.

La vue d'ensemble est en quelque sorte obligatoire, sans elle, nous ne pourrions pas savoir ce qu'il y a réellement dans les objets. C'est pourquoi nous pouvons dire que les méthodes de Shlaer/Mellor et Wirfs-Brock n'implémentent pas réellement le concept de masquage d'information.

La notion d'agrégation est prise en compte chez Coad/Yourdon par la notion de *sujet* qui permet de cacher une partie du modèle et de travailler seulement sur une autre partie, ou encore de cacher les couches qui ne sont pas intéressantes à ce moment là. Mais nous ne pouvons pas parler de vue restreinte, car lorsqu'un objet a besoin d'un autre, il peut voir entièrement celui-ci qu'il fasse ou non partie de ce sujet, ou agrégat.

4.5. Le cycle de vie

Le cycle de vie d'un objet peut être très variable. Nous en avons de trois types:

- statique (les objets ont le même cycle de vie que le système lui-même),
- dynamique (les objets peuvent être créés et détruits durant l'exécution du système),
- persistant (les objets peuvent vivre au-delà de la vie du système).

Les objets dont nous parlons ici, sont essentiellement des objets créés lors du démarrage du système. Ce sont donc des instances.

Voyons cela dans les différentes méthodes:

Méthode	*Cycle de vie*	*Cycle de vie Dynamique*		*Cycle de vie*
	Statique	Création	Destruction	*Persistant*
Booch	✓	✓	✓	✓
Coad/Yourdon	✓	✓	✓	
Colbert	✓	✓	✓	✓
HOOD	✓	✓	?	
Shlaer/Mellor	✓	✓	✓	
Wirfs-Brock	✓			

Table 4-7: Le cycle de vie des objets.

Généralement, les objets disparaissent avec le système, sauf dans le cas des méthodes de Booch et de Colbert. Il convient cependant de préciser que, généralement, le système sera couplé avec une base de données. Il placera les données nécessaires dans celle-ci et non les objets eux-mêmes. Cette remarque pourra éventuellement être revue lors de la mise en place de bases de données orientées objets qui ne nécessiteront plus de primitives dans les objets pour mémoriser les différentes valeurs de ses attributs dans les bases de données.

Selon Edward Colbert, l'auteur de la présentation qui sert ici de référence, la méthode HOOD, ne permet pas de détruire les objets créés durant la vie du système avant la fin de celui-ci. Cette interprétation ne nous a pas semblé évidente lors de la présentation de la méthode HOOD ci-dessus. Le manuel de référence n'en parlant pas et ne disposant pas d'informations complémentaires à ce sujet, nous avons placé un point d'interrogation dans la table ci-dessus.

4.6. La concurrence

La concurrence joue un rôle très important dans le contrôle du flux des messages et des actions dans le système. C'est ainsi que nous pouvons classer les objets selon qu'ils soient actifs, possédant leurs propres flux de contrôle et pouvant éventuellement être en train de s'exécuter lorsqu'un objet client tente de communiquer avec lui, ou qu'ils soient passifs, inactifs tant qu'une action ne lui est pas demandée par un autre objet.

Nous avons également deux types de concurrence au sein du système:

- une concurrence interne, qui est la propriété pour les objets actifs d'avoir plusieurs flux de contrôle,
- une exclusion mutuelle, qui est un mécanisme ayant pour but de garantir un état ou un objet passif partagé dans des systèmes concurrents.

Méthode	*Objets passifs*	*Objets actifs*	*Concurrence interne*	*Exclusion mutuelle*
Booch	✓	✓	✓	✓
Coad/Yourdon	✓			
Colbert	✓	✓	✓	✓
HOOD	✓	✓	✓	
Shlaer/Mellor	✓			
Wirfs-Brock	✓			

Table 4-8: La concurrence.

Comme nous pouvons le remarquer, c'est encore la méthode de Booch et celle de Colbert qui proposent le plus de mécanismes de contrôle de cette concurrence. HOOD offre une certaine concurrence interne, tandis que les autres méthodes se contentent d'objets passifs qui ne font que réagir aux sollicitations des objets clients. Pour certaines applications, cela suffit, mais pour d'autres les objets actifs, et donc une certaine concurrence interne, sont nécessaires.

4.7. La communication

La communication entre les objets est essentiellement un échange de messages. Ces messages contiennent des données ou des demandes d'exécutions.

Nous en avons de différents types:

- le mode synchrone, le client suspend son exécution jusqu'à ce que le serveur termine l'opération demandée,
- le mode asynchrone, le client envoie la requête au serveur et continue sa propre exécution,
- le mode contrarié, *Balking*, communication synchrone où le client annule l'opération si le serveur n'est pas prêt,
- le mode compte à rebours, communication synchrone où le client annule l'opération si le serveur n'est pas prêt après un certain temps donné,
- le mode événement, la communication est instantanée et atomique au sens analytique du terme,
- le mode appel de procédure, communication utilisée lors de la conception,
- le mode messagerie mutuelle, mécanismes de communication pour les appels récursifs et les rappels.

Méthode	*Synchrone*	*Asynchrone*	*Evénement*	*Appel de procédure*	*Messagerie mutuelle*
Booch	✓	✓	✓	✓	
Coad/Yourdon	✓			✓	
Colbert	✓	✓	✓	✓	
HOOD	✓	✓	✓	✓	✓
Shlaer/Mellor			✓		
Wirfs-Brock	✓				✓

Table 4-9: Mécanismes de communication.

Au niveau de la communication, Coad/Yourdon est relativement pauvre, de même que Shlaer/Mellor. Mais ce mode de communication entre les objets peut être largement suffisant pour les applications visées par ces méthodes.

Par contre, Booch, Colbert et HOOD ont pour but de couvrir un certain nombre d'applications plus spécialisées telles que les systèmes temps réels, nécessitant une plus grande richesse dans les modes de communication que les systèmes d'information plus traditionnels.

4.8. Conclusions

La couverture des concepts de l'approche orientée objets est relativement variée. Néanmoins, il faut souligner que toutes les méthodes offrent ces mécanismes, que ce soit de manière minimaliste ou au contraire avec un grand nombre de variantes. Il n'y a que HOOD, au niveau de l'héritage qui est un peu en retrait par rapport aux autres, mais il semble que cet héritage se fasse quand même de façon plus ou moins implicite.

Nous pouvons remarquer que les méthodes de Booch et Colbert sont les plus riches en ce qui concerne les concepts couverts, par contre les méthodes de Shlaer/Mellor et Wirfs-Brock sont les plus pauvres. Coad/Yourdon et HOOD se situant dans un juste milieu.

Après avoir vu les concepts, nous allons, dans le chapitre suivant, passer en revue les modèles utilisés par ces méthodes.

Chapitre 5 : Les modèles

Introduction

Les modèles présentés ici sont ceux définis dans les méthodes énoncées par les auteurs. Ces modèles sont différents les uns des autres par la manière dont ils sont créés et par les notations utilisées.

Dans un premier temps, nous allons comparer ces différents modèles selon les deux grandes divisions que l'on admet généralement : le modèle logique et le modèle physique.

Dans un deuxième point, nous comparerons les notations utilisées par les auteurs. Ces notations seront étudiées, non pas sur l'aspect graphique, aspect essentiellement esthétique, mais sur les caractéristiques elles-mêmes de la notation.

5.1. Les modèles

5.1.1. Les modèles logiques et physiques

Nous allons commencer par la comparaison des modèles logiques. Ceux-ci peuvent être situés à deux niveaux différents: le niveau du système lui-même vu dans son entier, et le niveau des composants de ce modèle. Dans chacune de ces deux subdivisions, nous pouvons faire la distinction entre l'aspect statique qui correspond à la structure du système et l'aspect dynamique qui lui, correspond au comportement temporel et fonctionnel. Nous ajouterons une colonne pour indiquer les modèles physiques utilisés par les différentes méthodes.

Méthode	*Modèle logique*				*Modèle physique*
	Niveau système		*Niveau des composants*		
	Statique	*Dynamique*	*Statique*	*Dynamique*	
Booch	Class Diag. Object Diag.	Timing Diag.	Class Diag.	State Diag.	Module Diag.
Coad/Yourdon	Class&object Diag.	Class&object specifications Services Charts	Class&object Diag.	Class&object spécifications Services Charts	Langage de programmation
Colbert	Object-Class Diag. Object-Interaction Diag.	State Diag. Information Flow	Object-Class Diag. Object-Interact. Diag. Object-hierarchy Diag.	State Diag. Information Flow	Structure Diag.
HOOD	Object Définition Skeleton	State Diag. Pseudo-Code	Object Définition Skeleton	State Diag. Pseudo-Code	ADA
Shlaer/Mellor	Information structure Diag. Communication Diag	State Diag			
Wirfs-Brock	Hierarchy Graph. Collaboration Graph.		Subsystem Card Collaboration Graph	CRC Card Contract Spécification	

Table 5-1: Comparaison des différents modèles utilisés.

Dans cette table, nous pouvons observer que chaque méthode possède un certain nombre de modèles en fonction des différents éléments qu'elle veut représenter.

Cependant, la méthode de Wirfs-Brock ne permet pas de représenter la dynamique au niveau du système. Par contre, elle le permet au niveau du composant. On peut dès lors se demander si l'ensemble de ses composants, dans leur aspect dynamique, n'est pas une représentation de la dynamique au niveau du système lui-même. Ce pourrait être le cas, mais il semble manquer un lien entre tous ses composants.

Pour la méthode de Shlaer/Mellor, c'est le niveau composant lui-même qui manque. Il n'y a donc pas de décomposition du système en ses composants. Le système est donc toujours envisagé comme un tout. Les différents composants ne peuvent pas être vus sans les éléments auxquels ils sont liés, ce qui peut être gênant lors de la modélisation de gros systèmes ou lors de la réutilisation de composants.

Pour les autres méthodes, les aspects système et les aspects composants sont repris soit dans les mêmes diagrammes, soit dans des diagrammes fort proches où la version composant est en quelque sorte une version réduite de la version système.

Le modèle physique est pris en compte par les méthodes de Booch et Colbert. Il est réduit au simple langage de programmation, chez Coad/Yourdon et HOOD, et n'est pas pris en compte du tout, chez Shlaer/Mellor et Wirfs-Brock. Dans ce dernier cas, les auteurs ont limité leur méthode à une simple analyse et n'ont pas poursuivi le cycle de développement plus loin que les aspects logiques de la conception.

5.1.2. Les représentations des objets dans ces modèles

Nous allons maintenant analyser où et comment sont représentés les objets dans ces différents modèles. Nous allons aborder les aspects de la structure de ces objets et donc des différents éléments formant l'objet, les contraintes que l'on peut poser sur ces objets et les responsabilités que les objets doivent assurer :

- leur état,
- leur comportement,
- les opérations proposées,
- les opérations requises.

Méthode	*Structure*		*Responsabilités*			
	Eléments	*Contraintes*	*Etat*	*Compor-tement*	*Opéra-tions offertes*	*Opéra-tions requises*
Booch	Object Diag. Object Template Module Template	Class Template	State Transition Diag.	Time-line Diagram		Object Diag.
Coad/Yourdon	Whole-part Structure	Class& Object Spec. Service Spec.	State Diag. in Class& Object Spec.		Service Charts Class&object Template	
Colbert	Object-Interaction Diag. Object-Hierarchy Diag. Object Description Forms	Object Description Forms	State-Transition Diag.	State-Transition Diag.	Object-Class Diag. Object Description Forms	Object-Interaction Diag. Object Description Forms
HOOD	Object Definition Skeleton HOOD Diag.	Object Definition Skeleton		Ada Based PDL	HOOD Diag. Object Definition Skeleton	ADA Based PDL
Shlaer/Mellor	Information Structure Diag.		State Diag.		Modèle des actions	
Wirfs-Brock				Collabora-tion Graph	Contract Spec.	

Table 5-2: Représentation des objets dans les différents modèles

Nous pouvons constater que, dans cette table, un nombre important de modèles ou de diagrammes interviennent. Il convient donc, lorsqu'on utilise une de ces méthodes, de ne pas chercher à comparer ses résultats avec ceux obtenus avec une

autre. En effet, les éléments recherchés ne sont pas à la même place d'une méthode à l'autre.

Nous pouvons cependant remarquer que les informations concernant les objets sont toutes reprises dans les canevas d'objets (Object Template, Object Description Form, Object Description Skeleton) fournis par chaque méthode avec, en plus dans certaines méthodes, une représentation graphique de tout ou partie de ces éléments. Il convient donc toujours de se rapporter à la définition non graphique qui semble la plus complète.

Les résultats d'une telle comparaison pour les classes sont sensiblement les mêmes. Seuls certains noms de modèles changent mais les résultats tirés des représentations des objets s'appliquent aux classes.

5.2. Les notations

Les notations utilisées se ressemblent par bien des aspects, mais sont toutes plus ou moins personnalisées dans les différentes méthodes. De ce fait, nous n'allons pas parler de la représentation graphique mais plutôt des caractéristiques sémantiques et syntaxiques apportées pas ces représentations.

Nous allons donc comparer l'expressivité de la notation qui est l'aptitude à représenter les propos. Elle doit être capable de représenter les concepts essentiels. En cas de problème de représentation, la notation nous conduit vers des descriptions plus complexes. De l'expressivité de la notation dépend en grande partie l'efficacité de l'analyse.

Nous nous intéresserons également à la syntaxe qui décrit les composants élémentaires et les combinaisons de symboles valides. Celle-ci doit être très précise et sans ambiguïté.

La sémantique est également importante. Elle donne une signification des primitives de la syntaxe et de leurs combinaisons. Le raisonnement et les possibilités de transformation de la notation sont un plus qui permet de modifier une partie de la représentation graphique obtenue.

La partition du modèle par le biais de la notation permet une certaine abstraction et un masquage des éléments qui ne sont pas pertinents pour les objets étudiés à ce moment là. De plus, elle permet de clarifier les dessins en ne montrant qu'une partie du système, ce qui les rend plus lisibles.

C'est également le but poursuivi par la possibilité offerte dans différentes méthodes de "changer d'échelle". En fait l'utilisateur peut changer la visibilité des différents éléments, c'est-à-dire en cacher un certain nombre, mais également ne faire apparaître sur la notation qu'une partie de l'information qu'elle contient.

La table ci-dessous reprend ces éléments et indique leur présence dans les différentes méthodes.

Méthode	*Expressivité*	*Syntaxe*	*Sémantique*	*Raisonnement et transformation*	*Partitionnement*	*Changement d'échelle*
Booch	✓	✓	✓		✓	✓
Coad/Yourdon	✓	✓	✓		✓	✓
Colbert	✓	✓	✓	✓	✓	✓
HOOD	✓	✓	✓	✓	✓	✓
Shlaer/Mellor	✓	✓	✓			
Wirfs-Brock	✓	✓	✓		✓	

Table 5-3: Les caractéristiques de la notation.

Nous pouvons remarquer que toutes les notations des méthodes ont une syntaxe, une expressivité et une sémantique très bien définies et non ambiguës, ce qui facilite à la fois le travail du concepteur et du lecteur des résultats de l'analyse. Tous les éléments du système sont représentés de manière univoque et précise, ce qui allège le travail de l'équipe d'implémentation.

En ce qui concerne les mécanismes de raisonnement et de transformation, la méthode de Colbert et la méthode HOOD sont les seules qui permettent ce type d'action directement sur la notation et ce, grâce à un certain nombre de règles de transformations. Dans les autres méthodes, ces modifications sont plus ou moins réalisables "à la main", mais il n'existe pas de règles de transformations quasi-automatiques.

Les différentes méthodes permettent une partition du modèle, sauf celle de Shlaer/Mellor où le modèle est constamment envisagé dans son entier. Le changement d'échelle est un autre mécanisme, il est fourni par Booch (par les différents niveaux d'abstraction utilisés lors de l'identification), Coad/Yourdon (les 5 couches), Colbert (les vues explosées des différents éléments) et HOOD (par l'utilisation des différents niveaux d'abstractions dans les graphes de hiérarchies).

Après nous être intéressé aux concepts supportés par les différentes méthodes, les modèles et les notations utilisées, nous allons nous pencher sur la mise en oeuvre des méthodes.

Chapitre 6 :

La mise en oeuvre de la méthode

Introduction

La phase suivante de notre analyse sera la mise en oeuvre de ces méthodes. Pour cela, nous nous intéresserons au contexte de développement, à la couverture du cycle de développement, et aux propriétés de cette mise en oeuvre. Nous terminerons en évoquant les divers langages et les domaines où ces méthodes s'adaptent le mieux.

Cette mise en oeuvre est très importante. En effet, une méthode peut être une des meilleures, mais si elle se révèle trop compliquée ou mal adaptée au système en cours de développement, l'analyse, et par la suite, la conception et l'implémentation conduiront à une modélisation inefficace voire erronée du système.

Dans ce cas, les développeurs choisiront une méthode peut être plus simple et moins riche, mais dont la mise en oeuvre est plus aisée, quitte à ne pas obtenir le système idéal mais une très bonne approximation. Les aspects non couverts étant simplement décrits de manière informelle sur le côté de l'analyse.

6.1. Le contexte de développement

Le contexte de développement va dépendre essentiellement de deux aspects: des données de départ et des résultats désirés. Nous allons les distinguer.

Nous aurons donc soit un développement à partir de rien (GreenField) ou du re-développement, soit un développement réutilisant des éléments d'une analyse précédente ou en vue de créer des éléments réutilisables par la suite.

Méthode	*A partir de rien*	*Re-développement*	*Avec réutilisation*	*Pour la réutilisation*
Booch	✓		✓	✓
Coad/Yourdon	✓		✓	✓
Colbert	✓	✓	✓	✓
HOOD	✓	✓	✓	✓
Shlaer/Mellor	✓		☑	✓
Wirfs-Brock	✓		☑	✓

Table 6-1: Le contexte de développement.

Nous pouvons observer que toutes les méthodes peuvent êtres utilisées pour le développement à partir de rien, mais que Colbert et HOOD fournissent, dans leurs méthodes, des éléments pour un re-développement.

Le développement pour la réutilisation est prévu par chaque méthode, mais l'utilisation elle-même de ces éléments n'est pas prise en compte au sein des méthodes de Shlaer/Mellor et Wirfs-Brock. Les autres intègrent, lors de la phase d'identification des différents éléments du modèle, la recherche d'éléments déjà développés lors d'analyses ou de conceptions précédentes. Cependant, cette recherche peut avoir lieu simultanément et de manière informelle lors de cette phase d'identification. Nous pouvons donc dire que cette réutilisation est possible dans toutes les méthodes.

6.2. La couverture du cycle de développement

Le cycle de développement est essentiellement composé de trois phases successives:

- la phase d'analyse, c'est la construction du modèle logique du système et de son environnement.
- la phase de conception, c'est la construction du modèle physique par raffinements successifs du modèle logique. Le système doit être différencié selon son environnement. Le modèle physique va dépendre du type de langage de programmation que l'on utilisera.
- la phase d'implémentation, c'est l'encodage du modèle physique dans un langage de programmation défini.

Méthode	*Analyse*	*Conception*	*Implémentation*
Booch	☑	✓	✓
Coad/Yourdon	✓	✓	
Colbert	✓	✓	✓
HOOD	☑	✓	✓
Shlaer/Mellor	✓		
Wirfs-Brock	✓	✓	

Table 6-2: La couverture du cycle de développement.

☑: Ce signe indique que si le livre de Booch et le manuel de référence de HOOD s'occupent beaucoup plus de la conception, ils donnent néanmoins une certaine méthode pour identifier les éléments du modèle. Nous ne pouvons pas leur donner la couverture de l'analyse car ils ne donnent pas de modèles logiques du système étudié.

En ce qui concerne la méthode de Coad/Yourdon, nous n'avons présenté ici que leur livre d'analyse orientée objets. Il existe un livre sur la conception crientée cbjets, mais nous ne l'avons pas présenté lors de la partie consacrée à ces deux auteurs. Nous en donnerons néanmoins les références pour les personnes désireuses de se documenter sur cette conception [COA91].

6.3. Les caractéristiques de la mise en oeuvre

Les caractéristiques possibles sont nombreuses. Nous allons nous limiter aux suivantes:

- la bonne définition du processus,
- la flexibilité de la mise en oeuvre,
- l'application d'heuristiques,
- la traçabilité, c'est à dire la possibilité de pouvoir reprendre par la suite, le cheminement suivi par les développeurs,
- les vérifications possibles au cours du processus,
- les validations des modèles réalisés.

Méthode	*Bonne définition*	*Flexibilité*	*Utilisation d'heuris-tiques*	*Traçabilité*	*Vérifica-tion*	*Validation*
Booch		✓	✓	✓		✓
Coad/Yourdon	✓	✓	✓	✓		✓
Colbert	✓	✓	✓	✓	✓	✓
HOOD	✓			✓	✓	
Shlaer/Mellor	✓	✓	✓	✓		✓
Wirfs-Brock	✓	✓	✓	✓		✓

Table 6-3: Les caractéristiques de la mise en oeuvre.

Nous pouvons remarquer que l'utilisation de la méthode de Booch n'est pas toujours facile car elle n'est pas clairement définie dans le livre. Il faut en effet réaliser soi-même la mise en oeuvre en fonction des différents éléments que l'on doit identifier. Mais celle-ci est relativement aisée dès lors que l'effort est consenti au début de l'utilisation de cette méthode.

Toutes les méthodes offrent une bonne flexibilité dans leur utilisation, si ce n'est HOOD qui semble un peu trop figée.

De même, l'utilisation de l'expérience acquise lors de développements antérieurs est facilement utilisable par toutes les méthodes, sauf pour HOOD, car la complexité des relations existant entre les différents objets et dans les graphes de hiérarchies nécessite une certaine remise en cause à chaque développement.

Généralement, toutes offrent le suivi des opérations et des choix faits lors du développement. Ces indications peuvent être très intéressantes lors de la remise en cause d'objets ou lors de la réutilisation de certaines parties du modèle.

Les possibilités de vérification sont relativement réduites. En effet, seule la méthode de Colbert et la méthode HOOD permettent une telle vérification.

Cependant, les autres méthodes ont développé un certain nombre de règles de validation qui permettent de déterminer la correction ou la validité d'un modèle ou d'un objet. Ces règles permettent de déterminer une forme plus ou moins standardisée des objets se trouvant dans le modèle. Elles permettent en plus de mettre en évidence les objets incorrects qui nécessitent une redéfinition ou une étude plus approfondie.

6.4. Les domaines d'applications et les langages utilisables

Dans ces différentes méthodes, il convient de faire un tri selon les langages vers lesquels elles sont plus ou moins orientées, ainsi que suivant les domaines pour lesquels elles sont plus appropriées.

C'est ce que nous présente la table suivante:

Méthode	*Langage*	*Domaine*
Booch	Indépendant	Tous
Coad/Yourdon	Indépendant	Systèmes d'information
Colbert	Indépendant	Tous
HOOD	ADA	Tous
Shlaer/Mellor	Indépendant	Systèmes d'information
Wirfs-Brock	Indépendant	Tous

Table 6-4: Les langages et les domaines d'applicabilité.

Comme nous pouvons le remarquer, seule HOOD est plus ou moins limitée au langage ADA. Ceci est dû essentiellement au but recherché par les auteurs de cette méthode. Les autres méthodes sont toutes indépendantes des langages. Néanmoins, il convient de rappeler que les modèles physiques créés lors de la phase de conception peuvent être dépendants du type de langage qui sera utilisé lors de l'implémentation, c'est-à-dire langage procédural, orienté objets ou autre. Ce ne sera que lors de la phase d'implémentation que le choix d'un langage précis sera effectué.

En ce qui concerne les domaines, nous pouvons remarquer que bon nombre de méthodes sont relativement polyvalentes mais que deux sont plus ou moins orientées vers les systèmes d'information, ce sont les méthodes de Coad/Yourdon et de Shlaer/Mellor.

Cette table est très importante lors du choix de la méthode qui sera employée dans une application donnée. De même, nous voudrions rappeler que le livre de Booch se termine par 5 exemples réalisés de bout en bout par la méthode, en fonction de 5 langages de programmation différents. Nous remarquerons également que l'exemple pris change en fonction du langage afin de choisir le type d'application le mieux adapté à celui-ci.

Conclusions

Les méthodes que nous venons de passer en revue nous proposent l'utilisation de tous les concepts de l'approche orientée objets.

Il est à remarquer que les méthodes provenant de la modélisation de l'information se rapprochent plus des diagrammes entité/association et diagrammes des flux. La modélisation des données est alors privilégiée. La transition vers l'une de ces méthodes sera donc plus facile si l'on vient des modèles entité/association. De plus, elles offrent une plus grande continuité pour la conception de la base de données qui sera liée à l'application en cours de développement.

Les méthodes provenant des applications temps-réel et ADA, se polarisent davantage sur les possibilités du temps-réel, sur les communications entre les objets et sur l'analyse du comportement de ceux-ci.

De ce fait, nous avons en quelque sorte deux grandes familles de méthodes. En fonction de l'application devant être développée, il reviendra au développeur de choisir une méthode dans une ou l'autre famille. Nous allons donc vers l'utilisation de boîtes à outils pour méthodes d'analyse et de conception. Cette tendance commence à apparaître et semble même être concrétisée par la volonté des grandes firmes de logiciel de mettre à la disposition des entreprises des outils respectant différentes méthodes au sein d'outils uniques. C'est le début des CASE Tools pour les méthodes de conception orientées objets.

Ce type d'outils permet également de faciliter le passage entre l'analyse et la conception, puis entre la conception et l'implémentation, et ce, grâce à une continuité de représentation et de langage de spécification.

Après avoir présenté ces différentes méthodes et après les avoir comparées entre elles, nous allons maintenant proposer une méthode qui voudrait pallier les défauts des unes et utiliser les avantages des autres. C'est le but de la troisième partie.

3ÈME PARTIE:

PROPOSITION DE MÉTHODE

Introduction

Suite à cette analyse critique des méthodes de conception orientées objets, nous allons proposer une méthode adaptée aux systèmes d'information. Le but de ces méthodes est de rester dans le monde de l'orienté objets, depuis la phase d'analyse jusqu'à la phase d'implémentation. Or, actuellement, nous n'avons pas les outils nécessaires à cette implémentation, du moins, dans la plupart des entreprises.

Nous allons donc proposer un méthode qui utilise cette nouvelle approche lors de la phase d'analyse. Durant la phase de conception, nous essayerons de garder cette approche le plus longtemps possible.

Etant donné que les langages orientés objets apparaissent de plus en plus et que nous pouvons simuler une approche orientée objets sur des langages actuellement utilisés, nous respecterons cette approche pour le placement des attributs et des opérations.

Les bases de données orientées objets n'étant pas encore opérationnelles, nous introduirons un SGBD traditionnel, ce qui nécessitera un certain nombre de modifications. Il en sera de même pour le module d'interface homme-machine.

Nous n'avons donc pas une approche purement orientée objets, mais plutôt une approche opérationnelle et donc utilisable à partir des outils actuellement disponibles.

Les méthodes proposées par les différents auteurs sont relativement théoriques et difficiles à utiliser telles quelles. De plus, les outils terminaux qu'elles sont censées utiliser ne sont pas encore disponibles. Notre méthode a donc un objectif plus opérationnel que théorique. Elle s'adresse en outre essentiellement à un domaine d'application composé des systèmes d'information. L'étude du cas qui suivra cette proposition permettra d'en montrer l'efficacité et l'opérationnalité.

Dans une première partie, nous allons détailler les éléments nécessaires durant la phase d'analyse et en donner une définition, une représentation graphique et la manière dont ceux-ci interagissent entre eux. Nous ajouterons des techniques permettant d'identifier et de valider ces différents éléments.

Nous étudierons ensuite les différentes étapes de la phase de conception, à savoir, les différentes modifications à apporter aux éléments résultant de la phase d'analyse. Nous y ajouterons les aspects d'interfaçage avec l'utilisateur et la base de données qui supportera l'application.

Chapitre 7 : La phase d'analyse

Dans cette première étape du processus de développement d'applications, nous allons devoir définir les concepts qui seront utilisés, les interactions existant entre ceux-ci et les moyens pour mener à bien cette analyse.

Nous commencerons donc par définir exactement les termes employés.

7.1. Les concepts

Nous avons essentiellement 6 concepts à passer en revue.

7.1.1. La classe et l'objet

Ces deux concepts sont les briques de base de toute méthode orientée objets. La distinction entre une classe et un objet n'est pas toujours facile à cerner. Les auteurs présentés ci-avant restent vagues à ce sujet. Voici une proposition de définition:

- *la classe* modélise un élément du système étudié. Cet élément est soit quelque chose de tangible ou de visible, soit quelque chose intellectuellement compréhensible, soit quelque chose devant mémoriser une information, soit encore ce vers quoi une action est dirigée. Une classe possède des attributs et des opérations.
- *l'objet* modélise une instance, une occurrence d'une classe.

De cette manière, nous aurons la classe Client et l'objet client_dupont.

Une classe est divisée en deux parties: une partie interface qui permet de montrer un certain nombre de choses aux autres classes et une partie implémentation qui masque la manière dont les attributs et les opérations sont réellement implémentés.

Une classe voulant utiliser une opération définie sur une autre classe obtiendra tout ce dont elle a besoin en ne regardant que l'interface de cette autre classe. Elle n'a pas besoin d'aller plus loin. Nous obtenons de ce fait un certain masquage d'information.

Ces classes seront représentées graphiquement au sein du schéma objet. Ce schéma reprendra les différentes classes identifiées dans le système.

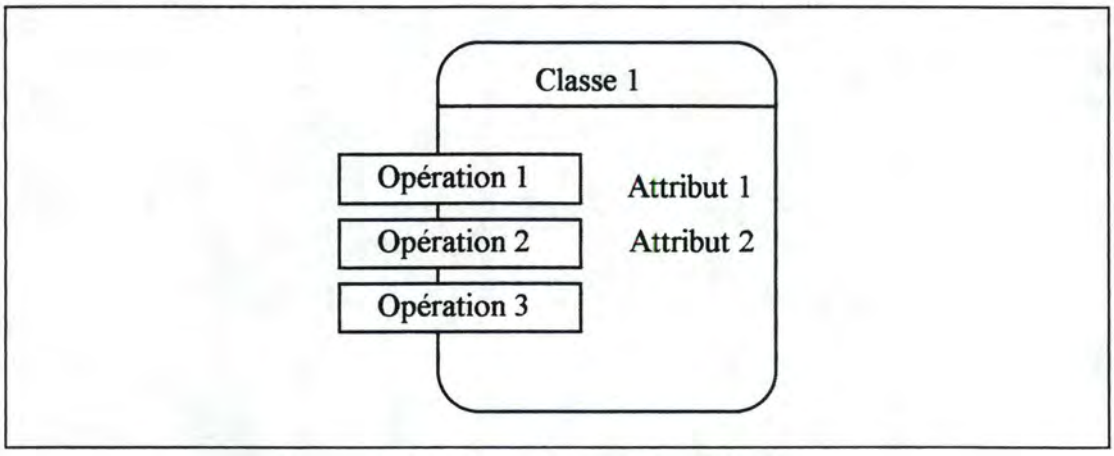


Figure 7-1: Représentation graphique de la classe.

Cette représentation ne donne pas toutes les informations nécessaires. Nous allons définir un canevas de définition de classe qui permet de décrire complètement cette dernière.

Nom : Nom_classe

Description :

Interface.

Relations :

Composée de : Nom_classe (connectivités)
Composant de: Nom_classe (connectivités)
Généralisation de: Nom_classe (connectivités)
Spécialisation de : Nom_classe (connectivités)
Utilise : Nom_classe (connectivités)

Attribut Identifiant: nom_attribut_identifiant : TYPE
Attributs: nom_attribut_1 : TYPE
nom_attribut_2 : TYPE

Opérations :

Nom_opération (paramètre : TYPE)
* Description de l'opération

Diagramme d'état : DET_nom_classe

Implémentation:

définition des opérations selon les Pré- et Post-conditions.
<..>

Figure 7-2: Canevas de définition de classe.

Nous voyons donc apparaître un certain nombre d'éléments qui sont ici définis de manière plus précise.

7.1.2. Les relations

La relation est définie par une dépendance d'une des deux classes participant à la relation vis-à-vis de l'autre.

Les relations entre les classes sont de trois types:

- la relation de spécialisation,
- la relation d'agrégation,
- la relation d'utilisation.

7.1.2.1. La relation de spécialisation

La spécialisation permet de modéliser le principe de généralisation dans lequel une classe générique est mise en relation avec les classes spécifiques qui héritent des propriétés de cette classe de plus haut niveau. Ces propriétés sont les attributs et les opérations définies sur cette super-classe.

Néanmoins, ces attributs et opérations hérités pourront être redéfinis pour les adapter plus particulièrement aux sous-classes lorsque nécessaire. Cette possibilité est l'application du principe d'overriding ou de surcharge. Lorsque nous demandons une opération héritée sur un objet appartenant à une sous-classe, cette opération sera exécutée si elle est redéfinie sur cette sous-classe, sinon, nous exécuterons l'opération définie sur la super-classe.

7.1.2.2. La relation d'agrégation

Cette relation modélise la notion de décomposition qui indique qu'une classe est composée d'un ensemble d'autres classes.

Cette relation va devoir intégrer les contraintes de connectivités présentes sur cette agrégation. Pour cela, nous placerons des cardinalités sur la représentation de cette relation ou à côté du nom de la classe dans le canevas selon les règles décrites pour les connectivités de l'association dans le modèle E/A [BOD89, p32].

7.1.2.3. La relation d'utilisation

Cette relation modélise une relation entre deux classes et indique une certaine contrainte d'intégrité. Par exemple, une classe Client sera en relation d'utilisation avec une classe Commande, modélisant la contrainte d'intégrité qui veut qu'une personne soit cliente si celle-ci a déjà passé une commande. Nous n'avons pas de relation d'agrégation, ni de spécialisation dans ce cas.

En fait, cette relation correspond à une association entre deux entités dans le schéma E/A. Nous donnerons un nom et des connectivités pour indiquer la nature exacte de cette relation.

Cette relation n'a donc rien à voir avec la simple transmission de messages lors de l'exécution d'une opération.

Voici les représentations graphiques choisies:

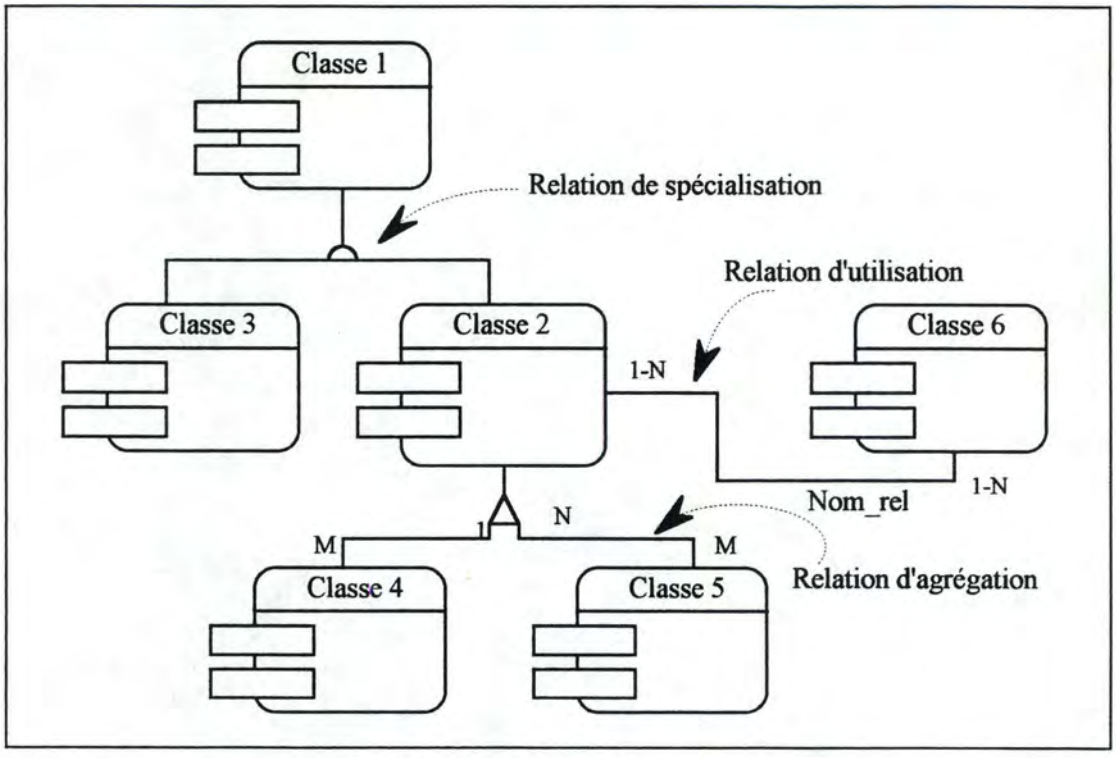


Figure 7-3: Représentation graphique des relations.

7.1.3. Les attributs

Les attributs sont les caractéristiques des classes. Leurs valeurs sont les données mémorisées par la classe.

Nous avons deux types d'attributs:

- un attribut identifiant, défini sur chaque classe, il permet d'identifier les différents objets d'une même classe,
- un ou plusieurs attributs représentant les différentes données de la classe.

Ces attributs ont un certain type et un domaine de valeur. Ils sont simples, décomposables et obligatoires selon les définitions données en [BOD89, p.23].

L'attribut identifiant est obligatoire, simple et élémentaire. Cette valeur est importante afin de pouvoir accéder à une occurrence particulière de la classe considérée.

7.1.4. Les opérations

Les opérations représentent le comportement d'une classe. Nous en avons de deux types:

- les opérations de base,
- les opérations ordinaires.

Les opérations de base permettent de créer ou de détruire un objet de la classe possédant cette opération et de consulter ou de modifier les valeurs des attributs définis sur la classe. Nous avons donc une opération de création et une opération de destruction par classe ainsi qu'une opération de consultation et une opération de modification par attribut défini sur cette classe.

Les opérations ordinaires fournissent réellement un certain comportement à la classe. Elles utilisent les opérations de base définies sur la classe, ainsi que les opérations définies sur les autres classes, par l'envoi de messages.

L'enchaînement de ces opérations par l'intermédiaire des messages au sein du système étudié permet de réaliser les fonctionnalités attendues du système étudié.

7.1.5. Les sous-systèmes

Les sous-systèmes permettent de rassembler de manière logique un certain nombre de classes reliées entre elles par des relations. Nous minimiserons les relations entre les sous-systèmes, formant ainsi une unité logique. Nous pouvons alors étudier plus facilement en détail les classes s'y trouvant.

La taille réduite de ces sous-systèmes nous permet de travailler sur des parties homogènes du système étudié, facilitant ainsi le travail.

Dans les représentations graphiques, ils sont considérés comme des sous-schémas d'objets où les extrémités extérieures des relations avec d'autres sous-systèmes sont représentées par un simple rectangle comportant le nom du sous-système et le nom de la classe vers laquelle cette relation est dirigée.

7.1.6. Les diagrammes d'état transition

Les différentes valeurs des attributs à un moment donné de la vie d'un objet permettent de déterminer l'état dans lequel celui-ci se trouve. Ces différents états peuvent être identifiés au niveau de la classe. Nous pouvons de ce fait associer un *diagramme d'état transition* à chaque classe. Chaque état y est représenté par un cercle, et chaque transition d'un état à un autre par une flèche portant le nom de l'opération responsable du changement d'état.

De cette manière, nous pouvons suivre l'évolution de la classe à travers ses opérations. Toutes les opérations ne donnent pas lieu à une transition. Nous ne retiendrons que les états intéressants pour le système.

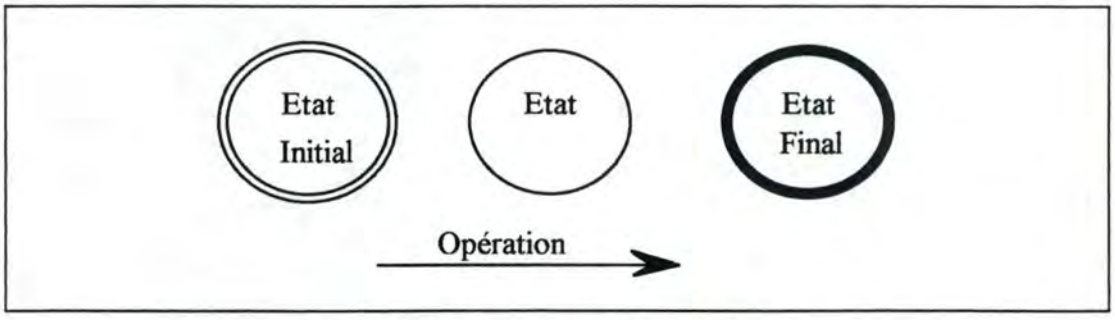


Figure 7-4: Le diagramme d'état transition.

Ces différents concepts sont regroupés au sein du modèle objet qui les met tous en relation. Nous obtenons grâce au schéma objet qui lui est associé une représentation graphique utilisable pour expliquer ce modèle aux personnes intéressées. Nous obtenons également une vision globale des structures et des relations existant entre les différentes classes. Ce schéma n'est qu'une illustration de la modélisation réalisée. Les canevas permettent de spécifier plus précisément et plus complètement les différents éléments du système modélisé.

Après avoir défini ces différents concepts, nous allons préciser une démarche permettant de mener à bien l'analyse du système.

7.2. La démarche

Cette démarche est un processus itératif et incrémental d'identification et de définition des différents éléments se trouvant dans le domaine du problème. Elle est composée d'un certain nombre d'étapes:

- l'identification des classes,
- l'identification des relations,
- l'identification des sous-systèmes,
- l'identification des attributs,
- l'identification des opérations.

Ce processus est incrémental car à chaque étape nous ajoutons un certain nombre d'éléments et itératif car nous pouvons et devons effectuer un certain nombre de retours en arrière pour définir de nouvelles classes identifiées dans les étapes suivantes.

Pour les personnes habituées au modèle E/A, nous allons donner un Mapping entre les concepts du modèle E/A et le modèle Objet:

- le type d'entité correspond à la classe,
- les attributs correspondent aux attributs,
- les associations:
    - soit aux classes si elles comportent des attributs,

- soit aux relations d'agrégation ou de spécialisation ou aux simples relations d'utilisation,
- les traitements étant un enchaînement d'opérations, ils vont être décomposés en opérations.

Nous reviendrons sur ces correspondances dans les différentes étapes. Il nous semble en effet important de donner celles-ci afin de faciliter le passage d'une méthode utilisée actuellement à une nouvelle méthode orientée objets.

Voyons ces différentes étapes.

7.2.1. L'identification des classes

Pour cette identification, notre attention se portera sur le cahier des charges afin d'identifier un certain nombre d'éléments parmis lesquels:

- les objets physiques,
- les entités conceptuelles,
- les objets immatériels,
- les données à mémoriser,
- les procédures opérationnelles,

et ce, grâce à une certaine pratique ou à l'utilisation d'indices grammaticaux tels qu'énoncés dans [BOD89, p.265] à propos du modèle E/A. Nous avons, en effet, une approche fort semblable entre ce modèle et le modèle Objet présenté ci-dessus.

Il est parfois utile d'identifier un certain nombre de classes, par exemple, toutes celles qui apparaissent lors d'une première lecture du cahier des charges. Dans les étapes suivantes, de nouvelles classes seront identifiées et un retour à cette étape effectué. De cette manière, nous construisons progressivement le modèle Objet sans trop nous attarder sur la première étape qui semble parfois difficile.

Après cette identification, étudions les relations existant entre les classes.

7.2.2. L'identification des relations

Pour chacune des classes en présence, nous déterminerons les relations qu'elle peut avoir avec les autres classes identifiées. Nous étudierons la sémantique de chaque classe vis-à-vis de celle des autres. De ce fait, nous mettrons en évidence des relations d'agrégation, relations "est composé de", et de spécialisation, relations "est la spécialisation de".

Lorsqu'une relation qui n'entre pas dans ces deux types apparaît, une relation d'utilisation est alors définie et nommée de manière significative. Ses connectivités sont données.

Ces relations peuvent également être identifiées selon un certain nombre de règles grammaticales citées ci-dessus.

La correspondance avec le modèle E/A est plus complexe et moins directe que pour les classes. Toutes les relations représentent une association, mais les associations peuvent être mises en correspondance avec une relation d'agrégation, une relation de spécialisation, une relation d'utilisation ou une classe.

Nous transformerons une association possédant des attributs en classe si les connectivités avec les entités en relation sont autres que 1-1. Dans le cas contraire, nous ne créerons pas de classe, mais nous placerons les attributs dans la classe se trouvant à l'extrémité possédant cette connectivité.

En effet, dans le cas où les connectivités sont autres que 1-1 pour les deux branches de l'association, nous n'avons pas d'autre choix que de créer une nouvelle classe car aucune des deux entités ne possède toujours de relation avec l'autre. Nous créerons également deux relations d'utilisation entre cette nouvelle classe et les classes se trouvant en relation et nous placerons sur ces relations les connectivités adéquates et respectant les spécifications.

De plus, comme cette association n'a pas d'identifiants autres que les rôles joués par les deux entités, ce qui est généralement le cas, nous allons devoir placer, comme attribut identifiant dans cette nouvelle classe, les deux attributs identifiants des classes en relation. C'est le couple de ces deux attributs qui forme l'identifiant de la classe nouvellement créée. Nous ajouterons de nouvelles contraintes d'intégrité pour vérifier cette redondance nécessaire.

Dans le cas où les connectivités d'une des deux branches sont 1-1, la création de classe n'est pas nécessaire. Il suffit de placer les attributs de cette association dans la classe participant à la relation et se trouvant sur la branche possédant cette connectivité. Nous remplaçons l'association par la relation adéquate: soit une relation d'agrégation, soit une relation d'utilisation.

Si une association ne possède pas d'attribut, nous la transformerons en relation d'agrégation, en relation d'utilisation ou en relation de spécialisation. Pour faire ce choix, il convient de s'interroger sur la signification de la relation, sur les données qu'elle représente et sur la possible composition des classes y participant. En ce qui concerne la relation de spécialisation, elle correspond à l'association de sous-typage.

Ces transformations permettent de limiter le nombre de classes. Plus le nombre de classes est important, plus il sera malaisé d'appréhender le modèle dans son entier. Nous aurons également un nombre plus important d'attributs et d'opérations à modéliser et à décrire. La notion de sous-système que nous introduirons plus loin permet seulement la réduction du nombre de classes et de relations visualisées simultanément.

La relation d'agrégation est souvent confondue avec une relation d'utilisation dont la sémantique se rapproche de celle-ci. Il convient de faire attention. Dans l'exemple utilisé ci-dessus, la classe Commande est composée d'une classe Ligne_commande, et la classe Produit fait partie de la ligne de commande.

La sémantique entre ces deux relations est relativement proche, mais lorsque l'on peut réellement utiliser l'expression "est composé de", nous avons une relation d'agrégation. C'est le cas pour la relation entre la classe Commande et la classe

Ligne_commande. Par contre, ce n'est pas possible pour la relation entre la classe Produit et la classe Ligne_commande où nous avons une relation d'utilisation dont le nom sera "Fait partie".

Les connectivités peuvent également aider à cette identification. Une relation d'agrégation a toujours une cardinalité au moins égale à un, tandis que la relation d'utilisation peut avoir n'importe quelle cardinalité.

Une fois ces relations établies et représentées dans le schéma d'objet, nous pouvons identifier un certain nombre de sous-ensembles logiques de classes qui formeront les sous-systèmes.

7.2.3. L'identification des sous-systèmes

Nous utilisons un certain nombre de règles pour identifier ces sous-systèmes. Nous pouvons regrouper ensemble les classes se trouvant en relation d'agrégation et en relation de spécialisation. En effet, elles forment une certaine unité logique. Nous essayons de minimiser les relations entre les différents sous-systèmes et, si cela n'est pas possible, d'avoir uniquement des relations d'utilisation entre ceux-ci.

Chacun de ces sous-systèmes sera nommé selon les classes s'y trouvant ou selon l'unité qui y est représentée.

Suite à cette identification, nous détaillerons les classes pour chacun de ces sous-systèmes en identifiant les attributs et les opérations. Ces deux étapes peuvent être menées dans chaque sous-système de manière plus ou moins indépendante.

7.2.4. L'identification des attributs

L'identification des attributs va repartir du cahier des charges et des classes identifiées jusqu'ici.

En premier lieu, nous allons relever un attribut identifiant sur chaque classe se trouvant dans le sous-système. Il sera défini en donnant son type et son domaine de valeur. Il permettra de distinguer les différents objets issus d'une même classe.

Ensuite, nous identifierons les données caractérisant la classe et devant être mémorisées.

Nous pourrons ainsi trouver par exemple, pour une classe Client, un identifiant: num_cli, et comme attributs: nom_cli, prénom_cli, etc.

Lors de la modélisation des relations, nous avons introduit des nouvelles classes. Nous devons les revoir et étudier les différents attributs et identifiants devant y être placés. Ces nouvelles classes n'ont pas d'attribut identifiant. Nous ajoutons les identifiants des deux classes en relation à cette nouvelle classe afin de constituer un couple identifiant pour celle-ci. Nous attirons l'attention du lecteur sur le fait que c'est le couple qui est identifiant et non pas chacun des deux attributs.

De cette manière, nous avons trouvé toutes les données servant à l'identification des objets et des données devant être mémorisées. L'étape suivante est l'identification et la définition des opérations.

7.2.5. L'identification des opérations

Reprenons chaque classe pour y définir ce que nous avons appelé les opérations de base, c'est-à-dire les opérations de création, de destruction, d'affichage des identifiants des objets existants et pour chaque attribut, les opérations de consultation et de modification de ceux-ci.

Nous obtenons ainsi un ensemble d'opérations propres à chaque classe et permettant d'en réaliser les fonctions minimales.

Ces opérations seront définies en donnant la syntaxe de leur appel, une description de ce qu'elles sont censées faire et les pré- et post-conditions d'exécution. Ces conditions seront décrites selon la logique des prédicats du premier ordre et selon le formalisme décrit dans [DUB90].

Les contraintes d'intégrité existant sur la création de certains objets peuvent être exprimées dans les pré- et post-conditions par des conditions d'existence dans les objets des classes concernées.

Dans la phase d'analyse, nous avons essentiellement une identification des relations, des attributs et des opérations. Une fois ces sous-systèmes analysés dans le détail, nous étudions les différentes fonctionnalités attendues du système.

7.2.6. Les fonctionnalités attendues

Le cahier des charges indique généralement les différentes fonctionnalités attendues du système à développer. Pour chacune d'elles nous déterminons un objet de départ, celui qui va demander l'exécution d'une opération ayant pour but de déclencher l'exécution d'un ensemble d'opérations sur cet objet ou sur d'autres objets se trouvant dans le système. Cet ensemble d'opérations va constituer et réaliser une fonctionnalité attendue du système.

Cette opération initiale sera également définie par des pré- et post-conditions qui indiqueront les états dans lesquels les objets se trouveront avant et après son exécution.

Suite à cette définition d'opération initiale, nous apparaîtront généralement de nouvelles opérations à définir sur cette classe ou sur d'autres classes concernées. Nous ajouterons ces définitions aux endroits adéquats où ces opérations seront effectuées.

Dans ces définitions, nous utiliserons les noms des relations d'utilisation pour suivre ces relations. Il est souvent nécessaire de parcourir le schéma afin de connaître les objets en relation avec la classe sur laquelle l'opération s'exécute. Pour reconnaître l'existence de cette relation, nous donnerons le nom de celle-ci ainsi que les identifiants des classes en relation.

A cette étape, nous voyons généralement apparaître un certain nombre d'opérations, de classes et éventuellement des attributs manquants.

De plus ces opérations, autres que celles de base, vont permettre d'identifier un certain nombre d'états par lesquels la classe passe tout au long de l'exécution des

différentes opérations. Nous pouvons construire le diagramme d'état transition au fur et à mesure que les opérations et les états sont identifiés.

Nous retrouvons donc le processus itératif et incrémental utilisé dans cette phase d'analyse.

7.3. La validation du modèle obtenu

Nous pouvons définir un certain nombre de règles de validation sur le schéma objet. Nous avons en effet, comme pour le schéma E/A, un ensemble de règles de complétude et un ensemble de règles de cohérence.

7.3.1. Règles de complétude

7.3.1.1. Pour les classes

Chaque classe doit posséder:

- un nom,
- un attribut identifiant,
- au moins un attribut non identifiant,
- des opérations de base, au moins création, destruction et consultation des identifiants,
- une relation avec au moins une autre classe,
- un DET correspondant à la classe.

7.3.1.2. Pour les relations

Chaque relation de spécialisation et relation d'agrégation doit posséder:

- au moins deux classes distinctes sur lesquelles la relation est définie,
- la valeur de la connectivité à chaque extrémité de la relation d'agrégation.

Chaque relation d'utilisation doit posséder:

- un nom,
- au moins deux classes distinctes sur lesquelles la relation est définie,
- la valeur de la connectivité à chaque extrémité de la relation d'agrégation.

7.3.1.3. Pour les attributs

Chaque attribut doit posséder:

- un nom,
- un type,
- une structure (simple ou répétitive, élémentaire ou décomposable),
- un domaine de valeur.

7.3.1.4. Pour les opérations

Chaque opération doit posséder:

- un nom,
- une syntaxe,
- une description,
- une pré-condition, qui peut être vide,
- une post-condition, qui contient les conditions de sortie de l'opération.

7.3.1.5. Pour les DET

Chaque DET doit posséder:

- un état de départ,
- un état de fin,
- chaque état est en relation avec au moins un autre état par une transition,
- chaque transition correspond à une opération.

Nous avons ensuite un certain nombre de règles de cohérence.

7.3.2. Règles de cohérence

Il faut:

- l'unicité des noms des classes au sein d'un schéma objet,
- l'unicité des noms des attributs au sein d'une même classe, ou d'une structure d'héritage,
- l'unicité des noms des opérations au sein d'une même classe, ou d'une structure d'héritage,
- l'unicité des noms des relations d'utilisation au sein du schéma objet,
- l'unicité des noms des DET au sein du schéma objet,
- que tout attribut identifiant soit minimal (réduit à un seul attribut simple),

- l'élimination de la redondance (par la suppression de certains attributs ou de relations d'utilisation, ou par la création de relations de spécialisation),
- l'élimination des attributs répétitifs (par la création de nouvelles classes ou de relations),
- un seul état initial et un seul état final dans un DET.

Ces quelques règles peuvent aider à la validation d'un schéma objet. Elles vérifient un certain nombre d'éléments et permettent parfois de modifier une structure existante en identifiant de nouvelles classes ou relations.

7.4. Conclusions

Cette phase d'analyse nous a permis d'identifier un certain nombre de classes en relation entre elles, de même qu'un certain nombre d'attributs et d'opérations sur celles-ci.

Nous avons pu constater le cheminement à la fois incrémental et itératif, aussi bien pour l'identification des classes et des relations que pour les attributs et les opérations.

Rappellons que ces opérations sont de deux types:

- les opérations de base : la création et destruction de l'objet et l'accès aux attributs,
- les opérations chargées de remplir un certain nombre de fonctions du système modélisé. Les enchaînements de celles-ci permettent de reconstituer les fonctionnalités que l'on aurait identifiées par une analyse fonctionnelle.

De plus, ce ne sont pas les opérations de base qui déterminent les différents états dans lesquels les objets vont se trouver, mais plutôt les opérations agissant au sein d'un enchaînement. Celles-ci vont permettre de définir un certain nombre de situations plus ou moins particulières qui vont nous indiquer les différents états dans lesquels les objets évoluent.

La phase suivante est celle de la conception.

Chapitre 8 : La phase de conception

Introduction

La première phase terminée, nous abordons maintenant la phase de conception qui va nous permettre d'obtenir une solution implémentable.

Pour cela, nous allons approfondir la définition des classes identifiées lors de la première phase. Chaque opération sera exprimée sous forme d'algorithme abstrait lié au type de langage de programmation choisi. Ici, nous considèrerons que nous utilisons un langage de programmation orienté objets.

Nous introduirons l'interface avec l'utilisateur et verrons comment celle-ci joue un rôle important dans la réalisation des fonctionnalités attendues du système.

Nous introduirons également la structure de la base de données. Cette base de données va mémoriser de manière permanente les différentes informations nécessaires à l'application. Cette étape sera très importante car elle va nécessiter une adaptation de notre modèle objet pour le rendre compatible avec le type de BD choisi. Etant donné que le modèle relationnel est relativement aisé à mettre en oeuvre, nous le choisirons.

Ces différents ajouts se réaliseront progressivement. Ces trois premières étapes peuvent se dérouler pour chaque sous-système de manière plus ou moins indépendante. Cependant, nous devons faire attention aux interactions existant entre ceux-ci. Nous allons donc concevoir indépendamment les classes de chaque sous-système, puis nous introduirons l'interface avec l'utilisateur et enfin la base de données.

Une fois ces sous-systèmes conçus, nous établirons les différentes interactions entre ceux-ci. Nous terminerons par la conception des différentes opérations comprises dans les fonctionnalités et agissant sur l'interface avec l'utilisateur et avec la base de données. Nous ne pouvons pas les introduire plus rapidement car nous avons besoin de la structure de la BD.

8.1. Conception des classes

Dans cette étape, nous allons affiner les classes c'est-à-dire essentiellement les opérations.

Nous reprendrons chaque opération et nous en donnerons un algorithme. Pour cela, nous utilisons un pseudo-langage essentiellement basé sur des opérateurs simples tels que la condition, la boucle, les additions, etc. Ces opérations vont donc être transformées en algorithme utilisant des instructions simples permettant de préciser exactement ce que l'on veut et ce, sans devoir aller jusqu'au langage d'implémentation. De cette manière, nous pourrons facilement réaliser le code lors de la phase d'implémentation.

De plus, nous décomposerons les différentes opérations de manière à n'utiliser que ces opérateurs simples. Les structures complexes dans les algorithmes seront éliminées par décomposition d'opérations en sous-opérations élémentaires. Nous voyons donc apparaître un certain nombre de nouvelles opérations. Nous les définirons sur les classes les plus adéquates, c'est-à-dire sur les classes utilisées. Cet aspect donne également un critère de décomposition, chaque opération agissant sur une classe ne doit pas agir sur une autre, mais elle peut demander l'information par un appel à une opération située sur cette autre classe.

Cependant, ces opérations de base vont utiliser les primitives propres au langage de programmation. Nous aurons donc un certain nombre d'opérations se ressemblant fortement et dont seuls les arguments seront différents. La recherche de la valeur d'un attribut ne peut pas se réaliser de manière multiple. Toutes les opérations de consultation et de modification seront dès lors semblables.

Pour représenter et préciser les appels aux opérations, nous utiliserons les conventions suivantes:

- Nom_opération (paramètre) si l'opération désirée est définie sur la classe considérée,
- Nom_classe.Nom_opération (paramètres) si l'opération est définie sur la classe Nom_classe.

Dans ces définitions, nous aurons un certain nombre d'opérations qui ne seront pas décrites de manière plus précise car elles correspondent à une primitive du langage de programmation. Nous pensons à l'opération de création d'un objet d'une certaine classe. Nous pouvons vérifier un certain nombre de choses et effectuer des opérations, mais nous ne pourrons pas dire quelle primitive va réellement créer cette instance. En effet, le langage de programmation offre généralement une primitive qui permet de le faire.

Cette étape ressemble à celle utilisée généralement par toutes les méthodes de conception de logiciels, qu'elles soient orientées objets ou non.

Nous poursuivrons en introduisant les interactions entre l'application et les utilisateurs par l'intermédiaire de l'interface homme-machine.

8.2. Conception de l'interface

Nous devons maintenant modéliser les interactions entre le système et l'utilisateur. L'interface sera réalisée dans un sous-système à part. Nous utiliserons une seule classe sur celui-ci qui nous permettra de ne pas connaître sa structure réelle. Généralement, un gestionnaire d'interface sera utilisé et cette classe réalisera la communication entre ce gestionnaire et l'application. Par exemple, lorsque nous demanderons l'affichage d'un message, le gestionnaire d'interface réalisera toutes les opérations nécessaires pour afficher ce message dans une "fenêtre" à l'écran. De même, lorsque nous aurons besoin d'une réponse, ou d'une saisie, nous utiliserons une seule opération qui donnera le libellé de la question et qui renverra le paramètre une fois que l'utilisateur aura répondu.

Ce gestionnaire sera représenté par une classe Utilisateur. Cette classe servira d'intermédiaire entre l'application et l'utilisateur, nous permettant ainsi d'introduire une certaine indépendance entre l'interface et l'application. La manière dont cette classe et les interactions avec l'utilisateur seront effectivement réalisées ne dépendra que de cette classe et sera "transparente" vis-à-vis de l'application. Nous implémentons donc le principe de modularité et de localisation des modifications. Par exemple, si nous désirons changer la langue de l'interface, ou l'interface elle-même, il nous suffira de changer de classe Utilisateur.

Les aspects esthétiques et de disposition des différentes fenêtres à l'écran seront réalisés de manière indépendante de notre conception. De cette façon, une équipe spécialisée dans le domaine de l'ergonomie sera chargée d'implémenter les différentes opérations que nous allons maintenant définir sur la classe Utilisateur.

Cette classe Utilisateur est relativement complexe. En effet, elle va contenir toutes les opérations élémentaires nécessaires aux objets de l'application.

Nous en avons essentiellement trois types :

- les opérations d'affichage d'information,
- les opérations de saisie,
- les opérations de sélection dans une liste.

Cette classe Utilisateur peut être décomposée selon les moyens d'interaction disponibles. Cette décomposition n'est utile qu'au moment de l'implémentation de cette classe. De plus, toutes les classes de l'application vont utiliser la classe Utilisateur à un moment ou à un autre, nous n'allons donc pas représenter les messages échangés. Nous considèrerons ces échanges de messages comme implicites.

De cette manière, nous pouvons créer une instance de la classe Utilisateur par utilisateur de l'application. Il est donc possible de gérer différents utilisateurs. La simultanéité des actions de ceux-ci sera gérée par des mécanismes de communication au sein de l'application toute entière. Ces aspects étant trop techniques et plus liés à l'implémentation, nous ne les aborderons pas ici.

En effet, cette concurrence provoquerait l'utilisation d'un autre langage de programmation qui serait multi-tâches et nécessiterait éventuellement certaines modifications dans les différents objets pour rendre compte des problèmes de concurrence qui pourraient se poser, de même qu'une gestion de la concurrence ferait intervenir un SGBD plus complexe .

Si le langage ne permet pas l'utilisation de types abstraits de données (ce qui est malheureusement souvent le cas) et pour permettre l'échange d'informations entre cette classe Utilisateur et les classes de l'application, nous pouvons adopter une technique qui permet de transmettre un paramètre général composé lui-même de nombreux paramètres regroupés dans un produit cartésien. Nous donnerons toujours le même paramètre en ne donnant une valeur qu'aux éléments dont nous avons besoin à ce moment-là.

Cette stratégie nécessite une multiplication des opérations de la classe Utilisateur pour permettre l'affichage ou la saisie des bons éléments du paramètre général. Cette multiplication permet de garder un contrôle plus précis de ce qui doit être fait. Nous pouvons de ce fait utiliser des algorithmes particuliers pour optimiser certaines opérations plus spécifiques, et ce, au prix d'une perte de généralité. Un compromis entre la généralité des opérations et leurs performances est donc souvent nécessaire.

Nous pouvons placer dans chaque opération définie jusqu'ici les appels réels à cette nouvelle classe.

L'ensemble des opérations décrites sur la classe Utilisateur va s'enrichir d'un certain nombre d'opérations en provenance des différents sous-systèmes. Nous essayerons, autant que possible, de définir sur la classe Utilisateur des opérations simples pouvant être réutilisées par d'autres opérations. Par exemple, nous pourrons définir sur la classe Utilisateur des opérations d'affichage et de saisie de nombres entiers ou de chaînes de caractères permettant d'obtenir des informations intermédiaires. Ces opérations génériques permettent d'éviter la création d'un grand nombre d'opérations semblables. Nous ajouterons le message ou la question à l'appel.

Une fois toutes ces opérations réalisées, nous abordons le problème de la base de données.

8.3. Conception de la base de données

Le choix de la base de données est important dès la conception. En effet, de son type va dépendre la structure des classes et des relations que l'on va y placer.

Nous allons présenter les modifications à apporter pour un SGBD de type relationnel. Nous aurons donc des transformations à réaliser, importantes sur les relations et plus simples sur les classes, pour obtenir la base de données conforme au modèle relationnel.

Les opérations d'accès à la BD vont se réaliser à partir d'une classe Base_Données. En effet, de cette manière, les objets n'interviennent pas directement sur celle-ci, ce qui permet de modulariser l'application. Chaque opération définie sur les classes peut avoir des opérations d'accès à la BD. Nous utiliserons donc les mêmes techniques que lors de l'étude de l'interface avec l'utilisateur.

En repartant du modèle objet du sous-système considéré, nous pouvons passer directement à un schéma de la BD. En effet, nous connaissons tous les objets, tous les attributs définis, ainsi que les opérations pouvant nécessiter un accès à la BD.

Nous aurons donc un Mapping entre les classes et leurs attributs d'une part et les tables avec leurs colonnes d'autre part. Les attributs identifiants sont les Keys, les identifiants secondaires sont un index secondaire posé sur la table. Chaque ligne (row) contiendra les données pour un objet particulier.

Pour cette étape de transformation du modèle objet en schéma de la BD, nous devons changer un certain nombre de structures. En effet, les relations entre classes ne sont plus permises. Nous les transformerons en suivant les règles fournies au chapitre VII de [HAI86].

Les relations à éliminer sont essentiellement de trois types:

- relation de spécialisation,
- relation d'agrégation,
- relation utilisation.

Nous allons nous y attarder quelque peu.

Les relations ne contiennent pas d'attribut et ce, par définition des relations dans le modèle objet. De ce fait, leur transformation va dépendre essentiellement de leur cardinalité.

Si la relation est de type N-N pour chacune de ces extrémités, nous créons un type d'articles représentant cette relation et dont les attributs sont les identifiants des classes participant à la relation. Nous ajoutons deux contraintes d'intégrité pour nous assurer que les valeurs des attributs de ce nouveau type d'article correspondent aux valeurs des attributs définis sur les classes participant à la relation. Nous obtenons donc un type d'article dont chaque élément est un couple.

Si la relation est de type 1-N pour au moins l'une de ces extrémités, nous éliminons cette relation en réalisant une rotation telle que définie dans [HAI86, p.60], c'est-à-dire que nous plaçons l'identifiant de la classe se trouvant à l'extrémité de connectivité 1-N dans la classe se trouvant à l'autre extrémité. De plus, nous ajoutons une contrainte d'intégrité sur ce nouvel attribut. En effet, il doit se retrouver dans la classe d'origine où il est identifiant.

8.3.1. Relation de spécialisation

La relation de spécialisation est composée d'une relation de type 1-1 aux deux extrémités.

Nous pouvons lui appliquer la deuxième transformation: échanger les attributs identifiants des deux classes participantes et créer deux contraintes d'intégrité, ce qui n'est pas toujours facile à gérer.

Une autre manière est de supprimer cette relation de spécialisation en plaçant les attributs de la classe générique dans les classes spécialisées. De cette façon, nous

diminuons la hiérarchie dans le schéma. Toutes les classes sont au même niveau, nous n'avons ni super-classes, ni sous-classes.

Généralement, c'est la seconde solution qui est choisie car nous réduisons le nombre de classes dans le schéma et n'introduisons pas de nouvelles contraintes d'intégrité. Les classes issues de la transformation sont plus complexes, en ce sens qu'elles intègrent à présent les attributs et les opérations définis avant sur la super-classe. Nous avons une duplication d'attributs et d'opérations, mais pas de redondance d'information. De plus, un schéma de ce type est plus lisible car plus simple.

8.3.2. Relation d'agrégation

Cette relation est transformée par l'application des techniques décrites ci-dessus en fonction des connectivités existantes. Cette transformation peut donc se réaliser très rapidement.

8.3.3. Relation d'utilisation

La relation d'utilisation possède des connectivités relativement diverses. Nous allons donc chaque fois l'étudier afin de déterminer avec précision quelle transformation choisir.

Ces relations forment un passage d'un sous-système à un autre. Elles doivent être construites et modélisées dans la base de données en tenant compte à la fois des classes participant à cette relation et des connectivités se trouvant sur celle-ci. Ces connectivités nous indiquent quel identifiant doit être dupliqué dans l'autre classe. Pour cela, nous reprenons les classes se trouvant dans les deux sous-systèmes. Nous avons donc une étape de transformation des relations existant entre ces sous-systèmes.

De plus, si nous trouvons plusieurs relations d'utilisation entre deux mêmes classes, nous les transformons l'une après l'autre en veillant à ne pas introduire deux fois le même attribut dans une des deux classes, ou à ne pas créer deux fois la même contrainte d'intégrité.

Une fois ces transformations effectuées, nous reprenons les différentes opérations liées aux fonctionnalités attendues de l'application.

8.4. Conception des fonctionnalités attendues

Les différentes opérations, ou plutôt les différents enchaînements d'opérations, vont être redéfinies en fonction des algorithmes introduits, de l'interface utilisateur et de la BD.

Nous commençons ce travail à partir des opérations déclenchant ces enchaînements. Nous les plaçons sur les classes qui les provoquent réellement et qui coordonnent en quelque sorte ceux-ci. Nous pensons tout naturellement à la classe Utilisateur.

Généralement, ces opérations initiales, comme on pourrait les nommer, sont exécutées suite à une sollicitation de l'utilisateur: sélection dans un menu, introduction d'une ligne de commande, etc.

Nous plaçons une opération dans la classe Utilisateur. Elle va d'une part vérifier les données saisies, et d'autre part exécuter la ou les opérations nécessaires pour mener à bien cette fonctionnalité. Cette opération initiale n'a pas pour rôle de déclencher toutes les opérations nécessaires, mais plutôt d'en déclencher quelques unes qui, à leur tour et en fonction des décompositions réalisées, vont demander l'exécution d'autres. Un paramètre de contrôle devra suivre ces opérations pour pouvoir rendre compte des différents problèmes rencontrés en cours d'exécution.

Ces décompositions, rappelons-le, sont liées à la structure de l'algorithme initial et aux accès aux diverses classes de l'application.

Cette étape va utiliser la BD issue des transformations du modèle objet, et de l'interface utilisateur. Nous allons généralement identifier de nouvelles opérations sur différentes classes et établir de nouveaux échanges de messages. Ces identifications devront être documentées et reportées sur le modèle objet afin que celui-ci reste conforme à l'application en cours de construction.

Les décompositions des opérations selon des critères de structure d'algorithmes et de localisation des informations (sur les classes) sont réalisées de la même manière qu'une décomposition fonctionnelle. La seule différence est le second critère qui peut parfois nécessiter une découpe supplémentaire.

Une fois cette étape terminée, nous sommes devant une application définie en termes d'opérations sur des classes possédant des attributs, d'une interface utilisateur comportant les opérations nécessaires au gestionnaire d'écran et d'un schéma de base de données conforme au type de SGBD choisi. Nous pouvons donc, relativement facilement, passer à l'étape d'implémentation.

A chaque classe va correspondre une structure dans le langage de programmation choisi, langage dans lequel nous implémenterons les différentes opérations. Des critères d'efficacité et de particularité de celui-ci peuvent remettre en question l'algorithme réalisé, mais pas la décomposition des opérations. Si cela s'avère nécessaire, une documentation devra y faire référence de manière à garder une trace de tout ce qui aura été modifié lors de cette dernière phase.

Conclusions

Dans cette méthode, nous avons pu remarquer que de nombreuses étapes sont nécessaires pour arriver à une analyse et à une conception correctes, cohérentes et complètes.

Il nous paraît important de souligner plusieurs choses:

- la relative simplicité dans la mise en oeuvre. En effet, le nombre de concepts différents est relativement réduit. De plus, nous avons une certaine systématisation des différentes étapes qui laisse, malgré tout, une très grande souplesse à l'ordre dans lequel le concepteur va les réaliser.
- la continuité dans les représentations tout au long du cycle de développement qui permet à tout moment de revenir en arrière.
- l'utilisation d'une représentation commune qui favorise une meilleure communication que ce soit entre les membres de l'équipe d'analyse, entre les membres de l'équipe de conception ou entre les membres ces deux équipes.

Loin de dire que notre méthode est parfaite, il nous semble qu'elle reprend de nombreux aspects importants des différentes méthodes présentées dans les premières parties. De plus, elle nous semble opérationnelle pour les environnements actuellement disponibles.

Pour terminer, nous précisons que le but poursuivi par cette méthode est le développement d'applications dans le domaine des systèmes d'information. Dès lors, il ne convient pas de l'utiliser dans des applications temps réel, bien que certaines modifications le permettraient.

Nous allons maintenant procéder à l'application de notre méthode à un cas bien précis.

4ÈME PARTIE:

ETUDE D'UN CAS:

ANALYSE ET CONCEPTION

Introduction

Après avoir défini la méthode, il nous faut l'essayer! C'est le but de cette dernière partie.

Nous allons reprendre un cas relativement commun au sein de l'Institut d'informatique, à savoir: la gestion de l'hôpital Mortsubite. Ce cas est généralement étudié lors d'un travail pratique de seconde licence et maîtrise en informatique. Il s'agit de réaliser la gestion des patients, des prescriptions de soins et de la facturation avec remboursement par un organisme assureur, d'un hôpital de taille moyenne.

Dans un premier temps, nous allons réaliser la phase d'analyse. Pour cela, nous utiliserons l'énoncé tel qu'il est donné lors du travail pratique. Pour plus de facilité, un exemplaire de cet énoncé se trouve en Annexe 2.

Après cette première phase du cycle de développement, nous aborderons la phase de conception qui vise à adapter les résultats de l'analyse aux types d'outils que l'on utilisera lors de la phase d'implémentation.

Chapitre 9 : La phase d'analyse

Dans cette phase d'analyse, nous allons partir de l'énoncé du problème et essayer de trouver les classes d'objets existant dans le domaine de l'application.

9.1. Identification des classes

Nous trouvons rapidement un certain nombre de classes : hôpital, lits, service de soins et chambres. La suite de la lecture nous précise la configuration des chambres au sein des services et de l'hôpital. Nous pouvons identifier une certaine relation d'agrégation entre ces différentes classes.

Par exemple, la classe Hôpital est composée d'un certain nombre de services de soins, modélisés par la classe Service_soins. Nous définissons alors une relation "est composé de", partant de la classe Hôpital et aboutissant à la classe Service_soins. De plus, nous pouvons poser une contrainte de connectivité pour modéliser le fait qu'un hôpital contient un certain nombre de services de soins. Dans le problème qui nous occupe, nous avons 12 objets de la classe service-soins contenus dans l'objet Mortsubite de la classe Hôpital. Cette connectivité sera exprimée par la cardinalité de la relation.

Si le nombre de services variait, il nous faudrait revoir cette connectivité. En effet, chaque objet de la classe Hôpital peut avoir un nombre différent de services de soins. Nous considérerons que cette connectivité est 1-N objets services_soins pour 1-1 objet hôpital. Le nombre exact d'objets service_soins sera créé lors de l'initialisation de l'application, c'est-à-dire lorsqu'un objet hôpital est créé.

L'opération de création d'instances de la classe Hôpital va donc demander, dans un de ses paramètres, ou par un dialogue avec l'objet sollicitant cette création, le nombre d'instances de la classe Service_soins à créer. Elle requerra l'exécution de l'opération de création de la classe Service_soins le nombre de fois qu'il sera nécessaire. Les connectivités réelles ne seront connues que durant la vie de l'application. De même, si nous désirons ajouter un objet service_soins à un objet hôpital, il nous suffira d'exécuter l'opération de création définie sur la classe Service_soins.

Dans notre exemple, le nombre de services de soins est de 12, l'opération Hôpital.Création demandera 12 fois l'exécution de l'opération Service_soins.Création.

9.2. Identification des relations

Un raisonnement semblable sera réalisé pour les différentes relations "est composé de". Dans notre cas, nous aurons la classe Service_soins en relation avec la classe Chambre et la classe Chambre avec la classe Lit. Les cardinalités de ces relations seront 1-M objets chambre pour un objet service_soin et 1-M objets lit pour un objet chambre. Les opérations de création d'objets agrégés devront chaque fois demander le nombre exact d'objets le composant.

De même dans les relations inverses, nous aurons des contraintes d'intégrité: chaque objet composé devra posséder une relation d'appartenance avec l'objet composant. C'est-à-dire que chaque objet lit doit appartenir à une chambre, que chaque objet chambre doit appartenir à un objet service_soins. Ces contraintes d'intégrités seront exprimées par des cardinalités de 1-1 entre les différentes classes.

Dans notre exemple, nous avons donc une relation "est composé de" entre la classe Service_soins et la classe Hôpital. Les cardinalités seront 1-M objets service_soins pour 1-1 objet hôpital. Nous aurons les mêmes cardinalités pour les autres relations, c'est-à-dire entre la classe Chambre(1-M) et la classe Service_soins(1-1) et entre la classe Lit (1-M) et la classe Chambre(1-1).

Il nous reste à déterminer comment nous allons modéliser les différentes catégories de chambres. Chaque objet chambre appartient à l'une des trois catégories définies dans Hôpital. Chaque catégorie va donc contenir un certain nombre de chambres et chaque chambre va appartenir à une seule catégorie. Il est donc logique de considérer ici une relation d'agrégation ayant pour classe agrégée Catégorie_chambre et pour classe composante Chambre. Nous placerons les attributs *cat* et *prix_plein_lit_j* dans la classe Catégorie_chambre.

Une des utilisations de cette classe est la mémorisation du prix du séjour dans celle-ci. Nous avons une relation d'agrégation plutôt qu'une relation de spécialisation car la répartition explicite des chambres en trois classes spécialisées n'est pas utile dans notre problème. Chaque type de chambre ne se distinguera des autres que par le nombre de lits compris dans celle-ci.

Cet ajout va entraîner un certain nombre de contraintes supplémentaires. Par exemple, un objet chambre étant en relation d'appartenance avec un objet catégorie_chambre, dont l'identifiant a pour valeur chambre_double, ne devra être en relation qu'avec deux et seulement deux objets lit. Cette contrainte d'intégrité sera contrôlée lors de la création d'objets chambre.

De cette manière, si la direction de l'hôpital décide de créer des chambres à 10 lits, il suffira de créer une nouvelle instance de la classe Catégorie_chambre et de vérifier les conditions de création de la chambre en utilisant la catégorie pour connaître le nombre de lits devant appartenir à cet objet chambre. Une autre contrainte est que chaque objet chambre doit être en relation avec un objet catégorie_chambre.

Nous ne trouvons pas d'autres classes à mettre en relation avec les quatre précitées. Nous allons donc nous y attarder un peu. En effet, ces quatre classes forment un certain concept propre au domaine du problème: la configuration de

l'hôpital. Sa présence nous permet de constituer, avec ces quatre classes, un sous-système. En effet, un sous-système a pour but de rassembler des classes au sein d'une unité logique de l'espace du problème.

Nous appellerons ce sous-système : Configuration de l'hôpital.

9.3. Le sous-système Configuration de l'hôpital

Comme nous avons une certaine unité logique, nous allons continuer l'analyse de ses composantes.

9.3.1. Identification des attributs

Nous pouvons ajouter un certain nombre d'attributs représentant la mémoire des classes. Afin de réaliser cette identification, nous recherchons les caractéristiques des classes. C'est ainsi que Hôpital a toujours un nom. Ce nom permettra de faire la distinction entre les différents objets de cette classe. Un tel attribut sera appelé identifiant. Chaque classe possède un identifiant qui distingue les objets de la classe étudiée c'est l'une des première caractéristiques à rechercher (nom, numéro, nom et prénom, etc.). Nous rechercherons ensuite les caractéristiques propres à la classe étudiée, les éléments à mémoriser, etc.

Dans notre exemple, nous avons la classe Hôpital qui possède un nom (nom_hop) comme identifiant , la classe Service_soins avec un nom de service (ser), la classe Chambre avec un numéro de chambre (nr_ch), la classe Catégorie_chambre avec un nom de catégorie (cat) et la classe Lit avec un numéro de lit (nr_lit) comme identifiant.

Le nom de Hôpital sera une chaîne de caractères, ainsi que le nom du Service_soins et la Catégorie_chambre. Le numéro de la chambre et le numéro du lit seront des entiers.

Dans la suite de notre analyse, il faudra éventuellement ajouter de nouveaux attributs. En effet, nous n'avons pas encore toutes les données en main en ce qui concerne les différentes classes et les différents attributs devant être mémorisés ou utilisés par l'application.

En reprenant les spécifications données dans l'énoncé des différentes entités, nous pouvons ajouter à la classe Hôpital la date de dernière facture (date_der_fact), indiquant la date à laquelle la dernière facturation a été réalisée. De plus, nous ajouterons à la classe Catégorie_chambre l'attribut prix_plein_lit_j, un réel, qui donne le prix d'un séjour d'une journée dans un lit de cette catégorie.

Après nous être intéressé aux attributs, nous allons rechercher les opérations définies sur ces classes.

9.3.2. Identification des opérations de base

Nous avons déjà relevé une de ces opérations ci-dessus: Création. Ce n'est pas une surprise étant donné que toute classe possède une opération de création, une opération de consultation pour chaque attribut défini, entre autres l'identifiant, et une opération de destruction.

Ces opérations se retrouvent dans toutes les classes sans exception. A côté de ces opérations obligatoires, nous trouvons les opérations propres à la classe concernée.

Dans notre cas, nous allons commencer par la classe Hôpital. Nous avons donc l'opération Création, Affiche_id et Destruction.

L'opération Création va comporter un paramètre: le nom de l'objet de la classe Hôpital qu'il faut créer. Nous aurons donc comme appel d'opération Hôpital.Création(nom_hop). Cette opération est décrite selon ses pré- et post-conditions exprimées selon la logique des prédicats du premier ordre.

Création (nom_h)	nom_h : STRING
* création d'un objet hôpital	
PRE : nom_h ∉ Affiche_id ( )	
POST: nom_h ∈ Affiche_id ( )	

L'opération Affiche_id, appelée par Hôpital.Affiche_id, va simplement retourner à l'objet demandeur un ensemble de chaînes de caratères contenant les valeurs de tous les noms d'objets de la classe Hôpital (nom_hop) actuellement présents dans le système. En effet, l'accès individuel aux objets est impossible si on ne donne pas l'attribut identifiant de l'objet recherché.

Affiche_id ( ) = liste_hôpital	liste_hôpital : SET [STRING]
* retourne une liste contenant les noms des hôpitaux créés.	
PRE : /	nom : STRING
POST : (nom ∈ liste_hôpital) ⇔ In(Hôpital(nom))	

Nous aurons deux opérations en ce qui concerne l'attribut date_der_fact: une opération de consultation et une opération de modification.

Date_dern_fact (nom_h) = date	nom_h : STRING date : DATE
* retourne la date de dernière facturation.	
PRE : nom_h ∈ Affiche_id ( )	
POST : date = Date_der_fact(Hôpital (nom_h))	

Tandis que l'opération de modification sera:

Mod_date_der_fact (nom_h, date) nom_h : STRING
date : DATE
* modifie la date de dernière facturation en remplaçant la valeur par celle
* donnée en argument.
PRE : nom_h ∈ Affiche_id ( )
POST : Date_der_fact(Hôpital (nom_h)) = date

L'opération Destruction va détruire l'objet dont l'identifiant possède la même valeur que le paramètre qui lui est associé. Nous aurons ainsi:

Destruction (nom_h) nom_h : STRING
* destruction d'un objet hôpital
PRE : nom_h ∈ Affiche_id ( )
POST: nom_h ∉ Affiche_id ( )

Notons que nous ne pouvons définir aucune opération de modification des identifiants. Ceux-ci devant être uniques et devant servir à l'accès aux objets, il ne serait pas bon d'avoir de telles opérations. En effet, des objets pourraient, par accident, prendre la même valeur d'identifiant qu'un autre ou perdre leur valeur d'identifiant et devenir, de ce fait, inaccessibles. Pour modifier un identifiant, il faut détruire l'objet et en créer un nouveau avec la nouvelle valeur d'identifiant.

De ces opérations nous pouvons déduire que le diagramme d'état de la classe Hôpital est réduit au plus haut point: nous avons un état de début juste au moment qui précède l'opération Création, un état stable dans lequel se trouve Hôpital durant sa vie, puis un état de fin lorsqu'on détruit l'objet. Le passage de l'un à l'autre se réalise grâce aux opérations Création et Destruction.

Nom : Hôpital

Description : Organisme assurant des soins médicaux à des malades lors d'une ou de plusieurs hospitalisations en son sein.

Interface.

Relations :
Composé de : Service_soins (1-N)
Composant de: /
Généralisation de: /
Spécialisation de : /
Utilise :

Attribut Identifiant: nom_hop : STRING

Attributs: date_der_fact : DATE

Opérations :

création (nom: STRING)
* création d'un objet hôpital

Affiche_id ( ) = liste: SET[STRING]
* retourne une liste contenant les noms des hôpitaux créés.

Date_dern_fact (nom_h: STRING) = date : DATE
* retourne la date de dernière facturation.

Mod_date_der_fact (nom_h: STRING , date: DATE)
* modifie la date de dernière facturation en remplaçant la valeur par celle
*donnée en argument.

Destruction (nom_h: STRING)
* destruction d'un objet hôpital

Diagramme d'état : DET_Hôpital

Implémentation:

< ... >

Nous avons donc indiqué un certain nombre d'opérations pour la classe Hôpital. Nous pourrions refaire les mêmes développements pour la classe Service_soins, Chambre, Lit.

Etant donné la complexité introduite dans la classe Chambre, nous allons également détailler ses opérations.

Nous retrouvons donc les opérations Création, Affiche_id et Destruction.

L'opération Création va comporter deux paramètres: le numéro de l'objet de la classe Chambre qu'il faut créer et le type de la chambre. Nous aurons donc, comme appel Chambre.Création(nr_ch, cat_ch).

Création (numéro_ch, cat_ch) numéro_ch, : INT
cat_ch : STRING

* création d'un objet chambre d'une certaine catégorie

PRE : num_ch ∉ Affiche_id ( )
∧ cat_ch ∈ Catégorie_chambre.Affiche_id ( )

POST: num_ch ∈ Affiche_id ( )

L'opération Affiche_id, appelée par Chambre.Affiche_id, va simplement retourner à l'objet demandeur un ensemble d'entiers contenant les valeurs de tous les numéros d'objets de la classe Chambre actuellement présents dans le système.

```
Affiche_id ( ) = liste_ch                                   liste_ch : SET [INT]
    * retourne une liste contenant les numéros des chambres existantes.
    PRE :    /                                              num : INT
    POST :   (num ∈ liste_ch) ⇔ ( In (chambre (num)))
```

L'opération Destruction va détruire l'objet dont l'identifiant possède la même valeur que le paramètre qui lui est associé. Nous aurons Chambre.Destruction(nr_ch).

```
Destruction (num_ch)                                        num_ch : INT
    * destruction d'un objet chambre
    PRE :    num_ch ∈ Affiche_id ( )
    POST:    num_ch ∉ Affiche_id ( )
```

Ces opérations nous permettent de définir un diagramme d'état également relativement réduit. Nous retrouvons les trois états tels que définis pour Hôpital.

Nous allons donner rapidement les autres opérations définies sur les autres classes de ce sous-système:

- la classe Service_soins,

```
Création (nom_ser)                                          nom_ser : STRING
    * création d'un objet service de soins
    PRE :    nom_ser ∉ Affiche_id ( )
    POST:    nom_ser ∈ Affiche_id ( )
Affiche_id ( ) = liste_ser                                  liste_ser : SET [STRING]
    * retourne une liste contenant les noms des services de soins existants
    PRE :    /                                              nom : STRING
    POST :   (nom ∈ liste_ser) ⇔ nom_ser ∈ Affiche_id ( )
Destruction (nom_ser)                                       nom_ser : STRING
    * destruction d'un objet service de soins
    PRE :    nom_ser ∈ Affiche_id ( )
    POST:    nom_ser ∉ Affiche_id ( )
```

- la classe Catégorie_chambre, pour laquelle nous définirons de manière similaire les trois opérations ci-dessus ainsi que deux autres pour

l'accès et la modification de l'attribut prix_plein_lit_j. Nous ne donnerons que ces dernières.

```
Prix_plein_lit (nom_cat) = prix                    nom_cat : STRING
                                                   prix : REAL
    * retourne le prix pour un séjour d'une journée dans un lit de cette
    * catégorie de chambre.
    PRE :    nom_cat ∈ Affiche_id ( )
    POST :   prix = Prix_plein_lit_j(Catégorie_chambre (nom_cat))
Mod_prix_plein_lit (nom_cat, prix)                 nom_cat : STRING
                                                   prix : REAL
    * modifie le prix d'un séjour d'une journée dans un lit d'une chambre de
    * cette catégorie. Le nouveau prix est donné en argument.
    PRE :    nom_cat ∈ Affiche_id ( )
    POST :   Prix_plein_lit_j(Catégorie_chambre (nom_cat)) = prix
```

- la classe Lit pour laquelle nous aurons, de manière similaire les trois opérations de base puisque nous n'avons pas d'autres attributs.

Pour chacun de ces objets, étant donné les opérations définies jusqu'ici, nous n'avons pas d'autre diagramme de transition d'état que le simpliste qui consiste en trois états, comme le diagramme de la classe Hôpital. Lorsque nous définirons d'autres opérations, nous identifierons d'autres états. Le diagramme sera alors modifié pour rendre compte de ces modifications.

Nous pouvons considérer que ce sous-système est plus ou moins complet quant aux définitions et aux fonctions de base que ses objets doivent assurer. Nous allons maintenant continuer la recherche de classes d'objets.

9.4. Identification des autres classes

Nous voyons rapidement apparaître la notion de patient, d'organisme assureur, de mouvement, de médecin, de prescription de soins et de prestation. Nous reprenons en fait le schéma Entité/Association qui nous est donné dans l'énoncé. Une partie de la phase d'analyse est en fait déjà réalisée par la construction de ce schéma et par la définition précise des entités et des associations. Normalement, le cahier des charges ne comprend pas ce type de schéma, mais nous manquons de trop d'informations pour nous en passer. Nous allons donc réaliser une transformation de ce schéma Entité/Association vers le modèle objet. Nous effectuons donc une sorte de "Mapping" entre ces deux modélisations.

Chaque type d'entité va être transformé en classe, tandis que les associations le seront soit en classes, soit en relations d'agrégation, de spécialisation ou en simples relations d'utilisation. Nous allons nous attarder quelque peu sur ces associations.

Nous allons transformer une association possédant des attributs en classe si les connectivités avec l'une des entités en relation sont autres que 1-1. Dans le cas contraire nous ne créerons pas de classe, mais placerons les attributs dans la classe se trouvant à l'extrémité possédant cette connectivité.

Pourquoi? Dans le cas où les connectivités sont autres que 1-1 pour les deux branches de l'association, nous n'avons pas le choix, car les deux entités ne possèdent pas toujours de relation avec l'autre. Par exemple, dans le cas de l'association Remboursement_prestation, nous avons un attribut prix_remb_prest et aucune des connectivités n'est 1-1. Cela signifie qu'une prestation n'est pas toujours remboursée par tous les organismes assureurs, il se pourrait qu'un de ceux-ci n'ait jamais eu une prestation particulière à rembourser. Nous utiliserons donc une classe Remboursement_prestation et deux relations d'utilisation entre cette classe et les classes Organisme_assureur et Prestation. Nous placerons sur ces relations les connectivités adéquates respectant les spécifications.

De plus, comme cette association n'a pas d'identifiant autre que les rôles joués par les deux entités, ce qui est généralement le cas, nous allons devoir placer, comme attribut identifiant dans cette nouvelle classe, les deux attributs identifiants des entités en relation; dans notre cas, le numéro d'organisme assureur et le code de prestation. C'est le couple de ces deux attributs qui forme l'identifiant de la classe nouvellement créée. Nous ajouterons de nouvelles contraintes d'intégrité pour vérifier cette redondance nécessaire. De cette manière, nous obtenons une classe qui va mémoriser la liste des organismes assureurs et la liste des prestations avec le prix que chaque organisme assureur rembourse pour chaque prestation acceptée par celui-ci.

Un autre exemple est l'association Remboursement_chambre qui sera transformée en classe. Nous créerons également deux relations d'utilisation entre cette classe et les classes Organisme_assureur et Catégorie_chambre. L'identifiant de cette classe est le numéro d'organisme assureur et le nom de la catégorie de chambre.

Dans le cas où la connectivité d'une des deux branches est 1-1, la création de classe n'est pas nécessaire; il suffit de placer les attributs de cette association dans la classe participant à l'association et se trouvant sur la branche possédant cette connectivité. Nous remplaçons l'association par la relation adéquate, soit une relation d'agrégation, soit une relation d'utilisation.

Tel va être le cas pour l'association Affiliation entre l'entité Organisme assureur et l'entité Patient. Nous obtiendrons une classe Patient, une classe Organisme_assureur et une relation d'agrégation entre ces deux classes. En effet, un organisme assureur est composé d'un certain nombre d'affiliés, c'est-à-dire de patients. Chaque patient va posséder un numéro de matricule de titulaire d'assurance et un type d'affiliation. Ces deux attributs sont propres à chaque patient. Nous les placerons donc dans la classe Patient.

Si une association ne possède pas d'attribut, nous allons la transformer soit en relation d'agrégation, soit en relation d'utilisation ou soit en relation de spécialisation. Pour faire ce choix, il convient de s'interroger sur la signification de la relation, sur les données qu'elle représente et sur la possible composition des classes y participant. En ce qui concerne la relation de spécialisation, elle correspond à une association de sous-typage.

Ces transformations permettent de limiter le nombre de classes. Plus le nombre de classes est important, plus nous aurons de mal à appréhender le modèle dans son entier. Nous aurons également un nombre plus important d'attributs et d'opérations à

modéliser et à décrire, etc. La notion de sous-système ne permet que la réduction du nombre de classes et de relations visualisées simultanément.

9.5. Identification des autres sous-systèmes

Dans notre application, nous pouvons définir trois sous-systèmes. Nous en avons déjà identifié un : la Configuration de l'hôpital qui comportait les classes Hôpital, Service_soins, Chambre, Catégorie_chambre, Lit.

Nous allons en définir un deuxième: le Malade. Celui-ci va comporter les classes Patient, Organisme_assureur, Remboursement_chambre et Remboursement_ prestation. Toutes ces classes ont un rapport direct avec le malade. Le patient et son organisme assureur sont liés par une relation d'agrégation. Ces relations d'agrégation permettent d'identifier des groupes de classes unies logiquement.

La classe Remboursement_chambre sera placée dans ce sous-système plutôt que dans le sous-système Configuration de l'hôpital car il nous semble que cette classe se rapporte plus à une des fonctions principales de l'organisme assureur qui est de rembourser les malades des frais de leur hospitalisation. Nous agirons de la même façon pour la classe Remboursement_prestation.

De ces deux sous-systèmes, nous voyons apparaître le troisième: les Actes médicaux. Nous avons en effet le médecin qui demande qu'un patient effectue un mouvement, qui prescrit des soins qui seront ensuite prestés. Cette unité va donc bien s'intégrer dans l'ensemble du système.

De par la signification de la classe Prescription, nous pouvons déduire qu'une prescription est composée d'une prestation. Cette remarque pourrait nous amener à définir une relation d'agrégation entre les prescriptions, classe agrégée et les prestations, classe composante. Cependant, une prestation peut appartenir à plusieurs prescriptions de soins, tandis que la prescription de soins n'est en relation qu'avec une seule prestation. Nous n'avons donc pas d'agrégation ici, mais une relation d'utilisation dont la sémantique est un peu plus riche, étant donné les classes mises en cause. Les contraintes de connectivité pour cette relation seront : une prescription est en relation de connectivité 1-1 avec les prestations et une prestation est en relation de connectivité 0-N avec les prescriptions.

La classe Médecin va être en relation d'utilisation avec la classe Mouvement et avec la classe Prescription. En effet, nous n'avons ni relations d'agrégation, ni relations de spécialisation entre celles-ci. De plus, le fait qu'un ensemble de prescriptions soient en relation avec un médecin sera représenté sur la relation d'utilisation par les connectivités.

Nous avons trois unités logiques cohérentes vis-à-vis du comportement général du système étudié: le médecin prescrit à un patient un ensemble d'actes médicaux au sein d'un hôpital, ces actes étant payés en partie par le malade et en partie par son organisme assureur. Nous retrouvons donc bien nos trois sous-systèmes comme étant des unités logiques distinctes au sein de notre application.

Le premier sous-système a déjà fait l'objet d'une étude un peu plus poussée, nous allons aborder les deux autres.

9.6. Le sous-système Malade

Nous avons donc 4 classes: Patient, Organisme_assureur, Remboursement_chambre et Remboursement_prestation. Sur chacune de ces classes, nous allons retrouver un certain nombre d'attributs et d'opérations. Ceux-ci proviennent directement de l'énoncé du problème ou des transformations des associations.

9.6.1. Identification des attributs

Nous aurons pour la classe Patient:

- le numéro de dossier, attribut identifiant, (nr_dos: INT) ,
- le nom du patient (nom_pat: STRING),
- le prénom du patient (pré_pat : STRING),
- la date de naissance du patient (dnais : DATE).

Ces trois derniers attributs forment un identifiant secondaire. Nous allons les regrouper dans un attribut décomposable : les coordonnées du patient (coord_pat : CP[nom_pat, pré_pat, dnais]),

- l'adresse du patient (adr_pat: STRING),
- le numéro de téléphone du patient (phone: STRING),
- le sexe (sexe: CHAR), possède deux valeurs (M ou F),
- l'état civil (état_civ : STRING),
- la date de sa dernière sortie de l'hôpital (date_sor: DATE),
- la durée de sa dernière hospitalisation (durée_séj: INT).

Certains attributs vont prendre une valeur nulle jusqu'au moment où le patient sortira de sa première hospitalisation. Tous ces attributs sont obligatoires, c'est-à-dire qu'ils doivent tous posséder une valeur différente de la valeur nulle à l'exception du cas évoqué ci-dessus.

Suite à la modification de l'association Affiliation, nous aurons en plus les attributs:

- le numéro de matricule du titulaire (nr_tit: REAL),
- le type d'affiliation (type_aff: STRING).

Pour la classe Organisme_assureur:

- le numéro d'organisme assureur (nr_oa: INT), identifiant,
- le nom de l'organisme assureur (nom_oa: STRING),
- l'adresse de l'organisme assureur (adr_oa: STRING).

Pour la classe Remboursement_chambre:

- le prix remboursé par l'organisme assureur pour un séjour d'une journée dans un lit d'une chambre de cette catégorie (prix_remb_lit_j: REAL),

Cette classe n'a pas d'attribut identifiant. Nous allons donc ajouter les deux identifiants des deux classes en relation avec cette nouvelle classe afin de constituer un couple identifiant pour celle-ci.

- le numéro d'organisme assureur (nr_oa: INT),
- le nom de la catégorie de chambre (cat: STRING).

Nous attirons l'attention du lecteur sur le fait que c'est le couple (nr_oa, cat) qui est identifiant et non pas chacun de ces deux attributs.

Nous suivrons le même raisonnement en ce qui concerne la classe Remboursement_prestation.

- le prix remboursé par l'organisme assureur pour cette prestation (prix_remb_prest : REAL).

Le couple identifiant pour celle-ci sera:

- le numéro d'organisme assureur (nr_oa: INT),
- le code de la prestation (code_prest: INT).

Une fois les attributs déterminés, nous pouvons passer à la détermination des opérations.

9.6.2. Identification des opérations de base

Nous commencerons par la classe Patient.

Nous avons, comme chaque fois, trois opérations de base: Création, Affiche_id et Destruction.

L'opération Création. Cette opération va comporter un paramètre: les coordonnées de l'objet patient que l'on désire créer. C'est-à-dire le nom, le prénom et la date de naissance. Comme ces coordonnées sont identifiantes, nous pourrons vérifier si le patient n'existe pas déjà et donc, de ce fait, vérifier la contrainte d'intégrité de l'identifiant. Nous aurons donc comme appel Patient.Création(coord).

Création (coord)	coord: CP[nom_pat: STRING, pré_pat: STRING, dnais: DATE]
* création d'un objet patient PRE : ¬ In (Patient (coord)) POST: In (Patient (coord))	

Comme nous avons deux identifiants, nous aurons deux opérations d'affichage d'identifiants:

```
Affiche_id_coord ( ) = liste_pat                    liste_pat : SET [
                                                    CP[nom_pat: STRING,
                                                       pré_pat: STRING,
                                                       dnais: DATE]]
    * retourne une liste contenant les coordonnées des patients.
    PRE :    /
    POST :   (coord ∈ liste_pat) ⇔ ( In (Patient (coord)))
Affiche_id_nr_dos ( ) = liste_pat                   liste_pat : SET [nr_dos : INT]
    * retourne une liste contenant les numéros de dossier des patients.
    PRE :    /                                      num_pat : INT
    POST :   (num_pat ∈ liste_pat) ⇔ ( In (Patient (num_pat)))
```

Nous aurons une opération de destruction:

```
Destruction (num_pat)                               num_pat : INT
    * destruction d'un objet patient dont le nr_dos est donné
    PRE :    In (Patient (num_pat))
    POST:    ¬ In (Patient (num_pat))
```

L'opération Détermination_nr_dos va déterminer le numéro de dossier d'un patient s'il existe, ou en créer un si nécessaire.

```
Détermination_nr_dos (coord) = nr_dos
    * opération retournant le numéro de dossier pour un patient dont on
    * donne les coordonnées, s'il existe, ou en en créant un si nécessaire.
    PRE : /
    POST:    IF In (Patient (coord))
             THEN nr_dos = Nr_dos(Patient(Coord))
             ELSE ¬ In (Patient (nr_dos))
```

De cette manière, nous renvoyons un numéro de dossier unique à l'opération qui en a demandé l'exécution.

Nous avons donc une partie des opérations de base définies sur la classe Patient. Nous y ajoutons les opérations de consultation et de modification de chacun de ses attributs. Nous n'allons pas les détailler toutes ici. Nous les donnons pour l'attribut adr_pat

Adresse_patient (nr_dos) = adresse — nr_dos : INT, adresse : STRING

* retourne l'adresse du patient possédant le numéro de dossier concerné
PRE : In (Patient (nr_dos))
POST : adresse = Adr_pat(Patient (nr_dos))

Mod_adresse_patient (nr_dos, adresse) — nr_dos : INT, adresse : STRING

* modifie l'adresse du patient possédant le numéro de dossier concerné
PRE : In (Patient (nr_dos))
POST : Adr_pat(Patient (nr_dos)) = adresse

Nous aurons donc un ensemble d'opérations permettant de réaliser ces consultations et modifications.

Les opérations définies sur la classe Organisme_assureur sont les trois opérations de base, plus deux couples d'opérations (consultation, modification) pour les deux attributs autres que l'identifiant.

Les opérations définies sur la classe Remboursement_chambre et sur la classe Remboursement_prestation vont être fort semblables. Nous ne détaillerons que les opérations de l'une d'entre elles: la classe Remboursement_chambre. Nous définirons les opérations de création, de consultation des identifiants de tous les objets, de destruction ainsi qu'une opération de consultation de l'attribut prix_remb_lit_j et une de modification de celui-ci. Les contraintes d'intégrité peuvent être exprimées dans les pré- et post-conditions par des conditions d'existence dans les objets des classes concernées. Dans notre cas, nous ajouterons des conditions du type : In (Catégorie_chambre(Cat (cat_oa))), par exemple.

Création (cat_oa) — cat_oa: CP[cat : STRING, nr_oa: INT]

* création d'un objet Remboursement_chambre
PRE : ¬ In (Remboursement_chambre (cat_oa))
∧ In (Catégorie_chambre(Cat(cat_oa)))
∧ In (Organisme_assureur(Nr_oa(Cat_oa)))
POST: In (Remboursement_chambre (cat_oa))

Affiche_id ( ) = liste_remb_ch — liste_remb_ch : SET [CP[cat : STRING, nr_oa: INT]

* retourne une liste contenant les couples catégorie de chambre et
* organisme assureur.

PRE : / — cat_oa: CP[cat : STRING, nr_oa: INT]

POST : (cat_oa ∈ liste_remb_ch)
⇔ (In (Remboursement_chambre(cat_oa)))
∧ In (Catégorie_chambre(Cat(cat_oa)))
∧ In (Organisme_assureur(Nr_oa(Cat_oa)))

Destruction (cat_oa) — cat_oa: CP[cat : STRING, nr_oa: INT]

* destruction d'un objet de la classe Remboursement_chambre

PRE : In (Remboursement_chambre (cat_oa))
∧ In (Catégorie_chambre(Cat(cat_oa)))
∧ In (Organisme_assureur(Nr_oa(Cat_oa)))

POST: ¬ In (Remboursement_chambre (cat_oa))

Prix_remb_lit (cat_oa) = prix — cat_oa: CP[cat : STRING, nr_oa: INT]
prix : REAL

* retourne le prix remboursé par cet organisme assureur pour un séjour
* d'une journée dans un lit de cette catégorie de chambre.

PRE : In (Remboursement_chambre (cat_oa))
∧ In (Catégorie_chambre(Cat(cat_oa)))
∧ In (Organisme_assureur(Nr_oa(Cat_oa)))

POST : prix = Prix_remb_lit_j(Remboursement_chambre (cat_oa))

Mod_prix_remb_lit (cat_oa, prix) — cat_oa: CP[cat : STRING, nr_oa: INT]
prix : REAL

* modifie le prix remboursé pour un séjour d'une journée dans un lit
* d'une chambre de cette catégorie. Le nouveau prix est donné en
* argument.

PRE : In (Remboursement_chambre (cat_oa))
∧ In (Catégorie_chambre(Cat(cat_oa)))
∧ In (Organisme_assureur(Nr_oa(Cat_oa)))

POST : Prix_remb_lit_j(Remboursement_chambre (cat_oa)) = prix

Le sous-système Malade est complet en ce qui concerne les opérations de base. Nous allons passer au troisième sous-système.

9.7. Le sous-système Actes médicaux

Dans ce sous-système, nous avons quatre classes: Médecin, Prescriptions_ soins, Prestations et Mouvements. Nous commencerons par identifier les attributs, puis nous identifierons les opérations de base.

9.7.1. Identification des attributs

Pour cette identification, nous reprenons les définitions données dans l'énoncé.

La classe Médecin décrit toute personne ayant le droit de prescrire une prestation ou un mouvement et possède comme attributs:

- le numéro du médecin (nr_med : INT), identifiant,
- le nom du médecin (nom_med : STRING),
- le prénom du médecin (pré_med : STRING),
- l'adresse du médecin (adr_med : STRING).

La classe Mouvement décrit le changement d'attribution de lit à une personne et possède comme attributs:

- le numéro du mouvement (nr_mv : INT), identifiant,
- la date du mouvement (date_mv : DATE) ,
- le type de mouvement (type_mv : STRING) dont le domaine de valeur est {entrée, transfert, sortie}.

La classe Prescription_soins décrit la prescription par un médecin d'une prestation à administrer à un malade et possède comme attributs :

- le numéro d'ordre de la prescription(nr_pr: INT), identifiant,
- la date prévue de l'exécution de la prescription (date_pr : DATE),
- l'état actuel de la prescription (état_pr : STRING), dont le domaine de valeur est {en attente, exécutée}.

et enfin, la classe Prestation qui décrit le type de soins ou de médicaments pouvant être donnés à un malade et donnant lieu à une facturation.. Elle possède comme attributs:

- le code de la prestation (code_prest : STRING), identifiant,
- le nom de la prestation (nom_prest : STRING),
- le prix plein unitaire de la prestation (prix_plein_prest : REAL).

9.7.2. Identification des opérations de base

Les opérations de base sont les mêmes : la création, l'affichage d'une liste des objets existants et la destruction de l'objet. De plus, nous aurons, des opérations de consultation et de modification pour chacun des attributs identifiés sur ces objets, à l'exception de l'identifiant.

Nous pourrions les reprendre, mais elles sont similaires aux déclarations réalisées ci-dessus. La relation d'agrégation ne sera pas traduite directement dans les pré- et post-conditions de création d'un objet prescription_soins. Celle-ci sera prise en compte lorsque l'on connaîtra le système dans lequel l'application sera implémentée et

donc quand on connaîtra la manière dont celui-ci permet de traiter ce problème. Dans la phase d'analyse, nous avons essentiellement une identification des relations.

Une fois ces trois sous-systèmes identifiés, nous allons analyser les différentes fonctionnalités attendues du système et voir comment elles vont se répartir sur les objets.

9.8. Les fonctionnalités attendues

Chacune de ces fonctionnalités a un objet de départ sur lequel une opération va s'effectuer. Celle-ci va demander que s'exécute, sur l'objet ou sur d'autres objets du système, un ensemble d'opérations. Cet ensemble va constituer et réaliser une fonctionnalité attendue du système.

L'énoncé de départ nous en offre un certain nombre. Etant donné que le but recherché est surtout une illustration de l'utilisation de la méthode proposée aux chapitres précédents, nous ne développerons pas toutes les fonctionnalités que comporte ce cas. Nous allons essentiellement nous attarder à l'admission d'un patient dans l'hôpital. Nous allons reprendre ci-dessous cette fonctionnalité qui sera vue comme une opération d'une classe utilisant les opérations définies sur les autres.

Cette fonctionnalité va être déclenchée sur un objet de la classe Patient. Elle reçoit un certain nombre d'arguments et retourne les trois messages prévus dans les spécifications.

```
Admission (arg, date_jour) = res          arg : CP[
                                          Coord : CP[
                                             Nom_pat: STRING,
                                             Pré_pat: STRING,
                                             Dnais : DATE] ,
                                          Ser: STRING,
                                          Cat : STRING,
                                          Nr_med: INT,
                                          Info : CP[
                                             Adr_pat : STRING,
                                             Phone : STRING,
                                             Sexe : CHAR,
                                             Etat_civ : STRING,
                                             Nr_tit : STRING,
                                             Type_aff : STRING],
                                          Nr_oa : INT]
                                          Date_jour : DATE
                                          res : CP[Nr_dos : INT,
                                                Nr_lit : INT]
PRE :     Ser (arg) ∈Service_soins.Affiche_id( )
          ∧ Cat(arg) ∈ Catégorie_chambre.Affiche_id( )
          ∧ Nr_med(arg) ∈ Medecin.Affiche_id( )
          ∧ Nr_oa(arg) ∈ Organisme_assureur.Affiche_id( )
          ∧ ¬ Mouvement.Présence_mv(Coord(arg),Date_jour)
```

POST: (¬ Malade (Coord(arg))
∧ ¬ Service.Service_plein (Ser(arg))
⇒ (Nr_lit (res) = Lit.Allocation_lit (Ser(arg), Cat(arg))
∧ Nr_dos (res) = Détermination_nr_dos (Coord(arg))
∧ In (Patient (Nr_dos(res))
∧ Coord (Patient(Nr_dos(res))) = Coord (arg)
∧ Info (Patient (Nr_dos (res))) = Info (arg)
∧ ∃ nr_mvt : [nr_mvt ∈Mouvement.Affiche_id( )
∧ Mouvement.Date_mv(nr_mv) = date_jour
∧ Mouvement.type_mv(nr_mv) = "Entrée"])

EXCEPTIONS:
Mess1 = "Le service est plein"
Mess2 = "Le patient est déjà malade"

CONDITIONS_EXCEPTIONS:
Mess1 si Service.Service_plein (Ser(arg))
Mess2 si Malade (Coord(arg))

Suite à cette définition de fonctionnalité, nous voyons apparaître de nouvelles opérations à définir sur les classes concernées:

- sur la classe Mouvement: l'opération Présence_mv (coord, date_jour),
- sur la classe Service: l'opération Service_plein (ser) et
  Allocation_lit (ser, cat),
- sur la classe Patient : l'opération Malade (coord).

Nous allons les ajouter aux différentes classes.

Dans la classe Mouvement :

Présence_mv (coord, date)

coord : CP[
Nom_pat: STRING,
Pré_pat: STRING,
Dnais : DATE] ,
date : DATE

* Test de l'existence d'un mouvement pour un certain patient à une date
* donnée
PRE :
POST : coord ∈Patient.Affiche_id_coord( )
∧ ∃ nr_mvt : (nr_mvt ∈ Accomplit(coord, nr_mvt)
∧ Date(Mouvement (nr_mvt)) = date)

Dans cette définition, nous voyons apparaître "Accomplit(coord, nr_mvt)". Cette expression permet de modéliser la relation d'utilisation existant entre Mouvement et Patient. Le nom Accomplit correspond au nom de la relation d'utilisation. Pour connaître l'existence d'une telle relation, nous donnons le nom de celle-ci ainsi que les identifiants des classes en relation. L'expression booléenne précise l'existence ou l'absence d'instance de cette relation.

Dans la classe Service:

- l'opération Service_plein (ser)

```
Service_plein (ser)                                    ser: STRING,
    * Test de l'existence d'un lit libre à l'intérieur d'un service
    PRE :     In (Service(ser))
    POST :    ¬ ∃ nr_lit : ( nr_lit ∈ Lit.Affiche_id( )
                ∧ ( ∃ nr_ch : Contient (nr_ch, nr_lit)
                      ∧ Possède (ser, nr_ch))
                      ∧ Lit.Lit_vide(nr_lit) )
```

Nous devrons introduire dans l'objet lit, l'opération Lit_vide(nr_lit) qui permet de déterminer si l'objet nr_lit de la classe Lit est actuellement occupé ou non.

De plus, cette opération nous indique qu'un objet service peut se trouver dans deux états différents: un état plein et un état non plein. Le passage de l'un à l'autre se réalise en fonction de la valeur booléenne de l'opération Service_plein.

- l'opération Allocation_lit (ser, cat)

```
Allocation_lit (ser, cat)= nr_lit                      ser: STRING,
                                                       cat: STRING,
                                                       nr_lit : INT
    * opération retournant un numéro de lit libre dans un service donné et si
    * possible, dans une chambre correspondant à la catégorie donnée.
    PRE :     ser ∈ Affiche_id ( )
              ∧ cat  ∈ Chambre.Affiche_id ( )
              ∧ ¬ Service_plein (ser)
    POST :    ∃ nr_lit : ( nr_lit ∈ Lit.Affiche_id( )
                    ∧ ( ∃ nr_ch : Contient (nr_ch, nr_lit)
                      ∧ Possède (ser, nr_ch) )
                      ∧ Lit.Lit_vide(nr_lit) )
              ∧ ∃ nr_lit' : ( nr_lit' ∈ Lit.Affiche_id( )
                ∧( ∃ nr_ch : Contient (nr_ch, nr_lit')
                  ∧ Possède (ser, nr_ch) )
                  ∧ Lit.Lit_vide(nr_lit')
                  ∧ Catégorie_chambre.Catégorie_lit(nr_lit', cat) )
              ⇒ Catégorie_chambre.Catégorie_lit(nr_lit, cat)
```

Nous avons utilisé ici, l'opération booléenne Catégorie_lit (nr_lit, cat) définie sur la classe Catégorie_chambre qui permet de déterminer si le lit dont on donne le numéro se trouve dans une chambre de catégorie donnée.

Dans la classe Patient :

- l'opération Malade (coord).

Malade (coord) coord : CP[
Nom_pat: STRING,
Pré_pat: STRING,
Dnais : DATE]

* Teste si un patient ayant les coordonnées données est actuellement
* malade dans l'hôpital
PRE : /
POST : coord ∈Patient.Affiche_id_coord( )
∧ ∃ nr_mvt : Mouvement.Dernier_mv (nr_mvt, coord)
∧ nr_mvt ∈ Mouvement.Affiche_id ( )
∧ Type_mv(Mouvement (nr_mvt)) ≠ "sortie")

A nouveau, nous utilisons une opération définie sur la classe Mouvement, Dernier_mv (nr_mv, coord) qui permet de dire si le mouvement dont l'identifiant à la même valeur que nr_mv est le dernier effectué par le patient dont on donne les coordonnées.

Nous voyons apparaître un deuxième état pour les objets patient: soit que le patient considéré est actuellement malade et donc se trouve dans l'hôpital, soit qu'il est un ancien malade. Le passage de l'état malade à l'état ancien malade se réalisera lorsque le patient effectuera un mouvement de type "sortie". Tandis que l'on passera de l'état ancien malade à l'état malade lorsque le patient effectuera un mouvement de type "entrée". De plus, la valeur de l'opération Malade changera. Si celle-ci est vraie, le patient concerné se trouve dans l'état malade, sinon, il se trouve dans l'état ancien malade. Les nouveaux patients entrent directement dans l'état malade puisqu'ils effectuent un mouvement de type "entrée".

Nous allons définir les opérations nouvellement utilisées:

- dans la classe Lit:
  - l'opération Lit_vide (nr_lit)

```
Lit_vide (nr_lit)                                        nr_lit : INT
    * Teste si le lit est actuellement libre.
    PRE :  nr_lit ∈ Affiche_id ( )
    POST :   ∀ nr_mvt :  nr_mvt ∈ Mouvement.Affiche_id( )
              ∧ (nr_mvt  ∈ (Destination(nr_lit, nr_mvt)
                     ∪ Origine(nr_lit, nr_mvt))
               ∧ ∀  nr_mvt' : nr_mvt'  ∈ (Destination(nr_lit, nr_mvt')
                     ∪ Origine(nr_lit, nr_mvt'))
               ⇒ Mouvement.Date_mv(nr_mvt')
                   < Mouvement.Date_mv(nr_mvt) )
             ⇒( Mouvement.Type_mvt(nr_mvt) = "sortie"
               ∨ ( Mouvement.Type_mvt(nr_mvt) = "transfert"
                   ∧ nr_mvt  ∈ Origine(nr_lit, nr_mvt)))
```

Cette opération nous indique également deux états pour un lit: il est soit libre, soit occupé. Selon le type du dernier mouvement effectué sur ce lit, nous pourrons dire si le lit est vide ou s'il est occupé. Le lit sera libre si le dernier mouvement effectué sur ce lit est de type "sortie" ou s'il est de type "transfert" où le lit est l'origine de ce mouvement. Le lit sera dans l'état occupé lorsque le dernier mouvement est de type "entrée" ou s'il est de type "transfert", mais où le lit est la destination. De plus, lors de la création, le lit va se trouver dans l'état libre. De même, pour qu'un lit puisse être détruit, il faut qu'il soit libre. Nous ajouterons donc une précondition à l'opération Destruction : Lit_libre (nr_lit).

- dans la classe Mouvement:
  - l'opération Dernier_mv (nr_mvt, coord)

```
Dernier_mv (nr_mvt, coord)                       nr_mvt : INT
                                                 coord : CP[
                                                    Nom_pat: STRING,
                                                    Pré_pat: STRING,
                                                    Dnais : DATE]
    * Teste si le mouvement considéré est le dernier du patient donné.
    PRE :    nr_mvt ∈ Affiche_id ( )
             ∧ coord ∈ Patient.Affiche_id ( )
    POST :   nr_mvt ∈ Accomplit (nr_mvt, coord)
             ∧ ∀  nr_mvt' : nr_mvt'  ∈ Accomplit(nr_mvt', coord)
             ⇒ Date_mv(nr_mvt') < Date_mv(nr_mvt) )
```

Un mouvement peut se trouver dans trois états: il est soit ordinaire, soit le dernier, soit le premier de la dernière hospitalisation. Il est le premier de la dernière hospitalisation si c'est le dernier mouvement de type "entrée", c'est-à-dire qu'il a la date la plus élevée dans l'ensemble des mouvements de type "entrée". Il sera dans un état dernier mouvement s'il est le mouvement dont la date est la plus élevée, quelque soit son type. Sinon, il est dans un état ordinaire. Ces différents états n'ont de sens que s'ils sont liés à un patient particulier, c'est-à-dire que nous ne considérerons que les mouvements réalisés par le même patient.

- dans la classe Catégorie_chambre:
  - l'opération Catégorie_lit (nr_lit, cat)

Catégorie_lit (nr_lit, cat)	nr_lit : INT cat: INT
* Teste si le lit considéré est dans la catégorie de chambre donnée.	
PRE : nr_lit ∈ Lit.Affiche_id ( )	
∧ cat ∈ Affiche_id ( )	
POST : ∃ nr_ch : nr_ch ∈ Chambre.Affiche_id ( )	
∧ Contient (nr_ch, nr_lit)	
∧ Catégorie (cat, nr_ch)	

9.9. Conclusions

Cette phase d'analyse nous a permis d'identifier un certain nombre de classes en relations entre elles et de les répartir dans trois sous-systèmes.

Nous avons pu constater le cheminement à la fois incrémental et itératif. Aussi bien pour l'identification des attributs que pour l'identification des opérations.

Ces opérations sont de deux types:

- les opérations de base: Création et Destruction de l'objet et l'accès aux attributs,
- les opérations chargées de remplir un certain nombre de fonctions du système modélisé. Les enchaînements de celles-ci permettent de reconstituer les fonctionnalités que l'on aurait identifiées par une analyse fonctionnelle.

De plus, ce ne sont pas les opérations de base qui déterminent les différents états dans lesquels les objets vont se trouver, mais bien les opérations agissant au sein d'un enchaînement. Celles-ci vont permettrent de définir un certain nombre de situations plus ou moins particulières qui vont nous indiquer les différents états dans lesquels les objets évoluent.

Si nous avons limité volontairement l'importance du problème, c'est pour garder une illustration relativement simple et complète du cas. Avant de passer à la phase de conception, nous aimerions rappeler que la résolution complète du cas avait été réalisée selon la méthode fonctionnelle durant la seconde licence et avait occupé une équipe de 7 personnes pendant environ 4 mois.

Chapitre 10 : La phase de conception

Introduction

Dans la phase de conception, les classes, et surtout les opérations, identifiées lors de la phase d'analyse vont être affinées.

Nous ajouterons la structure de celles-ci sous forme d'un pseudo-langage, le plus simple possible c'est-à-dire n'utilisant que des opérations élémentaires. Il nous faudra respecter le type de langage utilisé au niveau de l'implémentation, dans notre cas, un langage orienté objets.

Nous continuerons en constituant un module d'interface avec l'utilisateur. Le type de cette interface utilisée lors de la phase d'implémentation sera une interface de type Windows avec son gestionnaire. Les interactions entre ce module et nos classes seront définies. Les répercussions sur les classes seront indiquées.

Nous créerons ensuite, à partir du schéma d'objet, un schéma de la base de données utilisée. Notre choix de SGBD s'est porté sur une base de données de type SQL. Le passage du schéma d'objet au schéma de la BD suivra un certain nombre de transformations. Les modifications apportées sur chaque objet seront intégrées dans ceux-ci.

Nous retrouvons à nouveau un processus incrémental et itératif. Chaque étape va venir modifier les classes définies jusque-là et en ajouter de nouvelles. Etant donné que nous disposons de trois sous-systèmes, nous allons réaliser la tâche de conception sur chacun d'eux séparément, puis nous établirons les relations entre eux.

Le fait qu'il n'existe que des relations d'utilisation entre ces sous-systèmes permet de conduire la conception de manière plus ou moins indépendante dans ces derniers, en ne faisant intervenir les autres sous-systèmes que lorsque cela est absolument nécessaire. La construction de la BD est un exemple où ces relations vont jouer un rôle non négligeable.

L'ordre suivi dans la phase d'analyse sera respecté.

10.1. Le sous-système Configuration de l'hôpital

10.1.1. La conception des classes

Nous avons identifié les classes Hôpital, Service_soins, Chambre, Lit et Catégorie_chambre. Nous allons reprendre les différentes opérations définies selon les pré- et post-conditions.

Ces opérations vont être transformées en algorithmes utilisant des instructions simples permettant de préciser exactement ce qui est demandé, sans devoir aller jusqu'au langage d'implémentation. Ces opérations de base vont utiliser les primitives propres au langage de programmation. Nous aurons donc des opérations se ressemblant fortement, seuls les arguments seront différents. Nous ne reprendrons qu'une classe comme exemple: la classe Hôpital.

Nom : Hôpital

Description : Organisme assurant des soins médicaux à des malades lors d'une ou de plusieurs hospitalisations en son sein.

Interface.

Relations :
Composé de : Service_soins (1-N)
Composant de: /
Généralisation de: /
Spécialisation de : /
Utilise :

Attribut Identifiant: nom_hop : STRING

Attributs: date_der_fact : DATE

Opérations :
Création (nom: STRING)
* création d'un objet hôpital

Affiche_id ( ) = liste: SET[STRING]
* retourne une liste contenant les noms des hôpitaux créés.

Date_dern_fact (nom_h: STRING) = date : DATE
* retourne la date de dernière facturation.

Mod_date_der_fact (nom_h: STRING , date: DATE)
* modifie la date de dernière facturation en remplaçant la valeur par celle *donnée en argument..

Destruction (nom_h: STRING)
* destruction d'un objet hôpital

Diagramme d'état : DET_Hôpital

Implémentation:

```
Création (nom_h)      nom_h : STRING
    * création d'un objet hôpital
    PRE :       nom_h ∉ Affiche_id ( )
    POST:       nom_h ∈ Affiche_id ( )

    VAR :       nom_ser: STRING,
                nombre_service: INT

    BEGIN Création
          Création_objet_hôpital_ayant_pour_nom (nom_h),
          nombre_services := Demande_nombre_service_soins_à_créer,
          i :=1
          REPEAT
                nom_service := Demande_nom_objet_service_soins,
                Service_soins.Création(nom_ser)
                Créer_lien_hôpital_service (nom_h, nom_ser)
                i := i+1
          UNTIL i > nombre_services
    END Création.
Affiche_id ( ) = liste_hôpital                    liste_hôpital : SET [STRING]
    * retourne une liste contenant les noms des hôpitaux créés.
    PRE :       /                                 nom : STRING
    POST :      (nom ∈ liste_hôpital) ⇔ In(Hôpital(nom))

    BEGIN Affiche_id
          liste_hôpital := [ ]
          FOR nom := Hôpital ( : nom_hop) DO
                Liste_hôpital := liste_hôpital + Hôpital (: nom)
          END FOR
    END Affiche_id
Date_dern_fact (nom_h) = date                     nom_h : STRING
                                                  date : DATE
    * retourne la date de dernière facturation.
    PRE :       nom_h ∈ Affiche_id ( )
    POST :      date = Date_der_fact(Hôpital (nom_h))

    BEGIN Date_dern_fact
          date := Date_der_fact(Hôpital (: nom_hop = nom_h)
    END Date_dern_fact
```

```
Mod_date_der_fact (nom_h, date)                    nom_h : STRING
                                                   date : DATE
    * modifie la date de dernière facturation en remplaçant la valeur par celle
    *donnée en argument.
    PRE :     nom_h ∈ Affiche_id ( )
    POST :    Date_der_fact(Hôpital (: nom_hop = nom_h)) = date

    BEGIN Mod_date_der_fact
        Date_der_fact(Hôpital (: nom_hop = nom_h)) := date
    END Mod_date_der_fact
Destruction (nom_h)                                nom_h : STRING
    * destruction d'un objet hôpital
    PRE :     nom_h ∈ Affiche_id ( )
    POST:     nom_h ∉ Affiche_id ( )

    VAR       liste_services : SET[STRING]

    BEGIN Destruction
        liste_services := Affiche_nom_ser_de_hôpital (nom_hop),
        REPEAT
            liste_services := Append (liste_services, nom_service)
            Destruction_lien_hôpital_service(nom_hop, nom_service)
            Service_soins.Destruction(nom_service),
        UNTIL EMPTY liste_services
        Destruction_objet_hôpital (nom_hop),
    END Destruction.
```

Il reste un certain nombre d'opérations à définir. Elles sont classées selon qu'elles sont liées au langage d'implémentation, à l'interface avec l'utilisateur ou à la base de données :

- langage d'implémentation:

  Création_objet_hôpital_ayant_pour_nom(nom_h),

  Destruction_objet_hôpital (nom_hop).

- interface utilisateur:

  Demande_nombre_service_soins_à_créer,

  Demande_nom_objet_service_soins.

- base de données:

  Créer_lien_hôpital_service (nom_h, nom_ser),

  Destruction_lien_hôpital_service(nom_hop, nom_service).

- et faisant intervenir les deux derniers:

  Affiche_nom_ser_de_hôpital (nom_hop).

Nous pouvons remarquer que l'opération Affiche_id n'est que partiellement expliquée. En effet, le langage de programmation offre généralement une primitive qui permet d'obtenir la liste des identifiants d'une certaine classe.

Les opérations liées au langage de programmation ne seront pas détaillées plus avant, il suffit généralement d'une seule opération dans ce langage pour les réaliser.

10.1.2. La conception du module d'interface

Nous devons maintenant modéliser les interactions entre le système et l'utilisateur. L'implémentation de cette interface sera réalisée dans un langage d'interface de type Windows. Nous avons donc un certain nombre de primitives prédéfinies à notre disposition. Par exemple, lorsque nous demanderons l'affichage d'un message, le gestionnaire d'interface réalisera toutes les opérations nécessaires pour afficher ce message dans une "fenêtre" à l'écran.

De même, lorsque nous aurons besoin d'une réponse, ou d'une saisie, nous utiliserons une seule opération qui donnera le libellé de la question et qui renverra le paramètre une fois que l'utilisateur aura répondu.

Ce gestionnaire sera représenté par une classe Utilisateur. Cette classe servira d'intermédiaire entre l'application et l'utilisateur, nous permettant d'introduire une certaine indépendance entre l'interface et l'application. La manière dont cette classe et les interactions avec l'utilisateur seront effectivement réalisées ne dépendra que de cette classe et sera "transparente" vis-à-vis de l'application. Nous implémentons donc le principe de modularité et de localisation des modifications. Par exemple, si nous voulons changer la langue employée par l'utilisateur, il nous suffit de changer de classe Utilisateur. De même si nous désirons changer de gestionnaire d'interface pour passer d'un gestionnaire de type Windows (graphique) à un gestionnaire de type Texte.

Les aspects esthétiques et de disposition des différentes fenêtres à l'écran seront réalisés de manière indépendante de notre conception. De cette façon, une équipe spécialisée dans le domaine de l'ergonomie sera chargée de répartir les différentes opérations que nous allons maintenant définir sur la classe Utilisateur.

Cette classe Utilisateur est relativement complexe. En effet, elle va contenir toutes les opérations élémentaires nécessaires aux objets de l'application. C'est ainsi que nous aurons essentiellement de l'affichage d'information à l'écran et des opérations de saisie et de sélection dans une liste. Le reste du traitement des données sera réalisé au niveau des objets demandeurs. De plus, cette classe Utilisateur peut être décomposée selon les moyens d'interaction disponibles. Cette décomposition n'est utile qu'au moment de l'implémentation de cette classe. Généralement, toutes les classes de l'application vont utiliser la classe Utilisateur à un moment ou à un autre, nous n'allons donc pas représenter les messages échangés. Nous considèrerons ces relations comme implicites.

Notre cas s'intéresse à une application mono-utilisateur et mono-poste. La classe Utilisateur ne contiendra donc qu'un seul objet lié à un seul terminal. Cependant il n'est pas impossible de gérer différents utilisateurs et différents postes, la classe

Utilisateur serait alors réévaluée et modifiée, mais également les communications entre les différents objets des différentes classes de l'application. Ceci provoquerait l'utilisation d'un autre langage de programmation, qui serait multi-tâches, et nécessiterait éventuellement certaines modifications dans les différents objets pour rendre compte des problèmes de concurrence qui pourraient se poser. De même une gestion de la concurrence ferait intervenir un SGBD plus complexe que celui qui sera utilisé ici.

Nous allons définir trois opérations types sur cette classe:

- Affiche (information),
- Saisie (question : STRING, info),
- Selection_dans_liste (liste : SET[élément_liste], élément_liste).

Nous pouvons remarquer que nous n'utilisons pas de type pour les paramètres de ces opérations. Si le langage le permet, nous utiliserons un type générique qui nous permettra de ne pas préciser celui des paramètres, simplifiant les appels pour n'avoir que des opérations génériques pouvant être utilisées par tous les objets avec des paramètres de tout type. Un langage faiblement typé (n'imposant pas les types lors de l'exécution) et respectant les types abstraits de données en serait un exemple.

Par contre, si le langage ne le permet pas, nous utiliserons un type intermédiaire et définirons des opérations de transformation de type que nous placerons dans les différents objets. Par exemple, dans le deuxième cas, si nous devons afficher un entier nous le transformerons d'abord en chaîne de caractères. Nous pourrons faire de même avec les informations reçues sous forme de chaînes de caractères en les retransformant en entiers, par exemple.

Dans le même ordre d'idées, nous pourrons utiliser un paramètre composé lui-même de nombreux paramètres, regroupés dans un produit cartésien. Si le langage ne permet pas l'utilisation de types abstraits de données, ce qui est malheureusement souvent le cas, nous pourrons adopter une stratégie différente: constituer un produit cartésien composé des différents paramètres pouvant être transmis à la classe Utilisateur pour y être affiché ou saisi. Nous utiliserons toujours le même paramètre en ne garnissant que les éléments dont nous avons besoin à ce moment-là.

Cette stratégie nécessite une multiplication des opérations de la classe Utilisateur pour permettre l'affichage ou la saisie des bons éléments du paramètre général. Cette multiplication permet de garder un contrôle plus précis de ce qui doit être fait. Nous pourrons de ce fait utiliser des algorithmes particuliers pour optimiser certaines opérations plus spécifiques, et ce, au prix d'une perte de généralité. Un compromis entre la généralité des opérations et leurs performances est donc souvent nécessaire. Par exemple, pour saisir le nom d'un service, nous appellerons l'opération Utilisateur.Saisie_nom_ser ("Quel est le nom du service:", paramètre_général).

Les gestionnaires d'écran, et donc d'interface, utilisent généralement ce type de stratégie. Néanmoins, une certaine efficacité peut être donnée en utilisant toujours ce paramètre et sa référence. De ce fait, nous utilisons toujours le même espace mémoire, nous n'aurons pas de copies multiples de ce paramètre, mais seulement le transfert d'un pointeur vers cet espace.

Au niveau des messages échangés avec la classe Utilisateur, nous devons faire remarquer que si l'on transmet un message ou une question à partir de l'application, nous perdons la localisation des changements lors d'un changement de langage de l'utilisateur. Pour cela, nous allons définir un système de codage dont les libellés seront placés dans la classe Utilisateur, l'application ne communiquant qu'un code en lieu et place de phrases. La classe Utilisateur permettra de "traduire" ce code en fonction de la manière dont cette traduction aura été implémentée sur la classe Utilisateur.

De plus, les personnes développant l'application communiqueront les phrases et les messages à afficher à l'écran au groupe chargé de l'interface utilisateur-application, qui leur retourneront un code représentant cette phrase. Une bonne coordination entre ces équipes est donc nécessaire. Pour une certaine facilité de lecture et étant donné que cette codification a essentiellement lieu au moment de l'implémentation, nous exprimerons ces messages sans utiliser de code, c'est-à-dire tels quels en français.

Ces considérations sont essentiellement discutées et mises en oeuvre au niveau de l'implémentation en fonction du type de langage que l'on aura choisi pour la réaliser.

Dans notre cas, et plus particulièrement dans la classe Hôpital, nous allons pouvoir remplacer les opérations faisant intervenir cette classe Utilisateur par les appels réels à celle-ci.

Dans la classe Hôpital, nous allons remplacer:

- la ligne comportant l'opération "Demande_nombre_service_soins_à_créer" par les lignes suivantes:

  Utilisateur.Saisie_nbr_ser("Quel est le nombre de service de soins à créer dans l'hôpital", paramètre_général)

  nombre_services := nombre_ser (paramètre_général)

- la ligne comportant l'opération "Demande_nom_objet_service_soins" par les lignes suivantes:

  Utilisateur.Saisie_nom_ser("Quel est le nom du service de soins à créer", paramètre_général)

  nom_service := nom_ser (paramètre_général)

Nous pouvons faire de même pour les autres classes se trouvant dans le sous-système.

Nom : Utilisateur

Description : Classe servant de lien entre les modules d'interfaces et l'application.

Interface.

Relations :

Composé de : ...
Composant de: /
Généralisation de: ...
Spécialisation de : /
Utilise : ...

Attribut Identifiant: [dans un multi-utilisateurs, il sera nécessaire d'introduire un identifiant pour permettre la distinction entre ceux-ci]

Attributs: paramètre_général

Opérations :

Transforme_mess (code: INT) = message : STRING
* permet la traduction des codes de messages dans la langue choisie dans
* l'interface.

Saisie_nbr_ser(code: INT, paramètre_général)
* permet de demander à l'utilisateur le nombre de services désirés.

Saisie_nom_ser(code: INT, paramètre_général)
* permet de demander à l'utilisateur le nom du service désiré.

Diagramme d'état : DET_Utilisateur [contient deux états : attente_input(de la part de l'utilisateur), attente_output(attend la fin d'un traitement de l'application)]

Implémentation:

...

Nous allons maintenant introduire la base de données qui sera chargée de conserver les données permanentes du système.

10.1.3. La conception du module de la base de données

Le choix de la base de données est important dès la conception. En effet, de son type va dépendre la structure des classes et des relations que l'on va placer dans celle -ci. Dans notre cas, nous avons choisi un SGBD de type relationnel. Nous allons donc avoir des transformations importantes à réaliser pour obtenir la base de données relationnelles.

Les opérations d'accès à la BD vont se réaliser à partir d'une classe Base_Données. En effet, de cette manière, les objets n'interviennent pas directement sur celle-ci, ce qui permet de modulariser l'application. La spécification des accès n'est pas conforme au SGBD choisi.

Ces transformations ont lieu au niveau de la conception. Nous allons nous y attarder maintenant.

Nous allons repartir du modèle objet du sous-système considéré, nous pouvons passer directement du modèle objet à un schéma de la BD. En effet, nous connaissons tous les objets, tous les attributs définis, ainsi que les opérations pouvant nécessiter un accès à la BD.

Nous aurons donc une sorte de Mapping entre, d'une part, les classes et les tables et d'autre part les attributs et les colonnes. Les attributs identifiants sont les Keys, les identifiants secondaires sont un index secondaire posé sur la table. Chaque ligne (row) contiendra les données pour un objet particulier.

Chaque opération de création d'objet conduira également à la mémorisation de celui-ci au sein de la BD, les identifiants des objets correspondant directement aux identifiants dans les tables. De même, les opérations de destruction d'objet conduiront à la destruction des données le concernant dans la BD.

Nous allons donc passer du modèle objet à un schéma conforme avec le modèle relationnel. Pour cela, nous avons un certain nombre de structures à changer. En effet les relations entre les classes et entre les tables ne sont plus permises. Nous allons donc les transformer en suivant les règles fournies au chapitre VII de [HAI86].

Les relations entre les objets sont essentiellement de trois types: spécification, agrégation et utilisation. Ces relations vont donc être éliminées pour répondre au schéma relationnel.

Les relations d'utilisation indiquent un lien entre les classes participantes. Ces relations devront êtres mémorisées et donc reprises dans le schéma de la BD. Tel sera le cas pour la relation existant entre le Lit et le Mouvement. Chaque mouvement va comporter un attribut supplémentaire indiquant le ou les numéro(s) de lit (identifiant sur la classe Lit) participant à cette occurrence de Mouvement (un numéro de lit si le mouvement est de type "entrée" ou de type "sortie" et deux numéros si le mouvement est de type "transfert").

Ces relations forment un passage d'un sous-système à un autre, elle doivent être construites et modélisées dans la base de données en tenant compte à la fois des classes participant à cette relation et des connectivités se trouvant sur celle-ci. Ces connectivités nous indiquent quel identifiant doit être dupliqué dans l'autre classe.

Dans le cas de la relation entre Mouvement et Lit, nous avions 1-2 lits pour 1-N mouvements. Il est donc logique de placer, dans la classe Mouvement un attribut reprenant le numéro du lit d'origine du mouvement (lit_origine) et un attribut de destination du mouvement (lit_destination). Nous ajouterons des contraintes d'intégrité sur ces nouveaux attributs; ils doivent tout deux exister sur les instances de la classe Lit et doivent être différents pour une même instance de mouvement. De plus, si le type de mouvement est "entrée", alors la valeur de l'attribut lit_origine est "NULLE" et si le type de mouvement est "sortie", alors la valeur de l'attribut lit_destination est "NULLE".

En fait, nous avons ici une double relation. En effet, nous avions deux relations d'utilisation modélisant deux associations, c'est pourquoi nous avons la création de deux attributs. Nous ne pouvons les séparer car les contraintes d'intégrité jouent sur les deux attributs. La figure 10-1 en donne une illustration.

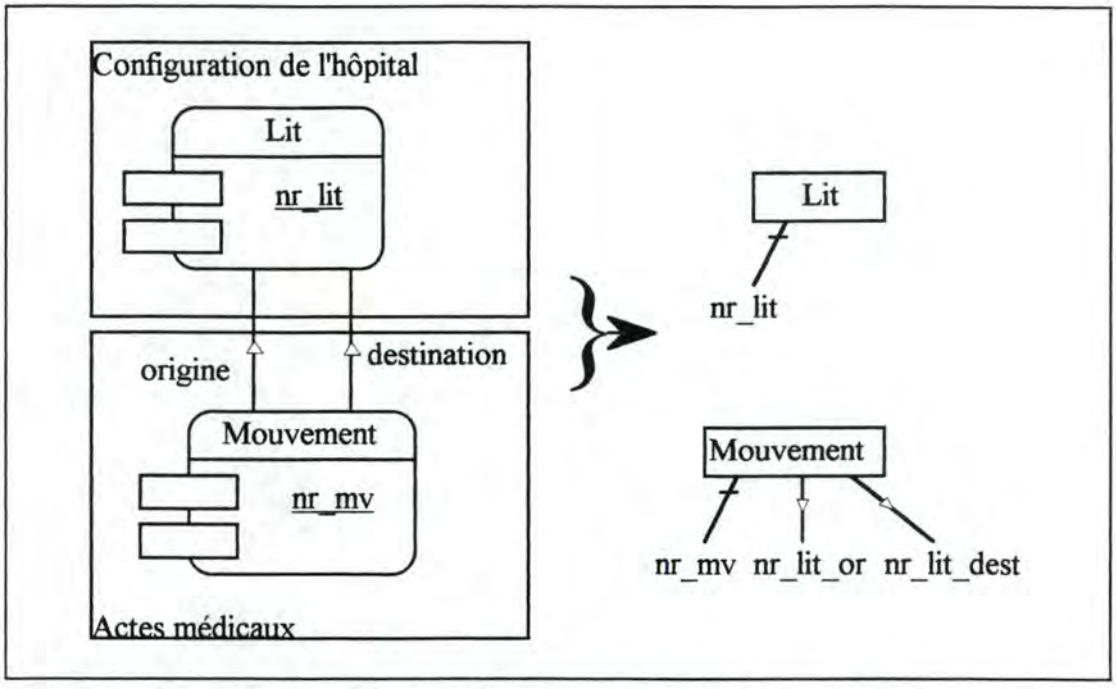


Figure 10-1 : Transformation d'une relation d'utilisation en schéma conforme relationnel.

Les relations de spécification et les relations d'agrégation ne contiennent pas d'attribut, et ce, par définition des relations dans le modèle objet.

Si la relation est de type N-N pour chacune de ces extrémités, nous allons créer un type d'articles représentant cette relation et dont les attributs seront les identifiants des classes participant à la relation. Nous ajouterons deux contraintes d'intégrité pour s'assurer que les valeurs des attributs de ce nouveau type d'articles correspondent au valeurs des attributs définis sur les classes participantes à la relation. Nous obtenons donc un type d'articles dont chaque élément est un couple.

Si la relation est de type 1-N pour au moins l'une de ces extrémités, nous éliminerons cette relation en réalisant une rotation telle que définie dans [HAI86, p.60]. C'est-à-dire que nous placerons l'identifiant de la classe se trouvant à l'extrémité de connectivité 1-N dans la classe se trouvant à l'autre extrémité. De plus, nous ajouterons une contrainte d'intégrité sur ce nouvel attribut. Il doit en effet se retrouver dans la classe d'origine où il est identifiant.

Dans notre exemple, nous avons un relation d'agrégation entre Hôpital et Service_soins. Nous avons donc une relation indiquant que (1-1) Hôpital possède (1-N) Service_soins. Nous allons rendre cette structure conforme: en plaçant, dans la classe se trouvant à l'extrémité possédant la connectivité 1-N, ici Service_soins, l'identifiant de l'autre classe, nom_hop définis sur Hôpital. Nous aurons, de ce fait, un type d'articles Service_soins possédant une colonne supplémentaire qui permet de déterminer l'hôpital dans lequel il est situé.

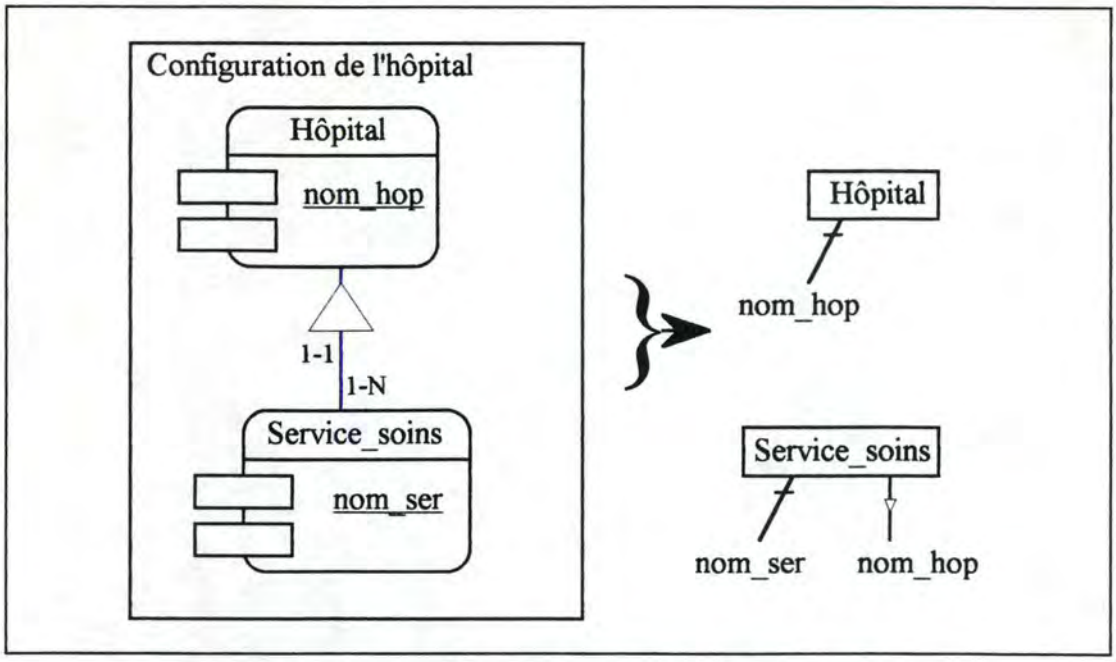


Figure 10-2 : Transformation d'une relation d'agrégation par rotation.

Cette transformation peut être réalisée sur les autres opérations d'agrégation définies dans ce sous-système. La classe Chambre va subir deux transformations, l'une qui permettra d'indiquer le service dans lequel la chambre est située et l'autre la catégorie de chambre à laquelle elle appartient.

Une fois ces dernières modifications faites, nous pouvons passer au sous-système suivant.

10.2. Le sous-système Actes médicaux

Pour ce sous-système, comme pour le suivant, nous allons reprendre la même suite d'étapes.

10.2.1. La conception des classes

En ce qui concerne les différentes classes définies dans ce sous-système, nous allons réaliser le même travail d'explication et de réalisation d'algorithmes pour chacune des opérations définies sur celles-ci.

Nous pourrions refaire ce même travail pour toutes les opérations que nous avons définies, mais cette étape ressemble à celle utilisée généralement par toutes les méthodes de conception de logiciels, qu'elles soient orientées objets ou non. Nous n'avons pas d'aspects spécifiques à montrer. Nous poursuivrons donc en étudiant les nouveaux éléments à insérer suite à l' introduction de l'interface avec l'utilisateur.

10.2.2. La conception du module d'interface

L'ensemble des opérations décrites sur la classe Utilisateur va s'enrichir d'un certain nombre d'opérations en provenance de ce sous-système. Nous aurons un ensemble d'opérations permettant d'afficher les attributs nécessaires aux opérations de

ces classes et un ensemble d'opérations permettant de saisir un certain nombre de valeurs devant servir à une opération précise ou à plusieurs opérations.

En effet, nous pouvons utiliser plusieurs fois la même opération sur les différentes classes. Nous pourrons donc définir sur la classe Utilisateur les opérations d'accès aux valeurs des attributs, les opérations de saisie d'informations de même type que nos attributs, mais également des opérations de saisie de nombres entiers ou de chaînes de caractères permettant d'obtenir des informations intermédiaires. Ces dernières opérations sont relativement génériques et peuvent être très utiles. Le code du message à afficher correspondra à la question particulière demandée.

10.2.3. La conception du module de la base de données

Dans ce sous-système, nous avons quatre classes qui donneront autant de types d'articles. La classe Médecin est en relation d'utilisation de type (0-N, 1-1) avec la classe Mouvement. Nous avons la même relation entre Médecin et Prescription_soins et entre Prestation et Prescription_soins. Nous allons donc supprimer ces relations en plaçant l'identifiant de la classe Médecin dans la classe Mouvement et dans la classe Prescription_soins et l'identifiant de la classe Prestation dans la classe Prescription_soins.

Nous ajouterons, en plus, trois contraintes d'intégrité pour vérifier d'une part que le numéro de médecin se trouvant dans les deux classes est bien repris dans les valeurs d'identifiants des objets de la classe Médecin et d'autre part que les valeurs de code_prest se trouvant dans la classe Prescription_soins se trouvent également dans les valeurs de l'identifiant de la classe Prestation.

Il faut également vérifier les relations d'utilisation existant entre les sous-systèmes. Ici, nous avons deux relations entre la classe Lit et la classe Mouvement, mais nous les avons déjà transformées. Nous avons également une relation entre la classe Prestation et la classe Remboursement_prestation. Cependant, étant donné que la classe Remboursement_prestation a été créée à partir d'une relation existant entre les classes Organisme_assureur et Prestation et que les identifiants de ces deux classes se trouvent déjà dans la classe, nous pouvons éliminer directement ces relations d'utilisation, sans modifier la classe Prestation ou la classe Remboursement_prestation.

Nous obtenons ainsi le schéma de la base de données conforme au relationnel.

10.3. Le sous-système Malade

10.3.1. La conception des classes

En ce qui concerne les différentes classes définies dans ce sous-système, nous réaliserons le même travail d'explication et de réalisation d'algorithmes pour chacune des opérations définies sur celles-ci.

Nous allons repartir de la classe Patient, et pour chacune de ces opérations, donner l'algorithme qui lui correspond:

Nom : Patient

Description : Toute personne hospitalisée ou ayant été hospitalisée dans l'hôpital.

Interface.

<...>

Implémentation:

Création (coord) coord: CP[nom_pat: STRING, pré_pat: STRING, dnais: DATE]

* création d'un objet patient

PRE : ¬ In (Patient (coord))

POST: In (Patient (coord))

```
BEGIN création
    liste_coord_patient_existant := Affiche_id_coord( )
    IF coord NOT IN liste_coord_patient_existant
        THEN  nr_dos := Détermination_nouveau_nr_dos,
              Création_objet_patient(nr_dos)
        ELSE  Erreur_création.
END création.
```

Affiche_id_coord ( ) = liste_pat liste_pat : SET [CP[nom_pat: STRING, pré_pat: STRING, dnais: DATE]]

* retourne une liste contenant les coordonnées des patients.

PRE : /

POST : (coord ∈ liste_pat) ⇔ (In (Patient (coord)))

```
BEGIN Affiche_id_coord
    liste_pat := []
    FOR nom := Patient ( : coord) DO
        Liste_pat := liste_pat + ( Patient : coord)
    END FOR
END Affiche_id_coord
```

Affiche_id ( ) = liste_pat liste_pat : SET [nr_dos : INT]

* retourne une liste contenant les numéros de dossier des patients.

PRE : / num_pat : INT

POST : (num_pat ∈ liste_pat) ⇔ (In (Patient (num_pat)))

```
    BEGIN Affiche_id
        liste_pat := []
        FOR ALL nr_dos In(Patient : nr_dos)) DO
            Liste_pat := liste_pat + (Patient : nr_dos)
        END FOR
    END Affiche_id

Destruction (num_pat)                                    num_pat : INT
    * destruction d'un objet patient dont le nr_dos est donné
    PRE :    In (Patient (num_pat))
    POST:    ¬ In (Patient (num_pat))

    BEGIN destruction
        liste_mv := Détermine_liste_tous_mv_de_pat(nr_dos)
        REPEAT
            {liste_mv}' := {liste_mv}U{nr_mv}
            Destruction_lien_patient_mv(nr_dos, nr_mv)
            Mouvement.Destruction(nr_mv),
            liste_mv := liste_mv'
        UNTIL EMPTY liste_mv
        liste_pr := Détermine_liste_toutes_pr_de_pat(nr_dos)  [liste de
                        toutes les prescriptions du patient]
        REPEAT
            {liste_pr}' := {liste_pr}U{nr_pr}
            Destruction_lien_patient_pr(nr_dos, nr_pr)
            Prescription_soins.Destruction(nr_pr),
            liste_pr := liste_pr'
        UNTIL EMPTY liste_pr
        Destruction_objet_patient_dont_identifiant_est (nr_pat),
    END destruction


Détermination_nr_dos (coord) = nr_dos
    * opération retournant le numéro de dossier pour un patient dont on
    * donne les coordonnées, s'il existe.
    PRE : /
    POST:    IF In (Patient (coord))
             THEN nr_dos = Nr_dos(Patient(Coord))
             ELSE ¬ In (Patient (nr_dos))

    BEGIN détermination_nr_dos
        Détermination_nr_dos := nr_dos(Patient : coord)),
    END détermination_nr_dos.
```

```
Détermination_nouveau_nr_dos (coord) = nr_dos
    * opération retournant le nouveau numéro de dossier pour un nouveau
    * patient dont on donne les coordonnées.
    PRE :     /
    POST:     IF¬ In (Patient(coord)
              THEN ¬ In (Patient (nr_dos))

    BEGIN détermination_nouveau_nr_dos
        liste_nr_dos := Patient.Affiche_id,
        max_nr_dos := MAX{liste_nr_dos},
        Détermination_nouveau_nr_dos := max_nr_dos +1
    END détermination_nouveau_nr_dos.

Adresse_patient (nr_dos) = adresse                    nr_dos: INT
                                                      adresse : STRING
    * retourne l'adresse du patient possédant le numéro de dossier concerné
    PRE :     In (Patient (nr_dos))
    POST :    adresse = Adr_pat(Patient (nr_dos))

    BEGIN Adresse_patient
        Adresse_patient := Adr_pat(Patient : nr_dos))
    END Adresse_patient

Mod_adresse_patient (nr_dos, adresse)                 nr_dos : INT
                                                      adresse : STRING
    * modifie l'adresse du patient possédant le numéro de dossier concerné
    PRE :     In (Patient (nr_dos))
    POST :    Adr_pat(Patient (nr_dos)) = adresse

    BEGIN Mod_adresse_patient
        Adr_pat(Patient : nr_dos)) := Adresse_patient
    END Adresse_patient
```

Nous pourrions refaire ce même travail pour les autres opérations que l'on vient de définir et pour les autres classes. Mais la même remarque que pour le sous-système précédent peut être émise, nous ne donnerons donc pas toutes les opérations.

Cependant, nous reviendrons plus tard sur les opérations liées à l'opération Admission. Nous devons en effet, avoir une image précise de la base de données. Les interactions avec la classe Utilisateur seront progressivement augmentées des opérations nécessaires pour mener à bien ces opérations.

10.3.2. La conception du module d'interface

Les opérations ajoutées par ce sous-système sont du même type que celles présentées lors des sous-systèmes précédents, nous ne les donnerons pas. Il convient, dans un problème réel, d'essayer de réutiliser au maximum les opérations définies précédemment sur cette classe Utilisateur en rendant ces opérations les plus

génériques possibles. Par exemple, nous pourrions définir une opération qui permet de saisir un nombre entier et l'utiliser pour saisir le numéro de chambre, le numéro de dossier, le numéro de lit, le numéro de médecin, etc. Les procédures de validation des données saisies se réaliseront sur les classes elles-mêmes, ce qui facilitera le travail. En effet, toutes les informations quant aux données valables se trouvent sur ces classes et non sur la classe Utilisateur.

10.3.3. La conception du module de la base de données

Nous avons essentiellement des relations d'utilisation entre les classes de ce sous-système. Nous pouvons donc les éliminer selon la même technique que celle présentée dans les autres sous-systèmes. Cela va se réaliser relativement vite.

Notre attention se portera sur les relations existant entre les sous-systèmes, c'est-à-dire ici, entre la classe Patient et la classe Mouvement, entre la classe Patient et la classe Prescription_soins, entre la classe Organisme_assureur et la classe Remboursement_prestation et entre la classe Organisme_assureur et la classe Remboursement_chambre.

La relation entre la classe Patient et la classe Mouvement est de type (1-N, 1-1), nous avons donc l'identifiant de la classe Patient qui sera placé dans la classe Mouvement et la création d'une contrainte d'intégrité vérifiant les valeurs de cet attribut ajouté. Nous avons exactement la même transformation entre la classe Patient et la classe Prescription _soins.

Nous avons déjà parlé de la transformation de la relation entre la classe Organisme_assureur et la classe Remboursement_prestation. Entre la classe Organisme_assureur et la classe Remboursement_chambre, nous aurons une transformation similaire.

Sont ainsi terminées les éliminations des relations du modèle objet. Nous possédons maintenant une base de données conforme au modèle relationnel.

Nous allons maintenant revenir sur l'opération Admission définie sur la classe Patient.

10.4. Conception des fonctionnalités

Cette opération est demandée par un utilisateur désirant faire admettre une personne dans l'hôpital. Nous devons définir une opération sur la classe Utilisateur qui va appeler les opérations nécessaires sur la classe Patient.

```
Opération Demande_Admission
    *Suite à la sélection de l'item "Admission" dans le menu "Mouvement"
    * de l'interface, la classe Utilisateur va demander l'exécution de
    * l'opération Admission définie sur la classe Patient.
    PRE:      Item_selected (Item) = "Admission"
    POST:     Admission_réalisée ∨ Opération_Annulée
```

```
BEGIN Demande_Admission
    Saisie_coord ("introduire les coordonnées du patient à
                admettre", nouv_coord),
    Patient.Patient_connu (nouv_coord, par_contrôle),
    IF par_contrôle = TRUE
        THEN  Patient.Adm_patient_connu (nouv_coord,
                                    statut_opération),
        ELSE  Patient.Création (nouv_coord)
              Patient.Adm_Patient_non_connu (nouv_coord,
                                    statut_opération)
    END IF
    IF Statut_opération = "OK"
        THEN  Sauvegarde_information,
              * permet de faire un 'commit' sur la BD pour
              *  ajouter réellement les données nouvellement
              *  introduites
              Admission_réalisée := TRUE
        ELSE  Annuler_information,
              * permet de faire un 'Rollback' sur la BD pour
              *  annuler les modifications qui y ont été apportées.
              Affiche_erreur_dans_admission,
              Opération_Annulée := TRUE
    END IF
END Demande_Admission
```

Les deux valeurs booléennes permettent de déterminer si l'opération s'est bien déroulée ou si il y a eu annulation de l'opération. Un paramètre *statut* sera ajouté au paramètre_général pour permettre un certain contrôle entre les différentes opérations. De plus, les deux opérations Sauvegarde_information et Annule_information permet de sauvegarder ou d'annuler les conséquences de cette opération sur la base de données. De cette manière, toute opération contiendra ces deux opérations permettant de faire ou de défaire les opérations intermédiaires réalisées sur la BD, dans le courant d'une opération plus globale.

Sur la classe Patient, nous avons un certain nombre de nouvelles opérations: Patient_connu, Adm_patient_connu et Adm_patient_non_connu. L'ancienne opération Admission va être remplacée par d'autres opérations. En effet, cette opération est maintenant située sur la classe Utilisateur qui va jouer le rôle de coordinateur. C'est donc cette classe qui va demander l'exécution des différentes fonctionnalités attendues du système, celles-ci étant demandées de l'extérieur de celui-ci. Nous allons donner ici les différentes opérations définies. De mêmes que celles qui y sont liées au sein de l'enchaînement d'opérations réalisant la fonctionnalité.

Dans la classe Patient:

```
Patient_connu (coord, contrôle)
    * le paramètre contrôle indique si le patient dont on donne les
    * coordonnées est connu.
    BEGIN Patient_connu
        liste_patients_existant := Affiche_id_coord ( )
        IF coord IN liste_coord_patient_existant
            THEN   contrôle := TRUE
            ELSE   contrôle := FALSE
        END IF
    END Patient_connu

Adm_patient_connu (coord, contrôle)
    *permet l'admission d'un patient qui a déjà été hospitalisé.
    BEGIN Adm_patient_connu
        nr_dos := Détermination_nr_dos (coord),
        Malade_actuel (nr_dos, contrôle)
            * permet de dire si le patient dont on donne le numéro de
            *  dossier est actuellement hospitalisé.
        IF contrôle = TRUE
            THEN   Utilisateur.Erreur ("Le patient est déjà
                       hospitalisé dans l'hôpital"),
                   Abandon_admission.
            ELSE Adm_anc_malade (nr_dos, contrôle)
                       * permet de réaliser l'admission d'un patient qui a
                       *  déjà été hospitalisé
        END IF
    END Adm_patient_connu

Adm_patient_non_connu (coord, contrôle)
    * permet l'admission d'un patient qui n'a jamais été hospitalisé.
    BEGIN Adm_patient_non_connu
        nr_dos := Détermination_nouveau_nr_dos (coord)
        Init_adm_paramètre_général,
            *  permet de donner les valeurs par défaut aux différents
            *   éléments nécessaires à l'admission dans le
            *   paramètre_général.
        Utilisateur.Saisie_info_malade ("Introduire les informations
            concernant le patient", paramètre_général)
            * permet de saisir les informations complémentaires liées au
            *  patient: son adresse, sexe, état_civ, nr_tit, organisme
            *  assureur, etc. en utilisant le paramètre_général comme
            *  valeurs initiales.
        Mise_à_jour_attributs_patient (nr_dos, paramètre_général),
            * permet la mise à jour des données que l'on vient de saisir.
```

```
        Utilisateur.Saisie_info_hop ("Introduire les informations
              concernant l'hôpital lors de l'admission", paramètre_général)
              * permet de saisir les informations concernant l'hôpital du fait
              *  de l'admission, c'est-à-dire le service, la catégorie de
              *  chambre, le médecin.
        Service.Allocation_lit (nom_ser, nom_cat, nr_lit, contrôle)
              * permet d'allouer un lit dans le service donné et dans une
              *  chambre de la catégorie donnée.
        IF contrôle = FALSE
              THEN Abandon_Admission.
              ELSE
                 Mouvement.Création (nr_dos, "entrée", *, nr_lit, nr_med,
                    date_jour, contrôle)
                    * crée un mouvement dont le patient qui le subit est
                    *  nr_dos, de type est "entrée", dont le lit d'origine
                    *  n'est pas donné, dont le lit de destination est nr_lit,
                    *  dont le médecin qui l'a prescrit est nr_med et dont la
                    *  date est date_jour. Le paramètre contrôle va dire si
                    *  l'opération s'est bien déroulée.
                 IF contrôle = FALSE
                    THEN Abandon_admission
                 END IF
        END IF
END Adm_patient_non_connu.
```

Nous avons introduit un certain nombre d'opérations dans diverses classes. Nous avons ainsi : Patient.Malade_actuel (nr_dos, contrôle), Patient.Admission_ abandon, Patient.Adm_anc_malade (nr_dos, contrôle), Patient.Mise_à_jours_attributs _patient(nr_dos, paramètre_général), Service.Allocation_lit (nom_ser, nom_cat, nr_lit, contrôle) et Mouvement.Création (nr_dos, type_mv, nr_lit_or, nr_lit_dest, nr_med, date_mv, contrôle).

Nous pouvons remarquer que la seule différence entre l'admission d'un ancien malade et celle d'un patient n'ayant jamais été hospitalisé est l'utilisation du numéro de dossier créé lors de la première admission et les valeurs des informations du patient comme valeurs par défaut pour les saisies des informations supplémentaires concernant le patient. Ceci permet de ne pas devoir réintroduire les données restées correctes.

Nous définissons maintenant les nouvelles opérations identifiées ci-dessus en commançant par celles de la classe Patient.

```
Malade_actuel (nr_dos, contrôle)
    * permet de dire si le patient dont on donne le numéro de
    *  dossier est actuellement hospitalisé.
    BEGIN Malade_actuel
        Mouvement.Dernier_mouvement (nr_dos, nr_mv, contrôle)
        Mouvement.Type_du_mouvement (nr_mv, type_mouv, contrôle)
        IF type_mouv = "Sortie"
            THEN contrôle := FALSE
            ELSE contrôle := TRUE
    END Malade_actuel

Admission_abandon
    * permet d'annuler ce que l'on vient de réaliser, c'est-à-dire de ne pas
    *  mémoriser l'admission. Une fois exécutée, nous retournons à
    *  l'opération initiatrice générale
    BEGIN Admission_abandon
        contrôle := FALSE
    END Admission_abandon

Adm_anc_malade (nr_dos, contrôle)
    * permet de réaliser l'admission d'un patient qui a
    * déjà été hospitalisé
    BEGIN Adm_anc_malade
        Recherche_info_patient (nr_dos, paramètre_général)
        Utilisateur.Saisie_info_malade ("Introduire les informations
            concernant le patient", paramètre_général)
            * permet de saisir les informations complémentaires liées au
            *  patient: son adresse, sexe, etat_civ, nr_tit, organisme
            *  assureur, etc. en utilisant le paramètre_général pour les
            *  valeurs par défaut.
        Mise_à_jour_attributs_patient (nr_dos, paramètre_général),
            * permet la mise à jour des données que l'on vient de saisir.
        Utilisateur.Saisie_info_hop ("Introduire les informations
                concernant l'hôpital lors de l'admission",
                paramètre_général)
            * permet de saisir les informations concernant l'hôpital du fait
            *  de l'admission, c'est-à-dire le service, la catégorie de
            *  chambre, le médecin
        Validation_info_adm (paramètre_général, contrôle)
            * permet de vérifier les domaines de valeurs de certaines
            *  valeurs introduites : pour le service, le médecin, et la
            *  catégorie
```

```
        IF  contrôle = FALSE
              THEN Abandon_Admission.
              ELSE
                 Service.Allocation_lit (nom_ser, nom_cat, nr_lit,
                       contrôle)
                    * permet d'attribuer un lit dans le service donné et dans
                    *  une chambre de la catégorie donnée.
                 IF contrôle = FALSE
                    THEN Abandon_Admission.
                    ELSE
                       Mouvement.Création (nr_dos, "entrée", *, nr_lit,
                          nr_med, date_jour, contrôle)
                       * crée un mouvement dont le patient qui le subit est
                       *  nr_dos, de type est "entrée", dont le lit d'origine
                       *  n'est pas donné, dont le lit de destination est
                       *  nr_lit, dont le médecin qui l'a prescrit est nr_med
                       *  et dont la date est date_jour. Le paramètre
                       *  contrôle va dire si l'opération s'est bien déroulée.
                       IF contrôle = FALSE
                          THEN Abandon_admission
                       END IF
                 END IF
        END IF
    END Adm_anc_patient.

Mise_à_jours_attributs _patient(nr_dos, paramètre_général)
    * permet la mise à jour des données que l'on vient de saisir.
    BEGIN Mise_à_jours_attributs _patient
          Modifie_adresse_pat (nr_dos, adr_pat),
          Modifie_telephone_patient (nr_dos, phone),
          Modifie_sexe_patient (nr_dos, sexe),
          Modifie_etat_civil_patient (nr_dos, etat_civ),
          Modifie_num_titulaire_patient (nr_dos, nr_tit),
          Modifie_type_affiliation_patient (nr_dos, type_aff)
    END Mise_à_jours_attributs _patient.
```

Dans la classe Service :

```
Allocation_lit (nom_ser, nom_cat, nr_lit, contrôle)
    * permet d'allouer un lit dans le service donné et dans une chambre de la
    *      catégorie donnée.
    BEGIN Allocation_lit
          Service_plein (nom_ser, contrôle)
          IF contrôle = TRUE
                THEN contrôle = FALSE
                ELSE
                   IF nom_cat = "ch_part"
                   THEN
                      Détermine_lit (nom_ser, nom_cat, nr_lit, contrôle)
                      IF contrôle = FALSE
                      THEN
                         IF nom_cat = "ch_2_lit"
                         THEN
                            Détermine_lit (nom_ser, nom_cat, nr_lit,
                                                        contrôle)
                            IF contrôle = FALSE
                            THEN
                               IF nom_cat = "ch_4_lit"
                               THEN
                                  Détermine_lit (nom_ser, nom_cat, nr_lit,
                                                        contrôle)
                               END IF
                            END IF
                         END IF
                      END IF
                   END IF
                   IF nom_cat = "ch_2_lit"
                   THEN
                      Détermine_lit (nom_ser, nom_cat, nr_lit, contrôle)
                      IF contrôle = FALSE
                      THEN
                         IF nom_cat = "ch_part"
                         THEN
                            Détermine_lit (nom_ser, nom_cat, nr_lit,
                                                        contrôle)
                            IF contrôle = FALSE
                            THEN
                               IF nom_cat = "ch_4_lit"
                               THEN
                                  Détermine_lit (nom_ser, nom_cat, nr_lit,
                                                        contrôle)
```

```
                        END IF
                     END IF
                  END IF
               END IF
            END IF
            IF nom_cat = "ch_4_lit"
            THEN
               Détermine_lit (nom_ser, nom_cat, nr_lit, contrôle)
               IF contrôle = FALSE
               THEN
                  IF nom_cat = "ch_2_lit"
                  THEN
                     Détermine_lit (nom_ser, nom_cat, nr_lit,
                                                   contrôle)
                     IF contrôle = FALSE
                     THEN
                        IF nom_cat = "ch_part"
                        THEN
                           Détermine_lit (nom_ser, nom_cat, nr_lit,
                                                   contrôle)
                        END IF
                     END IF
                  END IF
               END IF
            END IF

Détermine_lit (nom_ser, nom_cat, nr_lit, contrôle)
    * permet de déterminer un lit de libre dans une chambre du service donné
    * et de la catégorie donnée. Cette opération est nécessaire pour pouvoir
    * changer de catégorie si les lits de la catégorie voulue sont tous
    * occupés.
    BEGIN Détermine_lit
        Chambre.Détermine_chambre (nom_ser, nom_cat, liste_nr_ch)
        FOR EACH nr_ch IN liste_nr_ch DO
            Lit.Détermine_lit_dans_chambre( nr_ch, liste_lit)
            FOR EACH nr_lit IN liste_lit DO
               Lit.Lit_libre (nr_lit, contrôle)
               IF contrôle = FALSE
                  THEN NEXT
               END IF
            END FOR
            IF contrôle = FALSE
               THEN NEXT
            END IF
        END FOR
    END Détermine_lit
```

```
Service_plein (ser),
    * Teste la non existence d'un lit libre dans le service considéré.
    BEGIN Service_plein
          nom_cat := "ch_4_lit"
          Détermine_lit (nom_ser, nom_cat, nr_lit, contrôle)
          IF contrôle = FALSE
                THEN
                   nom_cat := "ch_2_lit"
                   Détermine_lit (nom_ser, nom_cat, nr_lit, contrôle)
                   IF contrôle = FALSE
                      THEN
                         nom_cat := "ch_part"
                         Détermine_lit (nom_ser, nom_cat, nr_lit, contrôle)
                   END IF
          END IF
          IF contrôle = FALSE
                THEN contrôle := TRUE
                ELSE contrôle := FALSE
          END IF
    END Service_plein
```

Dans la classe Mouvement, l'opération de création sera revue afin de respecter la syntaxe utilisée ici.

```
Création (num_dos, type_mouv, num_lit_or, num_lit_dest, num_med,
                        date_mouv, contrôle).
    BEGIN création
          nr_mouv = Max (mouvement (: nr_mv ) ) + 1,
          Création_objet_mouvement(nr_mouv)
          Type_mv(Mouvement (:nr_mv = nr_mouv)) := type_mouv
          nr_dos_pat(Mouvement (:nr_mv = nr_mouv)) := num_dos
          nr_lit_or(Mouvement (:nr_mv = nr_mouv)) := num_lit_or
          nr_lit_dest(Mouvement (:nr_mv = nr_mouv)) := num_lit_dest
          nr_med_pr(Mouvement (:nr_mv = nr_mouv)) := num_med
          date_mv(Mouvement (:nr_mv = nr_mouv)) := date_mouv
    END création.
```

Dans la classe Chambre, nous avons l'opération Détermine_chambre qui permet de donner la liste des chambres appartenant à un service et à une catégorie donnés.

```
Détermine_chambre (nom_ser, nom_cat, liste_ch)
    * permet de donner la liste des chambres appartenant à un certain service
    *   et à une certaine catégorie.
    BEGIN Détermine_chambre
        Liste_ch := nr_ch (Chambre ( : nom_ser_ch = nom_ser
                                    AND : nom_cat_ch = nom_cat))
    END  Détermine_chambre
```

Dans la classe Lit, nous avons les opérations Détermine_lit_dans_chambre et Lit_libre. Nous allons les détailler maintenant.

```
Détermine_lit_dans_chambre (num_ch, liste_lit)
    * permet de donner la liste des lits appartenant à une certaine chambre.
    BEGIN Détermine_lit_dans_chambre
        Liste_lit := nr_lit (Lit ( : nr_ch_lit = num_ch))
    END Détermine_lit_dans_chambre

Lit_libre (nr_lit, contrôle)
    * Teste si le lit considéré est libre
    BEGIN Lit_libre
        Mouvement.Date_dernier_mv_or (nr_lit, date_or)
        Mouvement.Date_dernier_mv_de (nr_lit, date_de)
        IF date_or > date_de
            THEN contrôle := TRUE
                * le lit a été l'origine d'un mouvement plus récent que le
                *   dernier mouvement l'ayant pour destination
            ELSE contrôle := FALSE
    END  Lit_libre
```

```
Date_dernier_mv_de (nr_lit, date_de)
    * permet de donner la date du dernier mouvement ayant pour destination
    *   le lit considéré.
    BEGIN Date_dernier_mv_de
          Liste_mv_de := nr_mv (Mouvement ( : nr_lit_de = nr_lit)
          Max_mv_de := MAX ( Liste_mv_de)
          date_de := Date_mv (Mouvement(: nr_mv = Max_mv_de))
    END Date_dernier_mv_de
```

Nous en avons terminé avec l'opération Demande_admission. Pour les autres fonctionnalités définies sur l'application, nous repartirons de la classe Utilisateur ou d'une autre classe suivant celle qui est à l'origine du déclenchement. Nous essayerons de réaliser des opérations relativement larges pour pouvoir les utiliser dans plus d'un enchaînement d'opérations.

Conclusions

Nous venons de voir, dans cette étude de cas, une illustration de ce que peut être l'application de la méthode que nous avons présentée dans les chapitres précédents. Nous avons tenu à redonner les justifications de certaines transformations ou éliminations afin qu'un lecteur n'ayant pas lu les chapitres théoriques puisse comprendre celles-ci.

En résumé, nous pouvons remarquer que l'analyse se rapproche fort de l'analyse utilisant le modèle Entité/Association. Nous rappelons le Mapping qui peut être fait entre ce modèle et le modèle Objet proposé. Ce Mapping permet de faciliter le passage de l'analyse traditionnelle à l'analyse orientée objet utilisant le modèle proposé. Ce passage n'est pas à sous-estimer. En effet, les analystes n'acceptent qu'avec résignation une méthode qui leur demande de mettre de côté une partie de l'expérience acquise au cours de longues années de pratique.

La méthode proposée n'est pas purement orientée objet du début à la fin, les outils terminaux n'étant pas encore tous disponibles. Les bases de données orientées objets n'existent qu'à titre expérimental, les langages de programmation deviennent progressivement opérationnels et les interfaces commencent seulement à apparaître.

Cette méthode débute donc, comme nous venons de le constater, par un modèle objet modélisant une application de gestion de système d'information, en l'occurrence une gestion d'hôpital. Un certain nombre de notions ont été définies afin de rendre compte des divers éléments du monde réel. Nous avons pu constater que ce sont les relations qui sont les plus difficiles à modéliser, tandis que les opérations semblent fort compliquées à définir et à identifier.

Nous avons suivi l'énoncé du problème et essayé de décomposer les opérations en d'autres plus simples sur les objets auxquels elles se rapportent. Nous en avons donc un nombre important réparties sur tout le schéma. Par souci de clarté, nous pouvons cacher les opérations de base, mais nous devons veiller à les retrouver dans les définitions textuelles afin de ne pas les oublier par la suite.

Nous sommes passés à la phase de conception de manière relativement douce. La décision de franchir cette étape doit se faire lorsque les classes identifiées sur le domaine du problème sont en relation entre elles, que tous leurs attributs modélisent toute leur mémoire et que leurs opérations réalisent tout ce que l'on attend d'elles. Cette décision peut apparaître un peu trop tôt, mais un retour en arrière est toujours possible.

Cette phase de conception est essentiellement un processus de raffinement des différentes classes. Nous allons donner un algorithme pour chacune des opérations définies sur ces classes. De nouvelles opérations peuvent apparaître suite à la décomposition des premières. Celle-ci permet de faciliter la réutilisation d'opérations relativement simples, telles que des recherches de valeurs d'attributs selon certains critères.

De plus, nous ajoutons l'interface avec l'utilisateur. Pour cela nous introduirons une ou plusieurs nouvelles classes selon les différents moyens d'interaction existant sur l'application. Ces classes seront définies de la même manière que les autres dans un sous-système propre à chacune et modélisant les différents éléments composant cette classe. Nous pensons aux différents éléments composant une fenêtre par exemple.

Ces classes jouent souvent un rôle important de coordination et de déclenchement des enchaînements d'opérations représentant les fonctionnalités attendues du système. En effet, celles-ci sont le plus souvent une réponse à une sollicitation de la part de l'utilisateur. D'autres sont déclenchées par des plannings. Dans ce cas, nous aurons une classe Agenda qui permettra le déclenchement en temps opportun de ces enchaînements.

Cette phase de conception introduit également la base de données. En fonction de son type et de ses caractéristiques, nous transformerons le schéma objet issu de la phase d'analyse en schéma conforme aux règles du type de SGBD choisi. Pour cela, nous aurons un certain nombre de transformations qui se rapprochent fort de celles utilisées lors du passage d'un schéma Entité/Association en un schéma des accès possibles. Nous en avons une illustration dans le cas étudié.

Cette introduction de l'interface et de la base de données permet de définir précisément par des algorithmes simples les différentes opérations définies sur les classes. Nous pouvons généralement donner les algorithmes des opérations de base dès le début, mais il est préférable d'attendre l'introduction de l'interface et surtout de la base de données pour les opérations réalisant les fonctionnalités de l'application et faisant intervenir plusieurs classes.

La suite de cette phase de conception est :

- l'implémentation des classes dans un langage de programmation orienté objets ou pouvant simuler l'orienté objets,
- l'implémentation et la réalisation du sous-système d'interface proprement dites à l'aide d'un gestionnaire,
- l'implémentation et la réalisation de la base de données à l'aide du SGBD choisi.

Nous sommes donc partis de l'analyse orientée objets, nous avons poursuivi par la conception orientée objets et nous terminons par une implémentation qui peut être orientée objets, mais pas nécessairement.

CONCLUSIONS GÉNÉRALES

"[...] *in the physical or abstract reality being modeled, the objects are just there for the picking!"* [MEY88, p.51]

Cette petite phrase de Bertrand Meyer montre que, même dans le monde de la conception d'applications, les avis divergent. Ce ne sont pas les auteurs des différentes méthodes présentées dans ce travail qui nous contrediront. L'examen de la démarche proposée lors de la phase d'analyse nous montre l'importance et la difficulté de cette identification d'objets.

L'historique nous a montré l'évolution des méthodes. Ensuite, une approche théorique a fait apparaître les notions que recouvraient les mots magiques de ces dernières années : l'Orienté Objets. A suivi une présentation de six méthodes proposées par des auteurs connus dans le monde des méthodes de conception structurées.

Une analyse critique a montré à la fois la diversité et le tronc commun de ces différentes méthodes. C'est ainsi que nous avons pu déterminer trois groupes de méthodes : les minimalistes, Shlaer/Mellor et Wirfs-Brock, les maximalistes, Booch et Colbert, tandis qu'entre ces deux extrèmes, nous trouvons les méthodes de Coad/Yourdon et la méthode HOOD.

Cependant toutes ces méthodes relèvent essentiellement de la théorie. En effet, les outils nécessaires à la finalisation de la conception et de l'implémentation ne sont pas encore disponibles actuellement.

La proposition de méthode se veut opérationnelle. Elle permet dès lors de quitter l'approche Orientée Objets pure des théoriciens pour fléchir durant la phase de conception vers les outils actuels que l'on utilisera durant la phase d'implémentation. Ces outils sont ou ne sont pas orientés objets.

A la fois simple et facile à utiliser, notre méthode permet de rendre compte et de modéliser le domaine du problème de façon correcte, complète et cohérente, comme le montre son application à l'exemple de la gestion d'hopital.

Cet exemple n'a pas été choisi au hasard. En effet, chaque méthode a plus ou moins un domaine d'application privilégié. Notre méthode s'adresse aux Systèmes d'Information.

L'apparition de nouvelles applications orientées objets dans le monde des langages de programmation, des interfaces homme-machine et des bases de données nous permettra de reconsidérer chaque méthode présentée ici dans un environnement réel de développement.

L'avenir de ces méthodes n'est pas la mise au point d'une méthode universelle s'appliquant à tous les problèmes, mais plutôt la constitution d'un ensemble de méthodes souples et adaptées à chaque type de problème pouvant être rencontré.

L'utilisation croissante des outils CASE permettra de constituer, au sein d'un même outil, un ensemble cohérent de méthodes permettant ainsi au concepteur de choisir celle qui lui semble la mieux adaptée au problème auquel il est confronté.

L'expérience acquise lors du stage réalisé au Centre Européen de Customisation d'outils CASE de Digital Equipment Corporation, situé à Ferney-Voltaire (F), durant les cinq premiers mois de l'année académique 91-92, semble confirmer cette tendance.

Gageons qu'à l'avenir, un outil de conception performant devra présenter, en son sein, un large éventail de méthodes plutôt qu'une exclusivité rigide pour faire face aux différents problèmes rencontrés.

BIBLIOGRAPHIE

[BOD89] BODART F., PIGNEUR Y., *Conception Assistée des Systèmes d'Information*, Masson, Paris, 2de édition, 1989.

[BOO86] BOOCH G, *Object Oriented Developpement*, IEEE Transactions on Software Engineering Vol SE-12, February 86.

[BOO91] BOOCH G, *Object Oriented Design with applications*, Benjamin/Cummings Publishing Company, Inc., California, 1991.

[COA91] COAD P., YOURDON E., *Object-Oriented Design*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1991.

[COA92] COAD P., YOURDON E., *Analyse Orientée Objets,* Prentice Hall Inc., London et Masson S.A., Paris, 1992.

[COL89] COLBERT E., *The Object-Oriented Software Development Method: A Practical Approach to Object-Oriented Development*, in TRI-Ada Proceedings, ACM, Octobre 1989.

[DUB90] DUBOIS E., *Cours de Méthode de développement de logiciel*, cours de seconde licence et maîtrise en informatique, FUNDP, Namur, 1990-1991.

[GAL85] GALINIER M., MATHIS A., *Guide du concepteur MARCH*, Thomson CSF, DSE et IGL Technology, 1985.

[HAI86] HAINAUT J.-L., *Conception Assistée des Applications Informatiques 2. Conception de la base de données*, Masson, Paris, 1986.

[HEN90] HENDERSON-SELLERS B., EDWARDS J. M., Object Oriented Systems Life Cycle, in *Communications of the ACM*, ACM, Septembre 1990, Vol. 33, N° 9, pp 142-159.

[HOO91] HOOD, *HOOD Reference Manual*, CISI Ingenierie, CRI A/S et MATRA Espace, 1991.

[KOR90] KORSON T., McGREGOR J.D., Understanding Object-Oriented: A Unifying Paradigm, in *Communications of the ACM*, ACM, Septembre 1990, Vol. 33, N° 9, pp 40-60.

[MEY88] MEYER B., *Object Oriented Software Construction*, Prentice Hall, Inc., Hemel Hempstead, 1988.

[ROL90] ROLLAND C., FLORY A., *La conception des systèmes d'information: état de l'art et nouvelles perspectives.*, Communication pour le 20ème Anniversaire des MIAGE, INFORSID'90, 1990.

[ROL91] ROLLAND C., CAUVET C., *Modélisation conceptuelle orientée objet*, INRIA, BD3, Lyon, 1991.

[SHL88] SHLAER S., MELLOR S.J., *Object-Oriented Systems Analysis, Modeling the World in Data.*, YourdonPress, Prentice Hall, Inc., Englewood Cliffs, NJ, 1988.

[WIR90] WIRFS-BROCK R., WILKERSON B. et WIENER L., *Designing Object-Oriented Software,* Prentice Hall, Inc., Englewood Cliffs, NJ, 1990.

ANNEXES

ANNEXE 1 :

PRÉSENTATION COMPLÈTE

DES MÉTHODES

Nous allons passer en revue, de façon plus complète et plus détaillée, les différentes méthodes présentées dans le chapitre 3. Nous aurons de ce fait, dans l'ordre, les méthodes de BOOCH, de COAD/YOURDON, de COLBERT, HOOD, de SHLAER/MELLOR et de WIRFS-BROCK.

Voici le plan de cette annexe:

A1.1. La méthode de BOOCH

Cette méthode, créée par Grady **BOOCH**, est générale et fait référence à de nombreux auteurs. La bibliographie de cet ouvrage est donc très intéressante pour les personnes désireuses d'étudier les différents auteurs de l'approche orientée objets.[BOO91]

1.1. Les concepts

1.1.1. La complexité du système analysé

Tout système destiné a être modélisé est complexe. Cette complexité se situe généralement au-delà de la capacité intellectuelle du concepteur. La tâche de l'équipe de développement est de créer une illusion de simplicité. Il convient donc de décomposer ce système en éléments plus simples et manipulables. Cette complexité peut donc généralement être représentée sous forme d'une hiérarchie d'éléments.

La conception orientée objets présentée par Booch est une méthode qui nous conduit à une décomposition orientée objets. Cette conception définit une notation et un processus de construction de système complexe et offre un ensemble riche de modèles logiques et physiques à partir desquels la modélisation des différents aspects du système étudié sera réalisée. Elle permet de créer des programmes qui soient meilleurs tant au point de vue des améliorations futures, que de celui de la correction et de la maintenance.

Pour réaliser cette décomposition, le modèle objet est utilisé.

1.1.2. Le modèle objet

Le modèle objet fournit un canevas conceptuel pour la méthode. Il met en oeuvre un certain nombre de principes:

- *le principe d'abstraction.* L'abstraction dénote les caractéristiques essentielles d'un objet qui le distingue de tous les autres. Elle fournit donc une délimitation des frontières conceptuelles relatives à la perspective choisie par le concepteur et élimine les détails qui ne sont pas pertinents pour le lecteur. L'abstraction se focalise sur l'aspect extérieur de l'objet et de ce fait, elle permet la séparation entre le comportement d'un objet et son implémentation.

- *Le principe d'encapsulation.* L'encapsulation est un processus qui permet de masquer tous les détails non essentiels d'un objet. Elle cache à la fois la représentation de l'objet et l'implémentation de ces méthodes.
- *Le principe de modularité.* La modularité est la propriété d'un système qui a été décomposé en un ensemble de modules possédant une forte cohésion interne et une faible cohésion externe.

Les principes d'abstraction, d'encapsulation et de modularité travaillent en synergie. Un objet propose une limite nette autour d'une abstraction simple, tandis que l'encapsulation et la modularité proposent des barrières autour de cette abstraction.

- *Le principe de hiérarchie.* Ce principe fournit un ordonnancement des objets du système. Les deux hiérarchies principales que l'on peut rencontrer dans un système complexe sont la structure de classe (hiérarchie "est de type") et la structure d'objet (hiérarchie "est un élément de"). Cette hiérarchie induit la notion d'héritage. Cet héritage peut être simple ou multiple, ce qui correspond au fait qu'un objet puisse hériter des propriétés d'un ou de plusieurs objets parents.
- *Le typage des objets.* Le typage est la propriété, pour une classe ou un objet, de ne pas pouvoir changer de type (typage fort), ou, au contraire, de pouvoir changer de type, et ce, dans un nombre limité de cas (typage faible).
- *La concurrence.* C'est la propriété qui distingue un objet actif d'un objet passif. Cette propriété est surtout utilisée dans des systèmes où plusieurs événements peuvent survenir simultanément.
- *La persistance.* La persistance est la propriété pour un objet d'exister plus ou moins longtemps dans le système. C'est donc la durée pendant laquelle ses caractéristiques essentielles (ses attributs) sont mémorisées. Nous parlerons essentiellement de persistance d'un processus à un autre.

Les avantages apportés par le modèle objet sont multiples:

- tout d'abord, l'utilisation du modèle objet permet d'utiliser toute la puissance des langages de programmation orientés objets. En effet, avant l'utilisation de tels outils, les aspects les plus puissants de ces langages étaient le plus souvent ignorés ou mal utilisés.
- l'utilisation du modèle objet encourage la réutilisabilité, non seulement, des programmes, mais aussi de la conception (modélisation) entière.
- le modèle objet produit des systèmes qui ont une forme intermédiaire stable, qui leur permet d'évoluer dans le temps de manière beaucoup plus aisée qu'avec l'approche traditionnelle.
- le modèle objet réduit le risque de développement de systèmes complexes, tout d'abord parce que l'intégration est répartie sur l'ensemble de la durée du cycle de développement au lieu d'être entièrement réalisée en une seule fois, ensuite parce que la séparation en objets permet une répartition efficace des objectifs et des problèmes.

- enfin, le modèle objet rejoint le côté intuitif du travail de conception.

1.1.3. Les classes et les objets

Un *objet* est soit quelque chose de tangible ou de visible, soit quelque chose d'intellectuellement appréhendable, soit quelque chose vers quoi une action est dirigée. Il a donc un état, un certain nombre de comportements bien définis et une identité propre.

L'état d'un objet comprend non seulement toutes les propriétés de l'objet (aspect statique) mais aussi, les valeurs courantes de ces propriétés (aspect dynamique). Le comportement est la manière dont l'objet agit et réagit en termes de modifications de son état et de génération de messages lors d'actions menées sur lui par d'autres objets.

Un objet en soit n'est guère intéressant. Par contre, des objets possédant des relations entre eux contribuent au comportement général du système. Les relations entre les objets peuvent refléter des actions ou des agrégations d'objets. Dans ces actions, un objet peut jouer trois rôles différents:

- *Acteur* : il porte son action sur les autres objets mais n'est pas lui-même concerné par des actions. On parlera aussi d'objet Actif.
- *Serveur* : il subit les actions des autres objets et ne porte aucune action sur un autre objet. On parlera également d'Objet Passif.
- *Agent* : il est à la fois Acteur et Serveur, c'est à dire qu'il peut à la fois agir sur des objets et subir l'action d'autres objets.

Lorsqu'un objet envoie un message à un autre, ils doivent être synchronisés d'une certaine manière. Une nouvelle classification d'objets apparaît:

- les objets séquentiels : objets passifs dont les actions ne sont garanties que dans un flux simple de traitement.
- les objets bloquants : objets passifs dont les actions sont garanties dans un flux multiple de traitements.
- les objets concurrents : objets actifs dont les actions sont garanties dans un flux multiple de contrôle.

Une *Classe* est un ensemble d'objets qui possèdent une structure commune et un comportement commun. Une classe possède une partie interface et une partie implémentation, ce qui permet de cacher cette dernière à l'utilisateur par le masquage d'information.

L'interface d'une classe peut être divisée en trois parties:

- la partie *publique* : une déclaration faisant partie de l'interface de la classe et visible par tous les clients, eux-mêmes visibles depuis celle-ci.
- la partie *protégée* : une déclaration faisant partie de l'interface de la classe et visible uniquement de ses sous-classes.
- la partie *privée* : une déclaration faisant partie de l'interface de la classe et invisible des autres classes ou sous-classes.

Les relations entre les classes sont de différents types:

- les relations de généralisation : "est de type" ("kind of")
- les relations d'agrégation : "est un élément de" ("part-of")
- les relations d'association associent des objets qui normalement ne sont pas en relation. Par exemple, on peut associer les classes représentant des objets que l'on peut trouver sur un bureau. On associe ainsi la classe représentant un Ordinateur et la classe Papier, alors que ces deux classes sont totalement différentes.

Ces différentes relations peuvent être combinées pour donner :

- les *relations d'héritage* : une sous-classe hérite la structure et le comportement de ses super-classes;
- les *relations d'utilisation* : une classe utilise une autre classe;
- les *relations d'instantiation* : une classe est instantiée en objets ou sous-classes selon un certain nombre de paramètres. Cette classe est dite paramétrique ou générique;
- les *relations de méta-classes* : une classe est méta-classe si ses instances sont elles-mêmes des classes.

Durant l'analyse et les premières phases de la conception, le développeur a deux tâches principales:

- identifier les classes et les objets qui forment le vocabulaire du domaine du problème, ce sont les abstractions-clés,
- inventer les structures dans lesquelles des ensembles d'objets travaillent ensemble pour fournir un comportement qui satisfait les spécifications du problème, ce sont les mécanismes.

1.1.4. La classification

L'identification des classes et des objets est la partie la plus difficile dans la conception orientée objets. Cette identification est à la fois une découverte des abstractions-clés et une invention des mécanismes.

La *classification* est le moyen par lequel les connaissances sont ordonnées. En effet, la classification qui résulte de l'analyse est un compromis entre différentes solutions possibles. Dans la phase d'analyse, l'identification des classes et des objets formant l'espace du problème permet de modéliser le problème. Tandis que la phase de conception se penche plus sur les abstractions et les actions.

La classification est un processus incrémental et itératif, incrémental dans le sens où le développeur avance progressivement dans le domaine du problème et itératif car de nouvelles découvertes peuvent remettre en cause les décisions précédentes et nécessiter un retour en arrière.

L'identification des classes et des objets peut se faire selon trois approches: la classification par catégorie (selon les propriétés), les regroupements conceptuels

(classification par concepts) et la théorie du prototype (classification par association avec un prototype).

1.2. La méthode

Pour l'auteur, la démarche se déroule entièrement dans l'esprit du concepteur. Les schémas que l'on obtient généralement ne sont qu'une simple représentation de ce qui a été conçu. Les notations proposées par Booch sont très nombreuses et très précises quant à leur définition et utilisation.

Dans cette activité, il est important de regarder le problème sous différents angles. Il y a des structures logiques et physiques, des sémantiques statiques et dynamiques. Tout ceci peut être représenté de la manière suivante:

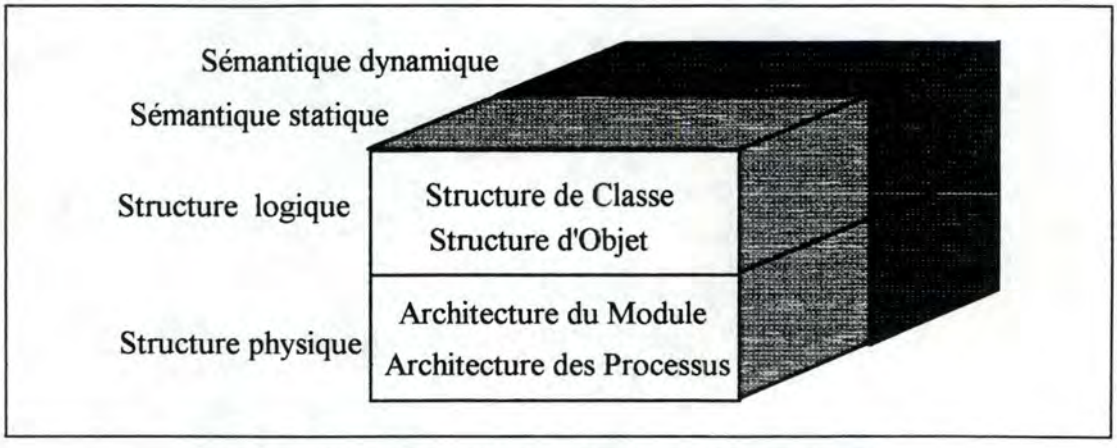


Figure A1.1-1: Le modèle de la Conception Orientée-Objets.

1.2.1. Les notations utilisées

De ces notations, nous ne reprendrons que les plus importantes. La lecture du livre de l'auteur fournira de plus amples détails.

La représentation de la conception orientée objets comporte quatre diagrammes principaux.

Le *diagramme des classes* montre l'existence de classes et leurs relations dans la conception logique du système. Il représente tout ou partie de la structure de classe d'un système. Il est composé de trois éléments principaux: les classes (icônes sans forme précise et portant un nom), les relations entre classes (des flèches simples, doubles ou pointillées, portant un nom et une direction), les catégories de classes (des rectangles portant le nom de la catégorie).

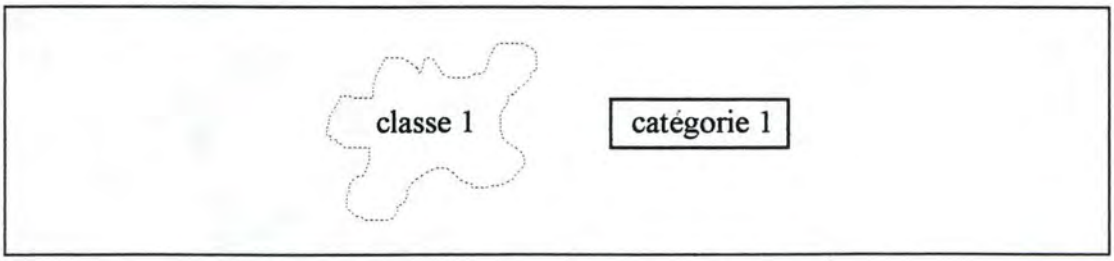


Figure A1.1-2: Notations du diagramme des classes.

Le *diagramme des objets* montre l'existence des objets et des relations dans la conception logique du système. Il représente tout ou partie de la structure objet d'un système ainsi que la sémantique des actions et méthodes de l'objet. Les relations entre les objets représentent les messages qui sont envoyés et reçus. Ils sont représentés par une simple ligne. De plus, nous pouvons ajouter à ces messages des notions de synchronisation et de visibilité sur ces relations.

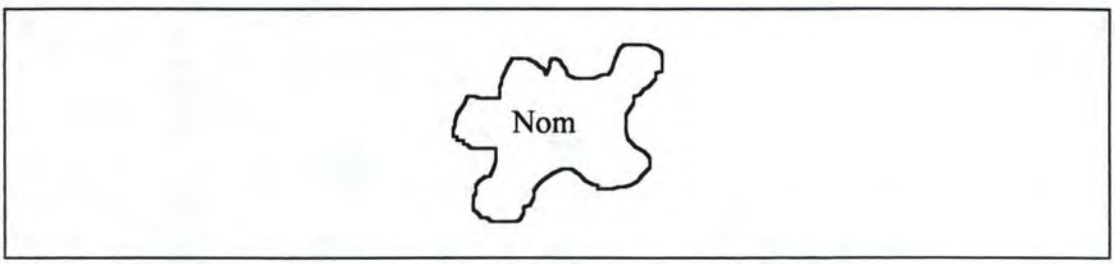


Figure A1.1-3 : Notations du diagramme des objets.

Le *diagramme des modules* montre l'implémentation des classes et des objets en modules dans la conception physique du système. Il représente tout ou partie de l'architecture des modules dans le système. Nous utiliserons divers symboles pour représenter les différents types de modules.

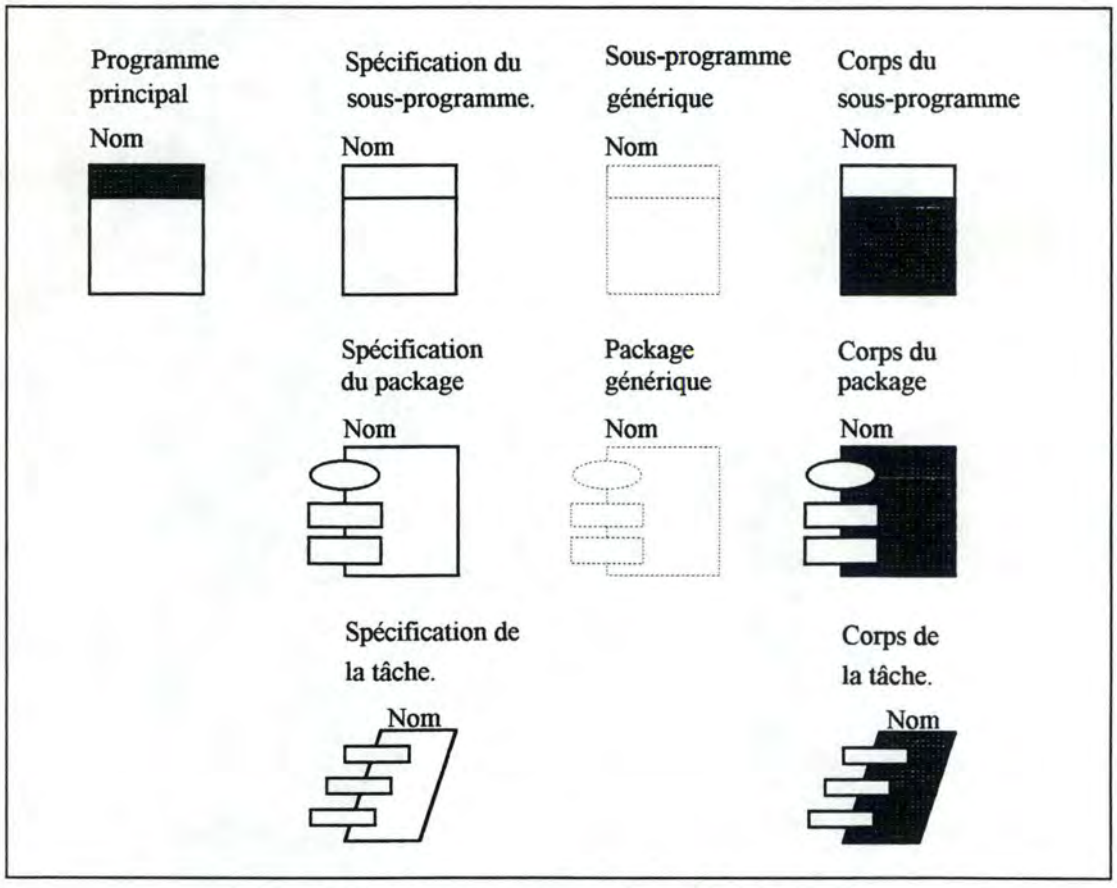


Figure A1.1-4 : Notations pour le diagramme des modules.

Un certain nombre de modules peuvent eux-mêmes être regroupés au sein d'un sous-système afin de fournir un niveau d'abstraction plus élevé et donc de faciliter la lecture. Ces sous-systèmes sont représentés par des rectangles dont les coins sont arrondis.

Le *diagramme des processus* montre l'implémentation des processus dans la conception physique du système. Ce diagramme est composé de processeurs, de dispositifs (device) et de connections. Un processeur est une partie du Hardware capable d'exécuter un programme, le dispositif n'a pas ce pouvoir d'exécution et les connexions sont les liens existants entre les différents processeurs et dispositifs (câble, ligne hertzienne, ...). La représentation de ces éléments est indiquée ci-dessous.

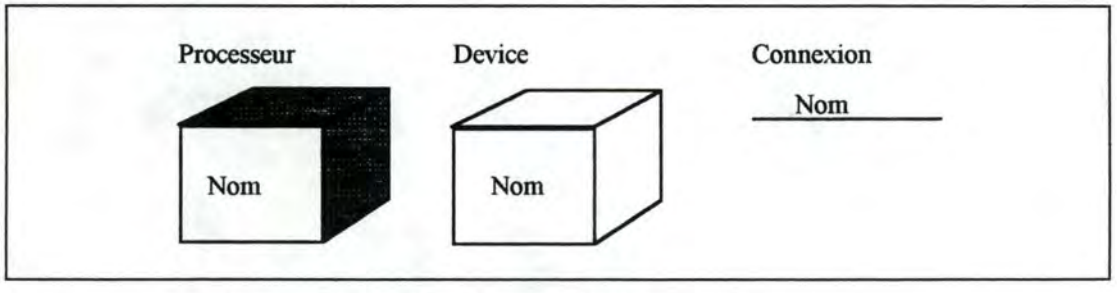


Figure A1.1-5 : Notations pour le diagramme des processus.

Deux diagrammes supplémentaires vont aider à la compréhension et à la conception.

Le *diagramme d'état-transition* montre l'aspect dynamique d'une classe, en montrant les états, les événements et les actions qui résultent des changements d'état. Celui-ci est représenté par un cercle portant un nom. L'état dans lequel se trouve la classe lors de la création d'une instance de cette classe est appelé état de départ. Celui à partir duquel l'instance peut être détruite est appelé l'état final. Les événements et les actions sont représentés par une flèche portant le nom de ceux-ci.

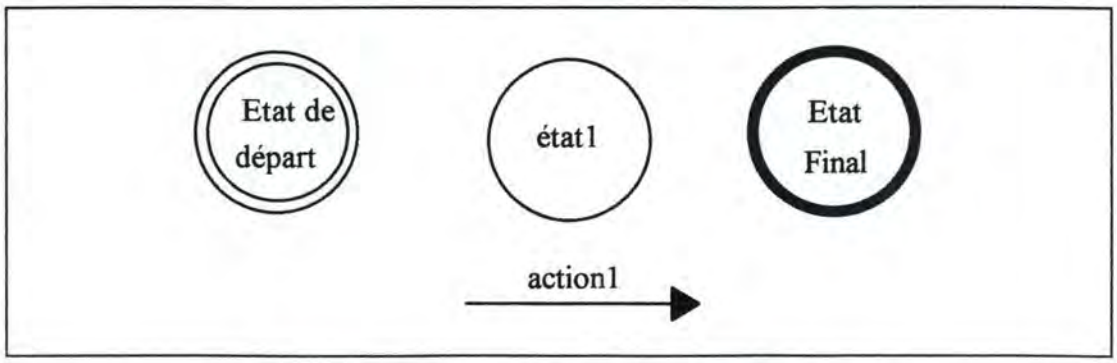


Figure A1.1-6 : Notations du diagramme état-transition.

Le *diagramme du temps* montre les interactions dynamiques entre différents objets dans le diagramme d'objets. Il consiste en la représentation, sur une ligne du temps, de l'exécution d'une opération, des opérations qui sont invoquées, ainsi que de la synchronisation requise lors de cette exécution.

1.2.2. Le processus de conception

Le processus de conception orienté objets n'est ni une approche Top-Down, ni une approche Bottom-Up mais plutôt une "Round-Trip Gestalt" [BOO91, p188]. Ce style de conception amplifie le développement incrémental et itératif d'un système à travers les raffinements successifs d'éléments logiques ou physiques déjà identifiés dans le système. On emploie donc un processus évolutif qui permet d'aborder le problème du point de vue convenant le mieux à ce moment-là du cycle de développement.

La conception orientée objets n'est pas un processus qui commence avec les spécifications, qui se termine avec une implémentation et qui possède une boîte noire entre ces deux extrêmes. La conception orientée objets est généralement composée de quatre étapes :

- *identification des classes et des objets à un niveau d'abstraction donné.* Les activités les plus importantes sont l'identification des abstractions-clés dans le domaine du problème et l'élaboration des opérations qui fourniront le comportement requis des objets. Les collaborations entre ces objets permettront de réaliser certaines fonctions.
- *identification de la sémantique de ces classes et objets.* Le concepteur doit agir comme quelqu'un d'extérieur au système. Il doit étudier chaque

classe à travers son interface, et donc identifier les opérations que l'on peut faire sur chaque instance de cette classe et celles que chaque objet de celle-ci peut faire sur un autre objet.

- *identification des relations existantes entre ces classes et objets.* Le but est d'établir comment les classes et les objets interagissent dans le système et non plus simplement entre classes. Ces interactions doivent être vues aussi bien de manière statique que dynamique.
- *implémentation de ces classes et objets.* La principale activité pour cette étape est de choisir la représentation pour chaque classe et objet, de la placer dans des modules et d'implémenter les opérations. Cette étape n'est pas nécessairement la dernière car sa réalisation nécessite habituellement la répétition de tout le processus à un niveau d'abstraction plus faible.

Ce processus est incrémental car l'identification de nouvelles classes ou objets cause généralement une révision de la sémantique et des relations entre les classes et les objets existants.

Le processus de conception commence par la découverte des classes et des objets qui forment le vocabulaire de l'espace du problème. Il se termine lorsque de nouvelles abstractions ou opérations élémentaires ne peuvent plus être trouvées, ou lorsque toutes les classes et les objets qui ont déjà été découverts peuvent être implémentés par la composition d'éléments logiciels réutilisables.

1.2.3. Le rôle et les impacts de la méthode

Dans le cycle de développement, l'impact de la méthode est très important. Le modèle relativement statique de la Chute d'Eau où les étapes s'enchaînent l'une après l'autre sans retour en arrière est abandonné pour adopter un cycle de développement qui utilise la méthode de la conception orientée objets où l'accent est mis sur les aspects incrémental et itératif du processus de développement.

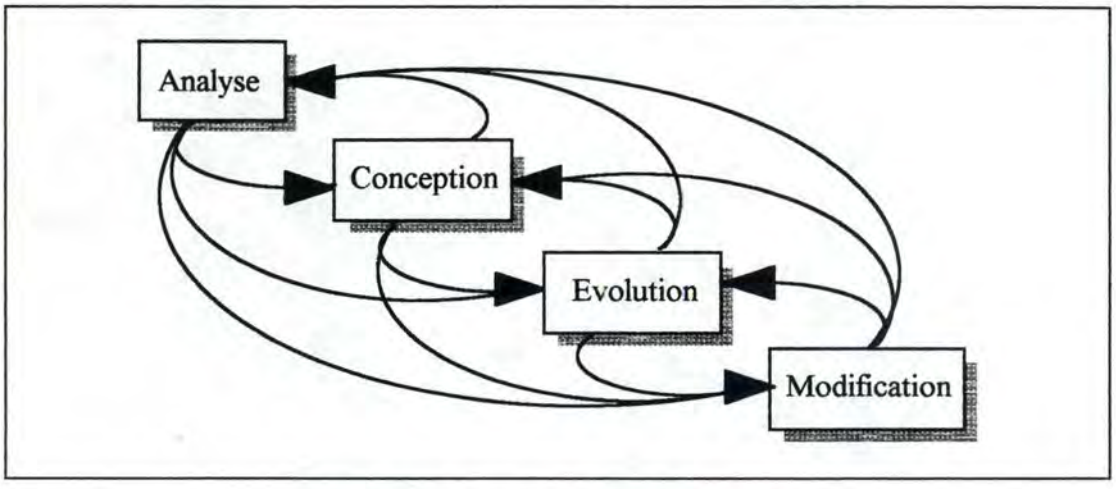


Figure A1.1-7: Le cycle de développement.

Dans ce cycle de développement, *la phase d'analyse* permet aux développeurs et aux utilisateurs de déterminer un vocabulaire commun. Il existe différentes

méthodes réalisant cette phase, telles que celles proposées par Coad et Yourdon [COA92] et par Shlaer et Mellor [SHL88].

La phase de conception peut débuter à partir du moment où l'on dispose d'un modèle, même incomplet, du problème qui doit être résolu. La conception est essentiellement le moment où l'on scinde le problème en sous-problèmes, ceux-ci étant à nouveau décomposés. Elle s'arrête lorsque les abstractions-clés sont suffisamment simples pour ne plus réclamer de subdivisions, mais être composées d'éléments existants réutilisables.

La phase d'évolution combine les aspects de codage, de tests, et d'intégration. Il n'y a donc pas d'intégration brutale, mais plutôt une intégration incrémentale à tous les niveaux. Les changements que cette phase peut apporter sont de différents types: ajout d'une nouvelle classe, changement de l'implémentation ou de la représentation d'une classe, réorganisation d'une structure de classe et changement de l'interface d'une classe. Chacun de ces changements peut avoir une raison différente, ainsi qu'un coût très différent. Parfois, l'ajout d'une classe peut provoquer des modifications qui peuvent conduire à une révision de tout le système sur un autre niveau d'abstraction, ce qui occasionne des coûts parfois imprévus et non négligeables en termes de ressources humaines et matérielles.

Les modifications sont essentiellement celles qui ont lieu lors de la maintenance et sont donc des ajouts aux spécifications de départ. L'ajout de nouvelles fonctionnalités au système peut être plus facile et plus rapide que dans les conceptions de type traditionnel si la méthode est suivie correctement.

La gestion d'un projet appliquant cette méthode est généralement plus facile, le temps de réalisation est plus court et les logiciels produits sont de meilleure qualité et respectent mieux les spécifications.

Les différentes phases de la conception produisent un certain nombre de documents qu'il serait bon de regrouper. Six types de documents peuvent ainsi être produits:

- spécifications des fonctions,
- diagrammes de classes et d'objets,
- modèles de classes et d'objets,
- diagrammes de modules et d'opérations,
- modèles de modules et d'opérations,
- résultats des prototypes exécutables.

Ces documents peuvent êtres intégrés dans divers outils semi-automatiques.

A1.2.

La méthode de COAD et YOURDON

Cette méthode, l'Analyse Orientée Objets (AOO), est fondée sur les concepts de base de cette nouvelle approche. Celle-ci est proposée par Peter **COAD** et Edward **YOURDON**. Ces auteurs ont d'abord édité une première version de leur méthode qui comportait certaines agrégations de concepts qui la rendait relativement simpliste. En outre, elle ne permettait pas de modéliser toute l'information nécessaire. Les auteurs ont corrigé ces défauts dans une seconde édition qui apparaît comme beaucoup plus complète dans sa couverture des différents concepts de l'orienté objets. C'est sur une traduction de celle-ci que nous nous appuierons dans le cadre de ce travail [COA92].

Dans une première section, nous évoquerons les aspects globaux de la méthode. Dans les sections suivantes, nous présenterons les cinq activités principales. Chacune de ces sections comportera deux sous-sections: la représentation graphique et la démarche nécessaire pour mener à bien l'activité étudiée. Nous terminerons en évoquant le passage de l'analyse orientée objets à la conception orientée objets.

2.1. L'approche globale de cette méthode

L'approche globale est constituée de cinq activités principales:

- identifier les classe&objets,
- identifier les structures,
- identifier les sujets,
- définir les attributs,
- définir les services.

Ces activités ne sont pas séquentielles. En effet, l'identification des classe&objets nécessite parfois de nouvelles structures, de nouveaux attributs, ... Nous avons donc différents niveaux d'abstraction.

Durant la phase d'analyse, le passage d'un niveau à un autre se fera de manière quelconque. Par contre, la démarche générale et globale suivra plus ou moins l'ordre de ces activités. Cette dernière sera reprise plus tard.

Le modèle lui-même est constitué de cinq couches:

- la couche Sujet,
- la couche Classe&objets,
- la couche Structure,
- la couche Attribut,
- la couche Service.

Chacune de ces couches présente un certain nombre de détails. La superposition de celles-ci permet d'en montrer certains et d'en cacher d'autres, à la manière de transparents que l'on superposerait. Le modèle est complet présenté lorsque les cinq couches sont présentées simultanément.

Nous allons maintenant passer en revue les diverses activités mentionnées ci dessus.

2.2. Identification des classe&objets

Cette section va donner les guides spécifiques pour identifier les classe&objets.

2.2.1. La représentation graphique

Il existe deux types de représentation: la classe&objets et la classe qui pourrait être qualifiée d'abstraite.

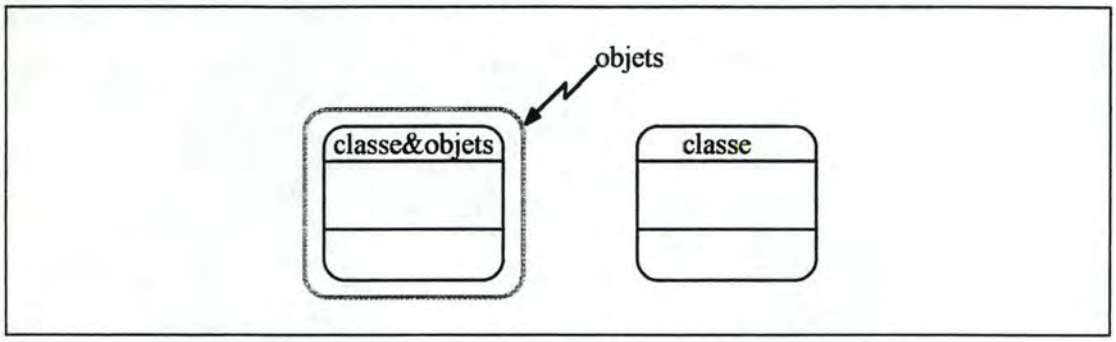


Figure A1.2-1: Les symboles "classe&objets" et "classe" en AOO.

2.2.2. La démarche

Les différents concepts recherchés sont: l'objet, la classe et la classe&objets.

"L'objet : une abstraction de quelque chose du domaine du problème, reflétant les capacités d'un système à garder de l'information sur elle, d'interagir avec elle ou de faire les deux choses à la fois; une encapsulation des valeurs des attributs et des services qui leurs sont spécifiques (synonyme: une instance).

> *La classe* : une description d'un ou plusieurs objets à l'aide d'un ensemble uniforme d'attributs et de services, avec une description de la manière de créer de nouveaux objets dans la classe.
>
> *Classe&objets*: un terme qui signifie "une classe et les objets qu'elle contient"."
>
> [COA92, p.44]

Généralement, un nom singulier sera donné. Celui-ci décrira un objet de la classe. Le vocabulaire généralement utilisé sera emprunté au maximum au sein du système analysé.

Pour identifier ces éléments, un certain nombre de pistes de recherche seront suivies. Il convient d'insister sur l'importance de cette étape car d'elle va dépendre la suite de l'analyse. Ces pistes de recherche sont l'observation du domaine du problème (observer un des acteurs dans ce domaine), l'écoute des experts de ce domaine, la vérification des résultats précédents de l'AOO (dans des domaines semblables), la lecture attentive du cahier des charges,et le prototypage.

Pour identifier ces objets, il faut également rechercher les structures internes au domaine du problème, les autres systèmes qui interagissent avec lui, les périphériques avec lesquels il échange des informations, les éléments à mémoriser, les différents rôles joués par les acteurs, les procédures opérationnelles qu'il doit maintenir, les sites (endroits physiques) que il doit connaître, les unités organisationnelles existant dans le système, ...

Ensuite, la pertinence de chaque classe candidate au sein du système considéré sera contrôlée. Pour cela, il faudra étudier et vérifier la mémorisation souhaitée, le comportement, les multiples attributs, l'existence de plus d'un objet par classe, constater que les attributs et services sont toujours applicables, etc.

Une fois ces classes et classe&objets identifiées, nous allons identifier les structures.

2.3. Identification des structures

> " *Structure* : une structure est une expression de la complexité du domaine, liée aux responsabilités du système. Le terme structure est utilisé comme terme global, décrivant la structure de généralisation-spécialisation (Gen-Spec) et la structure de composé-composants ."
>
> [COA92, p.66]

La structure Gen-Spec peut être considérée comme une structure de distinction entre classes, tandis que la structure composé-composants est vue comme une structure de décomposition. Si une classe participe aux deux types de structure, nous avons une structure multiple.

2.3.1. La représentation graphique

Ces deux types de structure auront deux types de représentation. Dans la structure Gen-Spec, les lignes de la représentation sont directement connectées aux classes afin de montrer que la relation a lieu entre les classes plutôt qu'entre les objets.

Par contre, les lignes de la structure composé-composants sont reliées aux objets car cette structure montre une liaison entre ceux-ci plutôt qu'une liaison entre les classes.

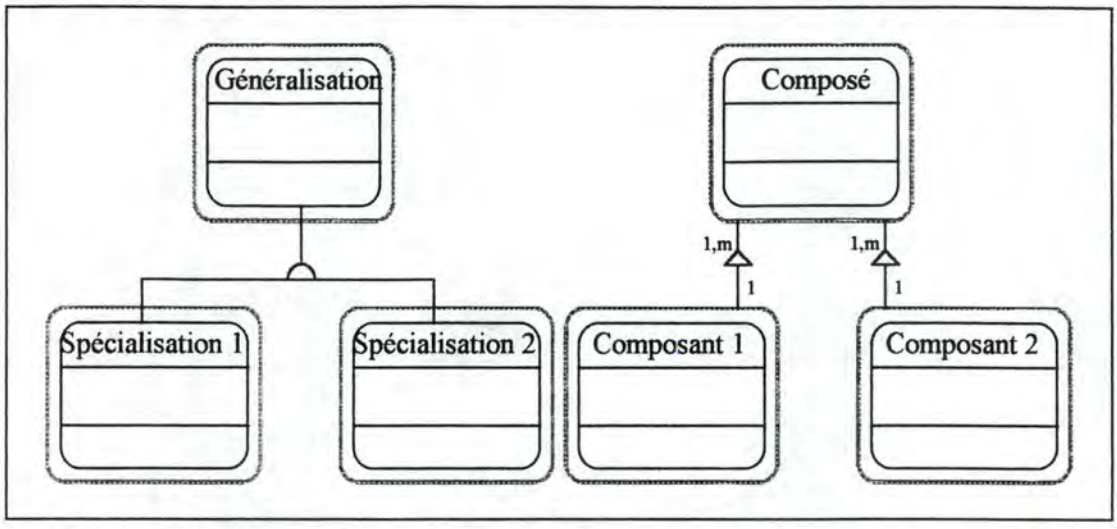


Figure A1.2-2 : Représentation graphique des structures Gen-Spec et composé-composants.

2.3.2. La démarche pour les structures Gen-Spec

Pour les identifier, chaque classe sera considérée comme une généralisation. Les questions suivantes seront posées pour chacune de ces spécialisations candidates: est-elle dans le domaine du problème, dans les limites des responsabilités du système, y a-t-il héritage, ...? Il sera fait de même en prenant l'autre point de vue, c'est-à-dire en considérant toutes les classes comme des spécialisations.

La forme la plus courante de Gen-Spec est la hiérarchie, mais on peut également utiliser un treillis pour montrer des spécialisations supplémentaires, extraire des éléments communs ou limiter la complexité du modèle. Ici, un treillis est la possibilité pour une classe d'être la spécialisation de plus d'une généralisation distincte.

2.3.3. La démarche pour les structures composé-composants

L'identification de ces structures doit prendre en compte des variantes telles que l'assemblage-parties, le contenant-contenu, la collection-membres.

De manière similaire à l'identification des Gen-Spec, chaque classe sera considérée comme un composé et les mêmes questions seront posées pour les classes composantes candidates. Il faudra ensuite faire la même chose dans l'autre sens en considérant les classes comme des composantes à la recherche de la classe composée potentielle. De plus, il faudra se poser la question de savoir si ces classes prennent en compte autre chose que la valeur d'un état et, dans la négative, étudier la possibilité d'inclure simplement un attribut dans la classe composée.

Une fois cette démarche réalisée pour les classes, elle sera recommencée pour les classe&objets.

2.4. Identification des sujets

La définition des sujets donnée par les auteurs est la suivante:

> "*Sujet* : mécanisme pour guider un lecteur (analyste, expert du domaine du problème, manager, client) à travers un grand modèle complexe. Les sujets sont également utiles pour organiser le travail en lots sur les gros projets, à l'aide des investigations initiales de l'AOO."
>
> [COA92, p.88]

Les sujets permettent de tenir compte de la fameuse règle de Miller : "*7 objets +/- 2*" qui donne le nombre idéal d'objets à visualiser simultanément afin de garder et guider l'attention du lecteur.

2.4.1. La représentation graphique

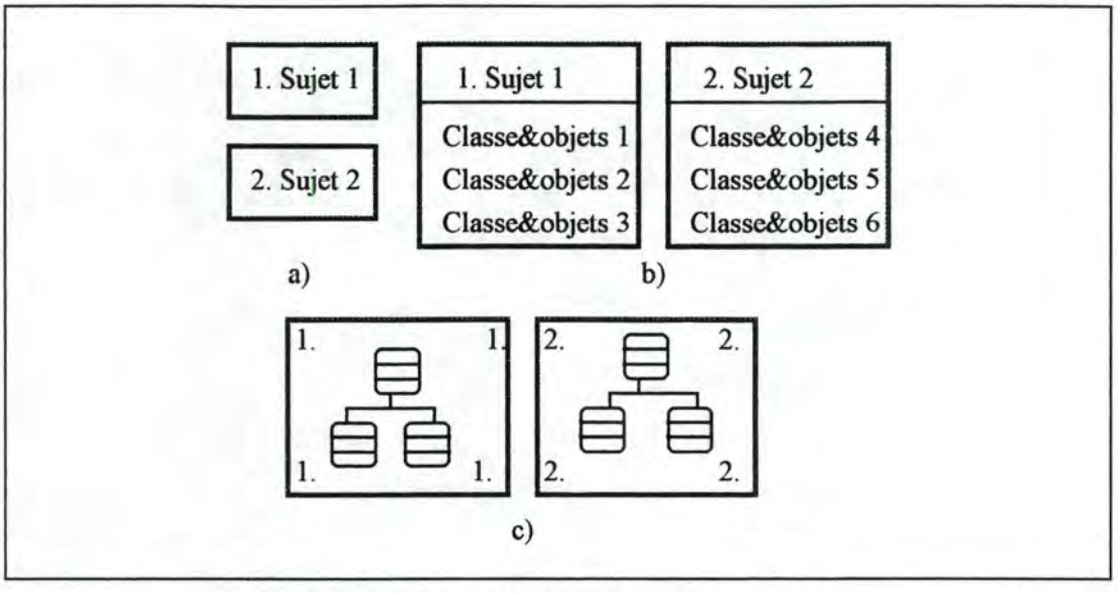


Figure A1.2-3 : Représentation graphique des sujets:
a) fermée, b) partiellement ouverte, c) ouverte.

Ces différentes représentations permettent de visualiser un certain nombre de choses, de cacher certains détails ou de simplifier au maximum la vue du problème.

2.4.2. La démarche

Pour sélectionner les sujets, les classes de haut niveau de chaque structure seront placées vers le haut d'un sujet et il sera fait de même pour les classe&objets n'intervenant pas dans une structure. Les sujets seront ensuite validés en vérifiant les sous-domaines du problème qui doivent plus ou moins correspondre avec les sujets. De façon similaire, il faudra vérifier que les relations entre les sujets et entre des classes se trouvant dans des sujets différents soient minimisées.

Les classes et les classe&objets seront placées dans les sujets ouverts (figure A1.2-3c). Lorsque l'on aura besoin de cacher ces classes, deux possibilités

seront offertes: soit garder les noms des objets se trouvant dans ce sujet (figure A1.2-3b), soit ne montrer que le nom du sujet (figure A1.2-3a).

Il ne faut pas oublier qu'au fur et à mesure que l'on ajoute des classes, il faut créer régulièrement de nouveaux sujets afin de guider le lecteur.

Des sujets peuvent contenir d'autres sujets conduisant à une modélisation multi-niveaux permettant de représenter des problèmes relativement importants tout en conservant une représentation claire et précise.

2.5. Définition des attributs

En AOO,

> "*Attribut*: un attribut est une donnée (information d'état) pour laquelle chaque objet dans une classe a une valeur propre."
>
> [COA92, p.100]

Continuant à détailler les objets, chaque classe&objets et chaque classe va comporter au moins un attribut.

2.5.1. La représentation graphique

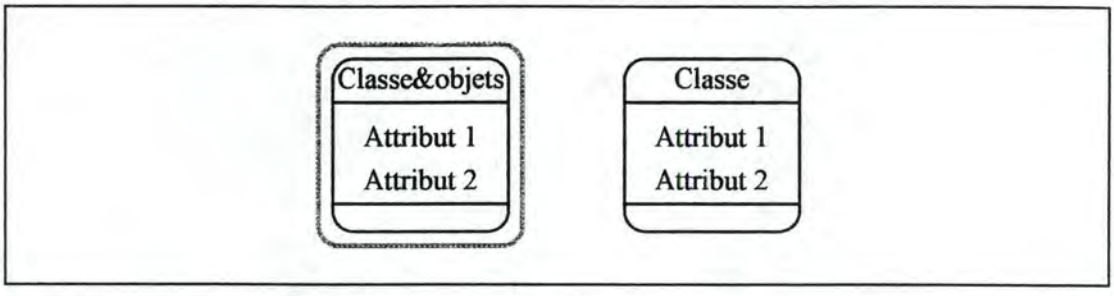


Figure A1.2-4 : Représentation graphique des attributs.

2.5.2. La démarche

La première phase d'identification des attributs consiste à se poser diverses questions. Comment l'objet est-il décrit en général, dans le domaine du problème et dans le contexte des responsabilités du système? Que doit-il mémoriser? ...

Ensuite, l'héritage des structures Gen-Spec sera étudié, c'est-à-dire qu'il conviendra de placer les attributs les plus généraux vers les hauts niveaux et les attributs spécialisés dans les niveaux plus bas.

2.5.3. Les connexions d'instances

Si les attributs représentent les états des objets, les connexions d'instances que l'on va ajouter vont montrer les liaisons requises demandées par un objet pour prendre en charge ses responsabilités. C'est donc une modélisation de l'association existant entre divers objets.

"*Connexion d'instances*: une connexion d'instances est un modèle des liaisons du domaine du problème qu'un objet doit avoir avec d'autres objets dans le but de remplir ses obligations."

[COA92, p. 106]

Une connexion d'instances est représentée par une ligne joignant deux objets (plutôt qu'entre classes). De plus, des cardinalités seront placées sur chacune de ces extrémités afin de rendre compte des contraintes existant entre ceux-ci.

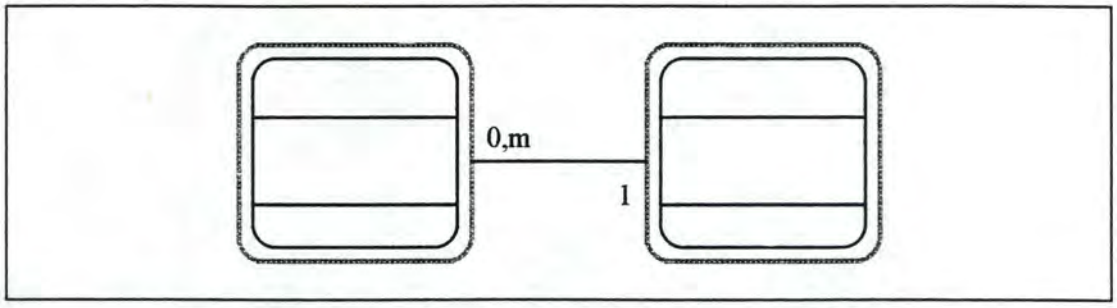


Figure A1.2-5 : Représentation graphique des connexions d'instances.

La démarche suivie est l'ajout de ces connexions d'instances chaque fois qu'un objet possède des relations dans le domaine du problème et les responsabilités du système. Il convient d'ajouter les cardinalités se trouvant sur ces relations.

2.5.4. Vérification de certains cas spéciaux

Ces cas spéciaux sont, entre autres:

- les attributs ayant une valeur dans certains objets et n'étant pas applicables dans d'autres et qui indiquent la présence d'une Gen-Spec.
- les classe&objets qui n'ont qu'un seul attribut et qui indiquent un mauvais placement d'attribut.
- les attributs ayant des valeurs multiples et qui indiquent une classe&objets qui devrait être décomposée en plusieurs.
- les connexions d'instances M-N. Il faut chercher les attributs qui décrivent cette relation et créer une nouvelle classe&objets reliée par des connexions 1-M.
- les connexions d'instances multiples entre objets qui nécessitent généralement la création d'une classe&objets entre les objets et la connexion de ceux-ci.
- ...

2.5.5. Spécification des attributs

Pour chacun des attributs, nous donnerons un nom, une description et des contraintes. Ces contraintes comprennent les unités de mesures, intervalles de valeurs, valeurs par défaut, la précision et les contraintes de création ou d'accès.

Ces spécifications peuvent êtres reprises dans des canevas très utiles et non graphiques afin de ne pas surcharger ce type de modèle.

2.6. Définition des services

"*Service*: un service est un comportement spécifique qu'un objet est en charge de fournir."

[COA92, p. 119]

2.6.1. La représentation graphique

La représentation graphique est donnée à la figure A1.2-6.

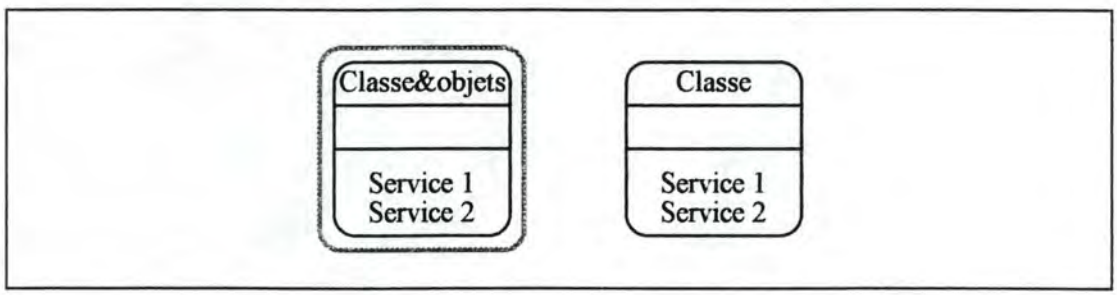


Figure A1.2-6 : Représentation graphique des services.

2.6.2. La démarche

La démarche comprend les activités suivantes:

- identifier les états de l'objet,
- identifier les services nécessaires,
- identifier les connexions de messages,
- spécifier les services.

2.6.2.1. Les états d'un objet

Les objets, au cours de leur vie, passent par différents états. Un état est donné par la valeur de ses attributs. Un changement de valeur implique donc un changement d'état. Nous pouvons représenter ces différents états par un graphique particulier. C'est le diagramme d'état. Un rectangle représentera l'état et une flèche la transition, le changement d'état correspondant. Le nom du service responsable de la transition sera ajouté sur la flèche la représentant.

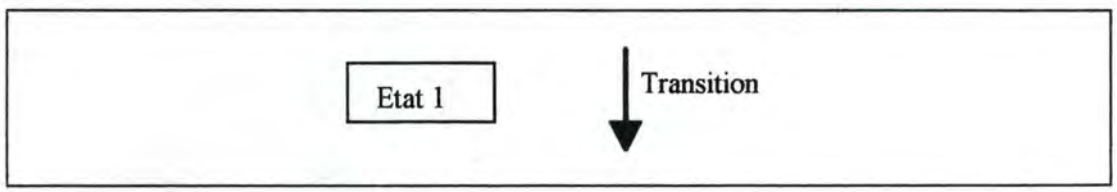


Figure A1.2-7 : Représentation graphique d'un diagramme d'état d'un objet.

2.6.2.2. Identification des services nécessaires

Des algorithmes simples tels que créer, connecter, accéder, libérer sont immédiatement identifiés. En effet, ils se retrouvent dans tous les objets. D'autres services sont également nécessaires: les services à algorithmes complexes. Nous en avons de deux types: calculer et superviser.

2.6.2.3. Identification des connexions de messages

> "*Connexion de messages*: une connexion de messages modélise la dépendance des traitements d'un objet indiquant un besoin en services lui permettant de remplir ces obligations."
>
> [COA92, p. 124].

Chaque fois qu'un service a besoin d'une information ou d'un autre service se trouvant dans un autre objet se produit un échange de messages. Ceux-ci seront représentés par une flèche partant de l'objet qui envoie le message, l'émetteur, et arrivant sur l'objet qui va fournir l'information ou le service, le récepteur.

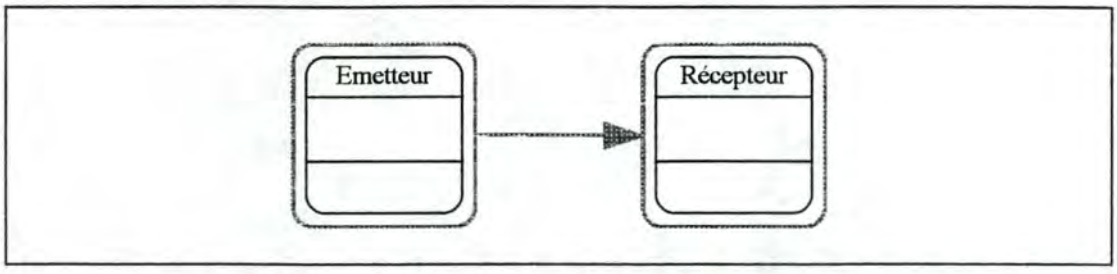


Figure A1.2-8 : Représentation graphique d'une connexion de messages.

La démarche suivie lors de l'identification des connexions de messages nécessaires se compose de deux étapes:

- un questionnement quant aux services et informations nécessaires pour chaque service de l'objet ou que l'on sait nécessaires à d'autres services d'autres objets,
- la vérification de son résultat avec d'autres résultats déjà obtenus lors de développements AOO précédents.

Cette vérification permet de réutiliser ce qui a déjà été fait et peut d'ailleurs s'appliquer à tous les processus d'analyse et d'identification ci-dessus.

2.6.2.4. Spécification des services

Des diagrammes de services seront utilisés pour représenter l'algorithme de chaque service de manière graphique.

Il faut souligner que des commentaires peuvent être placés sur les connecteurs, ce qui est pratique pour indiquer le résultat d'une condition. De plus, les diagrammes de services se lisent du haut vers le bas. Les connecteurs "partent" d'un élément en étant connectés au bas de la figure géométrique et "arrivent" par le haut de la figure

de l'élément visé. Il conviendra donc de faire attention à ce style de notation lors d'algorithmes complexes. Cependant, cette notation est suffisante car les organigrammes des services sont des graphes orientés. Ils ont tous une situation de départ et une situation de fin.

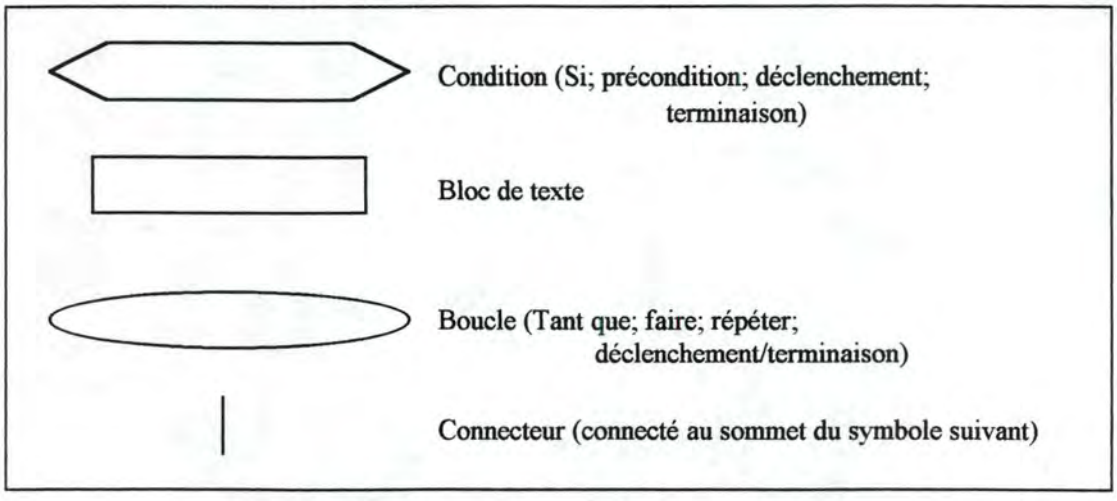


Figure A1.2-8 : Représentation des diagrammes de services.

Une table reprenant les services disponibles selon les différents états de l'objet permet de compléter leurs spécifications. Ils sont indiqués par un point.

	Etat 1	Etat 2	Etat 3
Service 1	•	•	
Service 2		•	•
Service 3	•	•	•

Figure A1.2-9 : Table services-états.

2.7. Mise en commun des éléments

Il reste maintenant à mettre ensemble tous ces éléments pour constituer la documentation du problème.

Cet ensemble va donc reprendre les cinq couches, les spécifications des classe&objets qui contiennent les spécifications des attributs et des services avec les diagrammes qui y sont définis.

2.8. Passage à la conception orientée objets

L'Analyse Orientée Objets identifie et spécifie les classes et les classe&objets qui représentent directement le domaine du problème. Par contre, la Conception Orientée Objets (COO) identifie et spécifie les classes et classe&objets supplémentaires représentant l'implémentation des besoins. Le passage de l'une à l'autre se situe donc dans le cycle de développement.

Durant la COO, quatre nouvelles composantes principales apparaissent:

- *l'interaction homme-machine* qui contient les affichages réels et les entrées nécessaires pour une interaction,
- *le domaine du problème* qui contient les résultats de l'AOO qui peuvent être revus pour des raisons de performance,
- *la gestion des tâches* qui contient la définition, la communication et la coordination des tâches temps réel, des protocoles externes, etc,
- *la gestion des données* qui contient l'accès et la gestion des données stables, le type gestion de ces données (relationnel, OO,...).

Nous voyons donc clairement apparaître le continuum entre l'AOO et la COO, qui se poursuivra avec la Programmation Orientée Objets, l'ensemble étant essentiellement un processus d'enrichissement du problème. L'AOO est totalement indépendante du langage de programmation choisi. La COO est indépendante du langage, mais est dépendante du type de langage choisi (procédural, par lot, orienté objets, ...). Les auteurs ont également produit un ouvrage traitant de la COO où l'on peut trouver plus de détails [COA91].

A1.3. La méthode de COLBERT

Edward **COLBERT** nous propose "The Object-Oriented Software Development Method" (OOSD). Cette méthode a été proposée lors de la conférence *TRI-Ada 89* et reprise dans les "TRI-Ada Proceedings" publiés par les ACM en Octobre 1989 [COL89].

Le but recherché par l'auteur est la réalisation d'une méthode aisée de développement de logiciels privilégiant les objets dés le début du cycle de développement. Il désirait en effet remplacer la phase d'analyse structurée par une phase d'analyse orientée objets lors de l'utilisation de langages orientés objets.

Dans cette présentation, nous allons définir un certain nombre de concepts. Ensuite, nous décrirons le cycle de développement qui se divise en une phase d'analyse et une phase de conception, possédant chacune un certain modèle de représentation.

3.1. Les concepts

Dans ces méthodes, l'élément de base est bien entendu l'objet. Sa définition permettra d'identifier les objets du problème, ainsi que ses caractéristiques qui permettront de définir ce qui doit être modélisé.

Un *objet* est une chose visible ou tangible d'une forme relativement stable. Elle peut être appréhendée intellectuellement; c'est une chose vers quoi ou à partir de quoi une action est dirigée. De plus, chaque objet peut être caractérisé d'une part par un ensemble d'opérations qui peuvent être réalisées sur lui ou sur d'autres objets et, d'autre part, par un ensemble d'états à travers lesquels il passe le long de son cycle de vie. Un objet est composé d'un ensemble d'éléments, par exemple des objets à partir desquels il a été construit.[COL91,p.1]

L'OOSD distingue deux types d'objets:

- les *objets actifs*, qui déclenchent les opérations sur d'autres objets, et ce de manière indépendante,
- les *objets passifs*, qui ne font que réagir aux opérations demandées par les objets actifs.

Une *classe* est un ensemble d'objets qui partagent un certain nombre d'états, d'opérations que l'on peut réaliser sur eux et d'opérations qu'ils peuvent déclencher sur les autres objets. Cette classe permet d'éviter une certaine redondance.

La *donnée* est également présente dans les objets. Elle permet de décrire les attributs et les états d'un objet.

L'OOSD va chercher à identifier les objets du problème, à déterminer les structures, les comportements et les propriétés de ces objets.

3.2. La phase d'analyse

Dans la phase d'analyse, le modèle présenté ici va chercher à montrer le QUOI: les objets, classes, fonctions, comportement et attributs du problème. Tandis que la conception va déterminer le COMMENT : l'architecture, etc.

La phase d'analyse va être composée de quatre activités afin de créer le modèle de l'application. Ces activités vont produire à la fois une représentation graphique et textuelle du modèle. Elles sont:

- la spécification des interactions entre les objets,
- la spécification des classes d'objets,
- la spécification du comportement,
- la spécification des attributs.

Ces activités pourront être réalisées dans n'importe quel ordre dès que la première aura été menée. Nous allons les passer en revue.

3.2.1. Spécification des interactions entre les objets

Cette activité va identifier les objets requis, leurs interactions et les relations hiérarchiques demandées. Deux types de représentations graphiques seront utilisés: le diagramme des interactions entre les objets (*Object-Interaction Diagram* OID) et le diagramme des hiérarchies d'objets (*Objet-Hierarchy Diagram* OHD).

Dans les OID, les objets seront représentés par des rectangles arrondis s'il s'agit d'objets actifs, par des rectangles ouverts s'il s'agit d'objets passifs et par des rectangles ordinaires pour les objets externes qui se situent en dehors du domaine du problème.

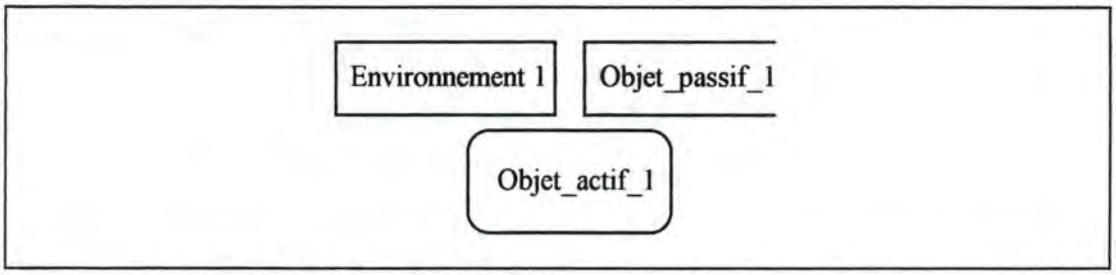


Figure A1.3-1: Représentations graphiques des différents types d'objets.

Un interaction implique une opération et des flux d'informations. Ces opérations sont symbolisées par des flèches partant de l'objet contenant l'opération qui a déclenché cette interaction et aboutit à l'objet qui est chargé de réaliser l'opération qui y est définie et qui est demandée. Les flux d'informations intervenant dans ces interactions peuvent être des flux de données, des flux d'objets, ou des flux d'erreurs. Ces différents flux seront représentés par les différents symboles repris dans la figure A1.3-2.

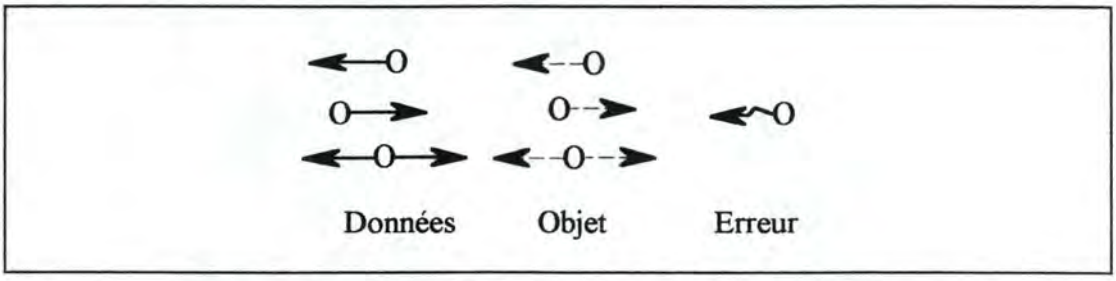


Figure A1.3-2: Représentations graphiques des flux d'informations.

Les décompositions des objets et de leurs interactions sont représentées par une hiérarchie de OID. Le niveau le plus haut est le diagramme du contexte. Il décrit l'objet système dans le contexte des objets se trouvant à l'extérieur et interagissant avec lui. Des OID additionnels sont crées afin de décrire la structure interne d'un objet, de la même manière que les "exploded views" dans les diagrammes de flux des données traditionnels.

Chaque diagramme interne décrit:

- les objets composant un objet se trouvant dans un diagramme de plus haut niveau,
- les interactions entre les objets composants,
- les interactions entre les composants et leur objet parent,
- les relations entre les interactions réalisées sur l'objet parent et celles réalisées sur les objets composants,
- les relations entre les interactions initiées par l'objet parent et celles initiées par les objets composants.

Pour reconnaître la vue interne d'un objet, une représentation agrandie du symbole de niveau supérieur sera placée autour de tous les composants de cet objet, sous forme d'un grand cadre. Nous aurons donc trois types de cadres selon la nature de l'objet dont nous avons la représentation interne.

Il y a quatre types de relations possibles dans ces interactions:

- une interaction sur un objet parent qui est décomposée en des interactions sur les objets composants de ce parent,
- une interaction initiée par un objet parent qui est seulement conceptuelle et qui est réellement initiée par un ou plusieurs objets composants de cet objet parent,

- une interaction interface par un objet parent qui est seulement conceptuelle et qui est réellement une interaction directe sur un objet composant de l'objet parent,
- une interaction sur un objet composant qui est initiée par un de ces objets parents actifs, indépendamment des initiations de n'importe quelle interaction sur l'objet parent.

L'OHD résume les relations établies dans les OID. Ils peuvent plus ou moins être déduits automatiquement des OID.

La figure suivante présente ces différents diagrammes.

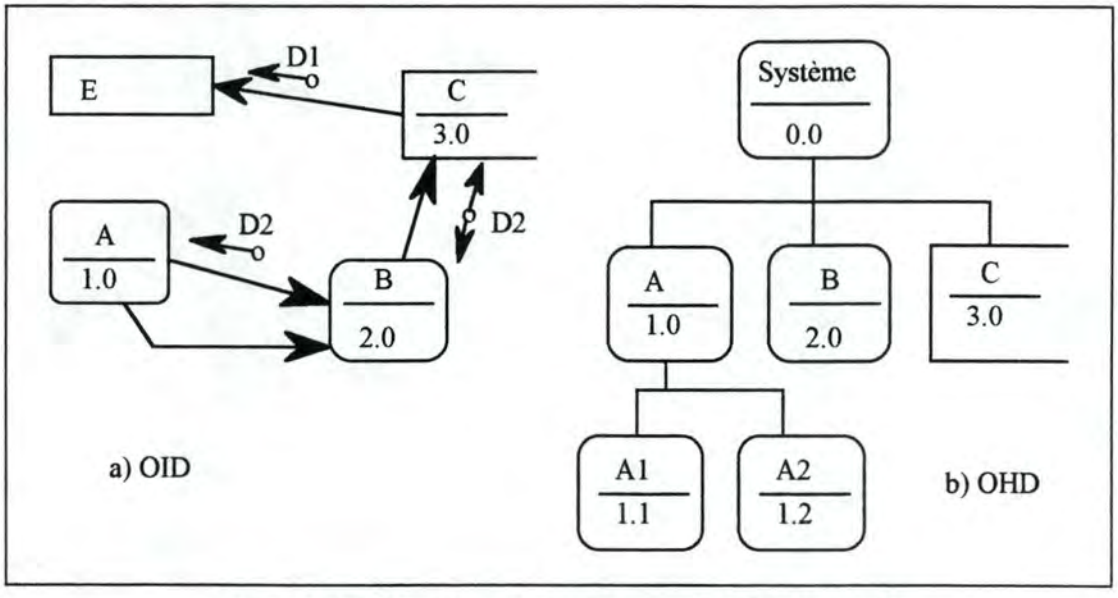


Figure A1.3-3: Représentation graphique de l'OID et de l'OHD.

Après la spécification des interactions, voici la spécification des classes d'objets.

3.2.2. Spécification des classes d'objets

L'activité principale de cette phase consiste en l'identification des classes d'objets et des relations entre elles en utilisant le Diagramme de Classes (Object-Class Diagram OCD). Celui-ci décrit la classe de chaque objet. De plus, les flux d'informations, dans les OID, indiquent celles en relation avec d'autres dans le système. Lorsque l'on identifie les classes d'un objet, le comportement et les structures de celles-ci sont également définis.

Ces OCD permettent de montrer différentes choses:

- les classes identifiées dès la première phase d'analyse,
- quels objets et flux d'informations du OID sont membres de quelles classes,
- les relations existant entre les classes,
- les structures et les comportements de chaque classe.

Durant cette phase d'analyse, seules les relations qui sont réellement nécessaires sont décrites.

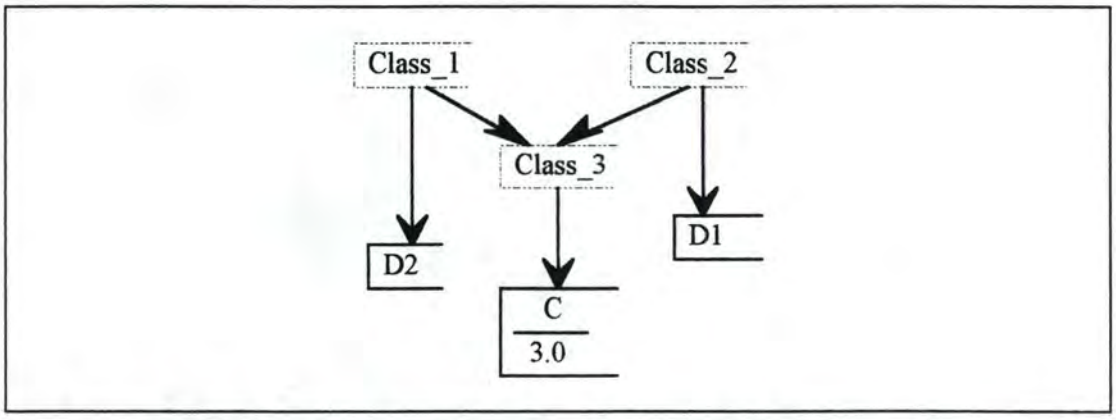


Figure A1.3-4 : Représentation graphique du diagramme des classes d'objets.

3.2.3. Spécification du comportement

L'OID donne une vue statique du comportement de chaque objet du système. En effet, il montre quels objets interagissent, mais pas sous quelles conditions. Sa spécification va remédier à ce problème en identifiant le comportement dynamique du système et de chacun de ses objets à l'aide d'un diagramme d'état-transition.

La représentation graphique choisie par l'auteur est reprise à la figure A1.3-4.

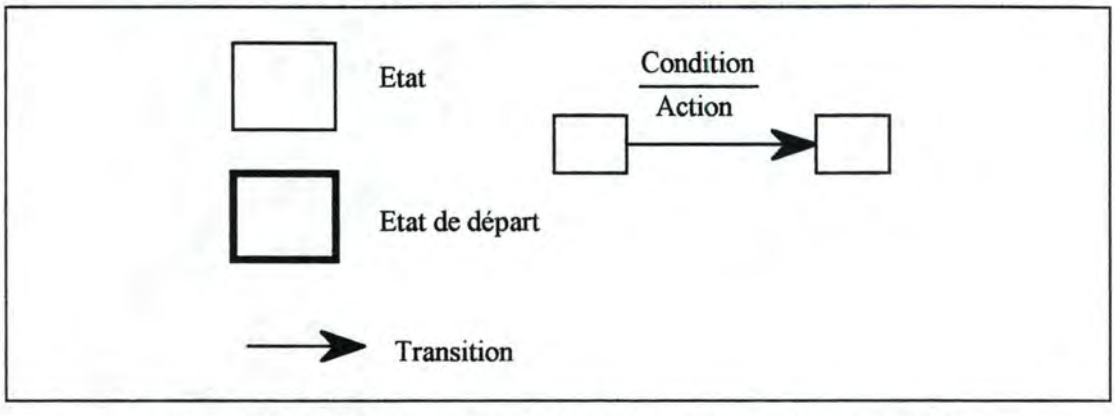


Figure A1.3-5 : Représentation graphique du diagramme d'état-transition.

Dans ce diagramme, le libellé de la flèche décrit la condition de réalisation de la transition ainsi que les actions dans la transition.

La spécification du comportement est composée de trois étapes principales:

- la description du comportement du système défini dans le problème qui est posé et l'analyse de la description pour vérifier la correction et la complétude,

- pour chaque objet du modèle, la définition des comportements externe et interne. Le comportement externe d'un objet met en relation les opérations qui sont réalisées sur lui à celles qu'il réalise sur les autres objets. Le comportement interne d'un objet met en relation les opérations qui sont réalisées sur lui et celles qu'il réalise sur les objets le composant.
- la démonstration du fait que le comportement de tous les objets du modèle est correct. Elle est réalisée lorsque l'on a déterminé ce qu'est un comportement correct pour chaque objet.

3.2.4. Spécification des attributs

L'activité de spécification des attributs identifie les mesures (caractéristiques) quantitatives et qualitatives, ainsi que les ressources qui seront nécessaires pour chaque objet du système. Nous utiliserons pour ce faire le formulaire de spécification des attributs qui va donner l'unité de mesure pour chaque qualité, caractéristique ou ressource du système ou d'un objet du système, de même que les quantités minimales et désirées qui devront être remplies.

Tous les attributs du système seront repris dans la table des attributs afin de vérifier les combinaisons nécessaires ou non désirables.

3.2.5. Résultats de cette analyse

Après toutes les étapes, nous obtenons un modèle complet du problème. Ce modèle décrit le "QUOI" désiré par l'intermédiaire d'un ensemble hiérarchisé d'objets qui possèdent certaines interactions, comportement et attributs. Le modèle est présenté de manière graphique et textuelle. De plus, la création d'un dictionnaire va résumer les résultats et faire le lien entre les différentes représentations.

3.3. La phase de conception

L'approche selon l'OOSD de la conception est essentiellement un raffinement du modèle produit dans la phase d'analyse vers une architecture logicielle. Il définit une représentation spécifique au langage de cette architecture.

Pour ce faire, il faut examiner une première étape qui sera une première conception. Suivra une conception plus détaillée.

3.3.1. Première conception

Cette conception va créer une description de la solution logicielle du problème de manière à rester indépendant de tout langage.

Le but poursuivi dans cette étape est la spécification de l'architecture logicielle du système. Pour cela, le modèle va être considéré comme une architecture de haut niveau à partir de laquelle seront ajoutés les éléments nécessaires afin de tenir compte du COMMENT le modèle sera implémenté.

Durant cette étape, la démarche choisie pour l'analyse sera en fait réutilisée après avoir été adaptée aux significations de cette première conception. Le résultat sera de ce fait un modèle de la solution reprenant les diagrammes décris à l'étape précédente.

3.3.2. Conception détaillée

L'étape suivante est la création d'une représentation tenant compte et intégrant les éléments nécessaires du langage d'implémentation qui a été choisi. Le but est donc maintenant de spécifier une implémentation de l'architecture logicielle produite par la phase de conception préliminaire.

Dans la méthode OOSD, cette étape est subdivisée en quatre activités. Celles-ci correspondent d'ailleurs aux étapes présentes dans la phase d'analyse:

- la spécification de l'architecture logicielle liée au langage choisi,
- la spécification des classes d'objets,
- la spécifications du comportement,
- la spécifications des attributs.

Ces étapes vont modifier les représentations graphiques et textuelles du modèle produites jusqu'ici. Elles vont induire de nouvelles représentations. Une fois la première étape réalisée, les trois autres peuvent être conduites dans n'importe quel ordre, mais l'auteur recommande néanmoins de respecter l'ordre ci-dessus. Ces activités nécessitent parfois l'utilisation des résultats des autres étapes. Il semblerait que l'ordre le plus performant soit celui donné par l'auteur.

3.3.2.1. Spécification de l'architecture logicielle liée au langage

Cette activité décrit, dans une notation spécifique au langage, les composants logiciels et les structures définies par l'architecture développée lors de la première conception. Pour chaque objet et classe définis dans les OID et les OCD des choix seront effectués afin de décider comment représenter ces objets et ces classes dans le langage d'implémentation.

Ce processus peut être plus ou moins automatisé. Cependant, le concepteur devra utiliser toute sa connaissance du langage afin de sélectionner efficacement les diverses alternatives de représentations.

Nous aurons un certain nombre d'activités:

- la représentation de chaque objet dans l'OID en tenant compte du langage d'implémentation,
- la représentation de chaque classe dans l'OCD en tenant compte du langage d'implémentation,
- la représentation de chaque interaction entre les objets dans l'OID,
- la révision du diagramme d'état-transition pour déterminer les détails du comportement qui peuvent affecter la structure du langage.

Une fois cette étape réalisée, nous pouvons passer à l'une des suivantes.

3.3.2.2. Spécification des classes d'objets

Dans cette activité seront identifiées les classes des objets logiciels dans l'architecture, ainsi que les relations entre les classes en utilisant le diagramme des classes d'objets.

Cette étape est essentiellement une opération de maintenance de documentation.

3.3.2.3. Spécification du comportement

L'activité de spécification du comportement représente le comportement dynamique de l'architecture liée au langage. De plus, elle démontre que le comportement de l'architecture implémente le comportement spécifié pour le modèle.

Un langage de conception de programme, le fameux PDL de Ada par exemple, est utilisé pour représenter le comportement des composants logiciels de manière textuelle.

La représentation du comportement dynamique du logiciel nécessite celle du comportement de tous les composants du logiciel.

Nous allons également avoir un certain nombre d'activités:

- représenter la vue extérieure du comportement des objets ou des classes correspondants,
- représenter la vue interne du comportement des objets correspondants,
- ajouter les descriptions des composants spécifiques au langage choisi et qui sont nécessaires pour la représentation globale.

Pour démontrer la correction du comportement dynamique de l'architecture du système, il suffit de montrer que celui des composants logiciels l'est. Cette démonstration nécessite l'analyse ou l'exécution de ceux-ci pour vérifier s'ils se comportent de la même manière que l'objet correspondant.

3.3.2.4. Spécification des attributs

L'activité de spécification des attributs identifie ceux propres à chaque composant logiciel, et démontre qu'ils correspondent aux attributs des objets correspondants. Les composants doivent être analysés ou exécutés pour vérifier qu'ils remplissent bien les conditions sur les attributs requis.

3.3.3. Résultats de la conception

Les résultats sont, d'une part une description du comment implémenter une solution au problème et, d'autre part, une représentation graphique et textuelle de l'architecture logicielle du problème. De plus, un dictionnaire de l'architecture logicielle va permettre de synthétiser et de mettre en relation les différents modèles.

3.4. Conclusions

L'Object-Oriented Software Development Method (OOSD) se distingue de l'analyse structurée en développant un modèle conceptuel simple et unique pour tous les éléments du modèle. Celui-ci va, durant la phase d'analyse, identifier le Quoi et, durant la phase de conception, déterminer le Comment.

Pendant la phase d'analyse, l'OOSD va développer un modèle qui va identifier les objets, classes, fonctions, comportement et propriétés (attributs) du problème. Durant la conception, le modèle est affiné en une architecture de composants logiciels possédant une transition aisée vers le codage.

L'identification des objets est l'étape la plus difficile de toute méthode orientée objets, mais Colbert estime que la définition et l'utilisation qui est faite de l'objet dans cette méthode est relativement abordable. Actuellement, l'identification ne peut pas encore être automatisée car elle requiert trop d'intelligence de la part du concepteur. Par contre, bon nombre d'étapes suivantes peuvent être en partie ou complètement automatisées.

Selon certains utilisateurs, cette méthode s'applique relativement bien aux problèmes du monde réel.

A1.4. La méthode HOOD

La méthode HOOD (Hiérarchical Object Oriented Design) est une méthode développée par CISI-Ingenierie, CRI et Matra-Espace pour l'European Space Agency. Cette méthode est intéressante dans le sens où elle décompose, de manière hiérarchisée, le domaine du problème en identifiant des objets et des opérations. La source de documentation est le manuel de référence de la méthode [HOO91].

4.1. Les concepts de base et les processus de conception

HOOD est, selon les auteurs, une union entre les machines abstraites [MAC85] et la conception orientée objets de Booch [BOO86]. Cette méthode s'appuie donc sur deux types de hiérarchisation importante:

- *la hiérarchie de supériorité*, où les objets se trouvant en haut de la hiérarchie contrôlent et utilisent les objets se trouvant en dessous,
- *la hiérarchie parent-enfant*, qui permet à un objet d'être la composition d'autres objets.

Les objets sont donc des entités du monde réel qui combinent à la fois des données et des opérations. Ils peuvent être identifiés comme les Packages dans ADA. La conception orientée objets est une technique qui utilise les objets comme unité de base de la modularité dans la conception de systèmes. HOOD renforce la structure existant entre ces objets selon un certain nombre de principes:

- *principes d'abstraction, de masquage d'information et d'encapsulation.* Un objet est défini par les services qu'il propose à ses utilisateurs, par les services qu'il requiert auprès des autres objets et par son comportement alors que la structure est cachée à l'utilisateur. Les services sont définis dans la structure de contrôle d'opérations (OPCS) et le comportement de l'objet est défini dans la structure de contrôle de l'objet (OBCS).
- *principes hiérarchiques*:
  - les objets peuvent utiliser des opérations d'autres objets. Ce système peut donc être représenté comme des objets parents utilisant des objets enfants au sein d'une hiérarchie.
  - les objets peuvent être décomposés en objets. Ce système peut être représenté comme objet parent incluant des objets enfants.

- *principes de contrôle* : les opérations sont activées à travers des flux de contrôle qui correspondent à l'exécution de processus logiques sur la machine cible. Il peut y avoir un certain nombre de flux de contrôle opérant simultanément dans un objet.

La conception d'un système peut être modélisée selon trois vues différentes:

- une *vue logique*, qui consiste en un espace de conception comprenant un ensemble d'arbres de conception (Design Trees) structurés par des objets et qui dépend d'une bibliothèque commune d'éléments pour les objets (l'espace de classe). Chacun des arbres de conception peut être considéré comme un contexte ou un environnement pour les autres,
- une *vue de distribution*, qui traite de l'identification des unités de distribution.
- une *vue physique*, qui permet d'attribuer les unités de distribution aux noeuds physiques.

HOOD s'occupe de la vue logique et de la vue de distribution.

4.2. Les objets et les opérations dans HOOD

HOOD propose un mécanisme qui permet de dépasser les limitations de l'OOD dans les grands systèmes complexes: définition d'un processus de conception par étapes Top-Down, de procédures de vérification et de validation qui ont conduit à une formalisation du concept objet. Cette formalisation est basée sur une notation graphique et textuelle, soutenue par le Squelette de Description de l'Objet (ODS).

4.2.1. Propriétés statiques

Un objet a une partie visible, l'interface, et une partie cachée, interne, qui n'est pas accessible directement. Un objet n'est accessible à partir du monde réel que par son nom. Une représentation graphique se trouve en figure A1.4-1.

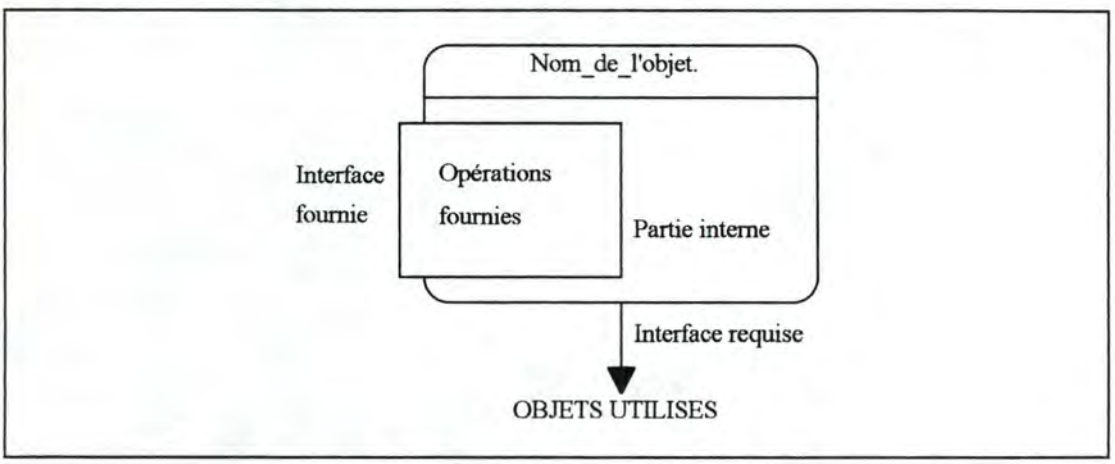


Figure A1.4-1: Représentation graphique d'un objet.

L'interface définit les types, les constantes, les opérations et les exceptions fournies par l'objet et les paramètres associés ainsi que leur type.

La partie interne est l'implémentation de l'interface fournie. Cette partie comprend des types, des constantes, des opérations et des exceptions, de même que des objets.

4.2.2. Propriétés dynamiques

La communication entre un client et un serveur se réalise par l'exécution d'une opération se trouvant dans l'objet serveur. Différents types de flux de contrôle existent:

- *séquentiel*, le contrôle est directement transféré à l'opération requise. Lorsque l'opération est terminée, le contrôle est rendu au client.
- *parallèle*, le contrôle n'est pas transféré, mais la structure de contrôle de l'objet (OBCS) reçoit la demande d'exécution. La réaction à cette demande va dépendre de l'état dans lequel se trouve le serveur. Elle sera exécutée lorsque le serveur se trouvera dans l'état défini dans l'OBCS.

Une opération peut être contrainte dans son exécution par :

- *l'état interne de l'objet.* La sémantique d'une opération peut dépendre d'un ensemble de conditions logiques internes. Ces opérations sont déclenchées par des "Trigger Arrows" sans aucune annotation.
- *le type d'exécution demandée.* Le type est indiqué sur le "Trigger Arrow". Nous avons essentiellement quatre types possibles:
  - requête d'exécution fortement synchronisée (REFoS): le client attend que l'opération soit terminée avant de continuer son flux de contrôle,
  - requête d'exécution faiblement synchronisée (REFaS): le flux de contrôle du client est suspendu jusqu'à ce que la requête ait été acceptée par l'objet serveur,
  - requête d'exécution asynchrone (REA): le flux de contrôle du client n'est pas affecté, mais l'opération est déclenchée au sein du serveur par un message,
  - requête d'exécution temporisée (RET): l'opération requise par le client doit s'exécuter ou être acceptée en un temps donné. Le serveur doit prévenir le client, lorsque le temps est écoulé, de l'état d'avancement de la requête. Cette requête est combinée avec les deux premières, les requêtes synchrones.

Il existe également deux types d'objets:

- les *objets passifs*: objets qui n'ont pas de sémantique associée à leur flux de contrôle. Lorsqu'une opération de cet objet est exécutée, elle n'interagit jamais avec une autre,

- les *objets actifs*: le flux de contrôle des opérations contraintes est géré par l'OBCS. Les opérations qui ne sont pas contraintes, sont exécutées séquentiellement comme pour les actions des objets passifs.

La figure A1.4-2 donne une représentation de ces deux types d'objets.

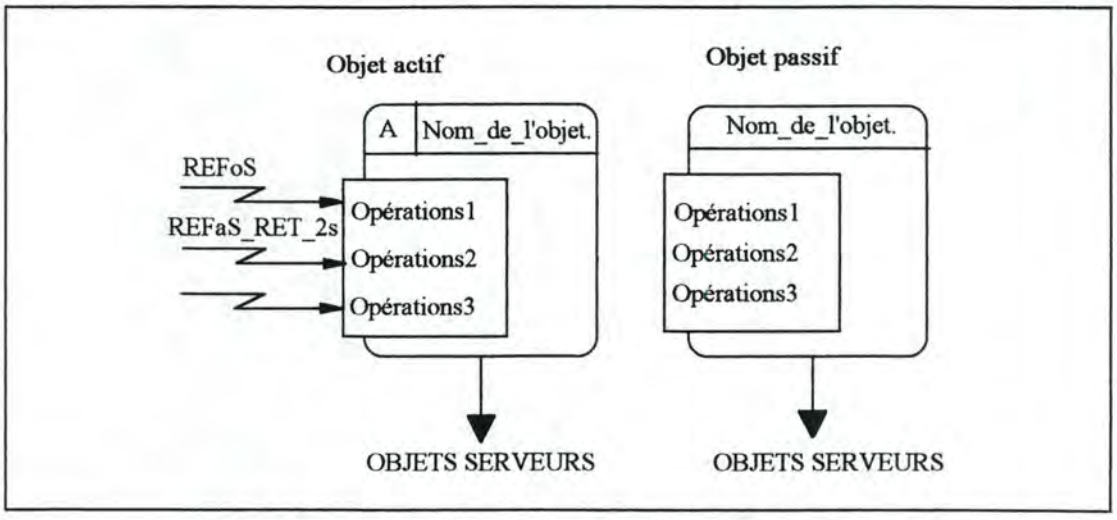


Figure A1.4-2: Représentation graphique des objets actifs et passifs.

4.3. Les relations entre les objets

Un objet en utilise un autre si le premier requiert une ou plusieurs opérations fournies par le second. HOOD autorise que les objets actifs utilisent n'importe quel type d'objets, mais il interdit les cycles dans ces relations : A utilise B, B utilise C, mais C ne peut utiliser ni A, ni B.

Cette relation d'utilisation définit la relation Client-Serveur entre les objets. Pour cela, HOOD préconise l'utilisation d'un graphe d'interconnexion entre les différents objets. Ce graphe doit être acyclique et de complexité la plus faible possible, c'est-à-dire que les objets devraient utiliser le moins d'objets possibles.

4.4. Les relations d'inclusion entre les objets

Pour fournir une approche Top-Down, un objet parent est décomposé en objets fils qui proposent collectivement la même fonctionnalité que le parent. Ce processus de décomposition est basé sur la relation d'inclusion entre objets. Nous aurons donc que :

- la relation d'inclusion est représentée par l'inclusion du dessin de l'objet fils au sein de l'objet parent,
- la correspondance entre les opérations de l'objet parent et les opérations des objets fils est représentée par des flèches pointillées,
- un objet au niveau du parent qui est utilisé par un objet parent doit aussi être utilisé par au moins un des objets fils. Cet objet, appelé "Oncle" et représenté par une boîte annotée pour préciser si l'objet est actif

ou passif, est attaché au bord de l'objet parent. La boîte définit la connexion entre l'objet parent et l'interface. La flèche "utilise" est dessinée entre l'objet fils et l'objet oncle.

4.4.1. Le processus de conception dans HOOD

Le processus de conception de HOOD est basé sur le principe de décomposition successive de l'objet HOOD représentant le système à conceptualiser. Celui-ci sera la racine d'une hiérarchie parent-enfant. Cette hiérarchie représente l'arbre de conception HOOD. Cet arbre possédera des objets intermédiaires et terminaux.

Une décomposition est raisonnablement constituée de 2 à 6 niveaux de décomposition.

4.4.2. Les décompositions d'opérations

Pour les objets terminaux, la structure de contrôle des opérations contient une description complète incluant le pseudo-code. Ces objets peuvent utiliser des opérations internes qui n'apparaîtront pas dans le diagramme, mais qui seront décrites dans la partie interne de l'objet terminal.

Les opérations d'un objet parent sont implémentées par au moins une opération des objets fils. Dans la description de ces opérations, nous indiquerons que celles-ci sont "implémentées par" suivi du nom de l'opération de l'objet fils. Si une opération en requiert plusieurs, nous utiliserons un objet terminal spécial "contrôle d'opération" qui gérera les différentes opérations nécessaires.

Les opérations non contraintes des objets parents sont implémentées par des opérations contraintes d'objets fils à travers un objet de contrôle d'opération. De manière similaire, les opérations contraintes des objets parents le sont par des opérations non contraintes, toujours sous le contrôle d'un objet de contrôle d'opération.

Un "Operation_set" correspond à un ensemble d'opérations qui seront implémentées à un niveau ultérieur de décomposition. Un operation_set peut lui-même être décomposé en plusieurs operation_sets et en opérations.

La figure suivante montre la plupart des éléments que nous venons de décrire.

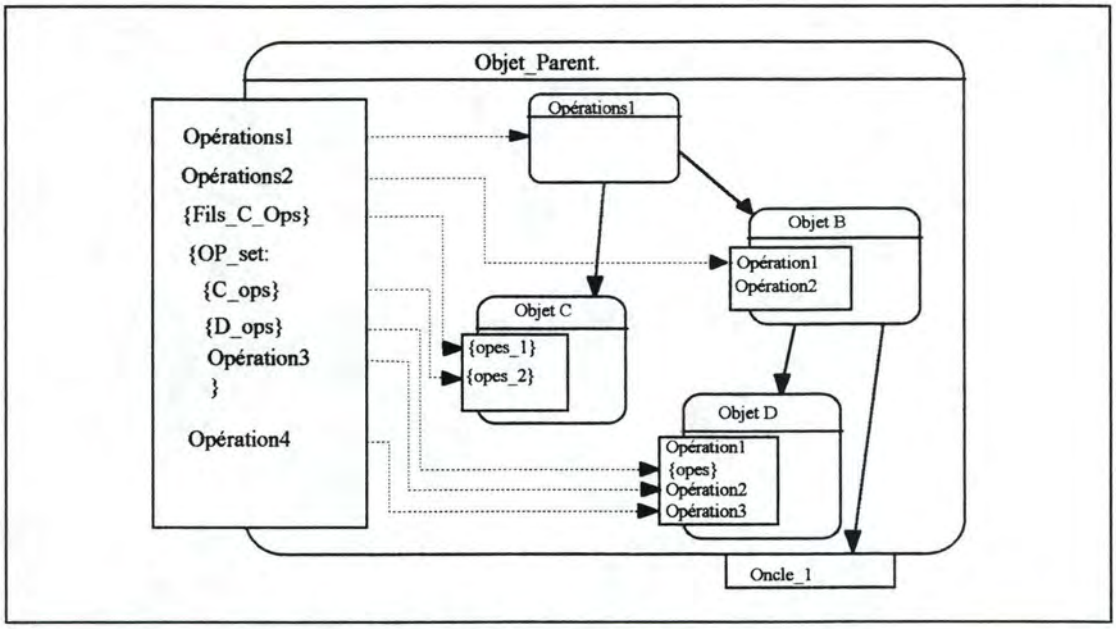


Figure A1.4-3: Décomposition des opérations.

4.4.3. La décomposition de la structure de contrôle d'un objet

Les interactions des flux de contrôle au niveau parent sont décrites par une structure de contrôle de l'objet et peuvent être décomposées selon deux cas:

- l'OBCS est conduit par un objet spécialement créé pour cela. Cet objet reçoit tous les flux venant de l'interface et les redirige vers les autres objets fils internes, voir la figure A1.4-4,
- l'OBCS est conduit par plusieurs objets fils, comme dans la figure A1.4-3.

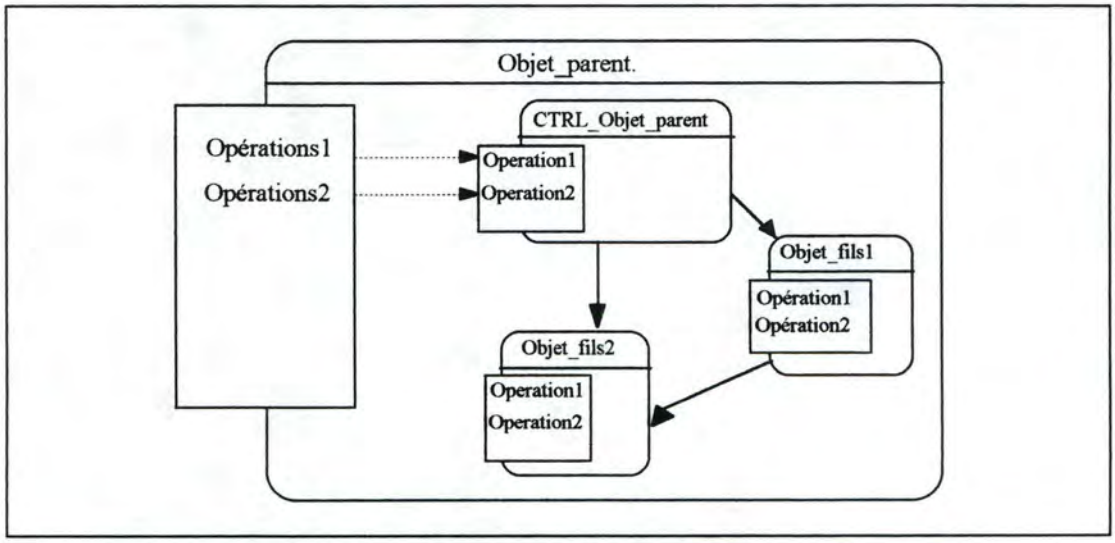


Figure A1.4-4 : L'OBCS géré par un objet spécialisé.

4.5. Les flux de données

Pour montrer les flux de données, une petite flèche avec un cercle qui portera le nom de la donnée sera placée sur ces relations. Ces flèches indiqueront la direction dans laquelle la donnée circule. Un exemple est donné à la figure A1.4-5.

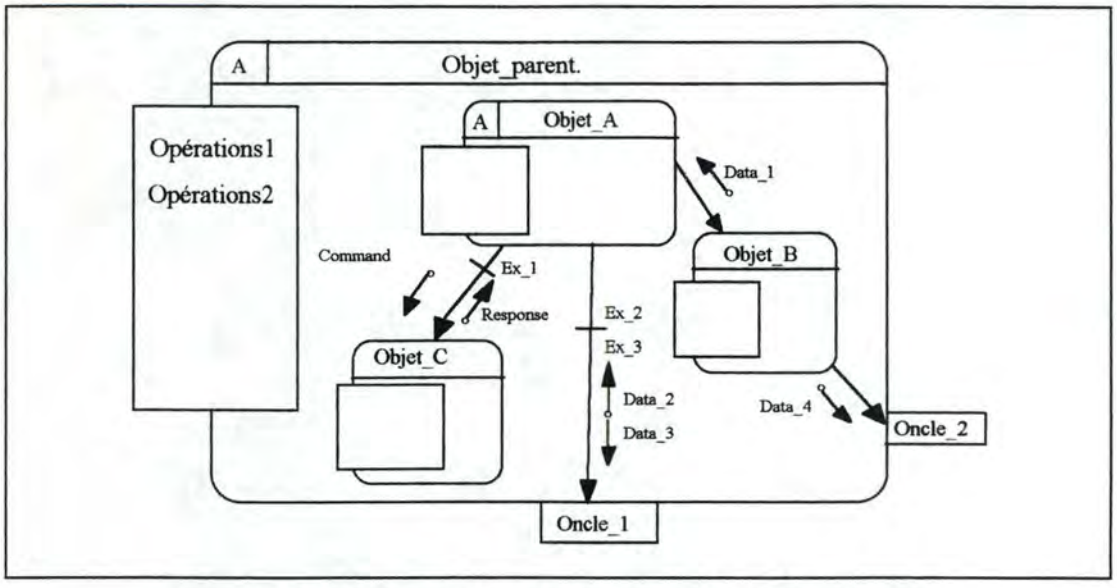


Figure A1.4-5 : Représentation des flux de données et des exceptions.

4.6. Les flux d'exceptions

Une *exception* est le retour anormal du flux de contrôle pendant l'exécution d'une opération. Cette exception est représentée par une ligne portant le nom de l'exception et qui croise la relation. Une exception se propage le long des relations depuis l'opération où elle a été rencontrée jusqu'à l'opération de l'objet client qui va alors exécuter le code de récupération. Voir figure A1.4-5.

4.7. L'objet environnement

Un *objet d'environnement* est un objet racine qui ne fait pas partie de l'arbre de conception HOOD, mais qui est utilisé par le système en cours de conceptualisation. Cet environnement permet l'incorporation d'autres logiciels dans la conception de HOOD, sans affecter les principes de décomposition de HOOD.

Un objet environnement est toujours représenté comme un objet oncle avec la lettre "E" dans sa partie gauche. Cet environnement fait l'objet d'un autre arbre de conception HOOD, où les objets du système étudié sont indiqués en tant qu'objets d'environnement.

4.8. La classe

Une *classe* est une représentation d'un modèle, d'un plan de l'objet, qui peut être réutilisée et paramétrée par des types, des constantes, des opérations ou des ensembles d'opérations.

4.8.1. Définition de la classe

Une classe peut être spécialement conçue lorsqu'un certain nombre d'objets similaires est nécessaire. Les classes ont comme paramètres des types, des constantes, des opérations ou des ensembles d'opérations et peuvent être "actives" ou "passives". Une classe est une racine d'un modèle HOOD et n'utilise que des objets d'environnement.

Les classes utilisent des paramètres formels, représentés comme un objet oncle et portant la lettre "F" comme identifiant. La figure A1.4-6 montre la représentation graphique d'une classe.

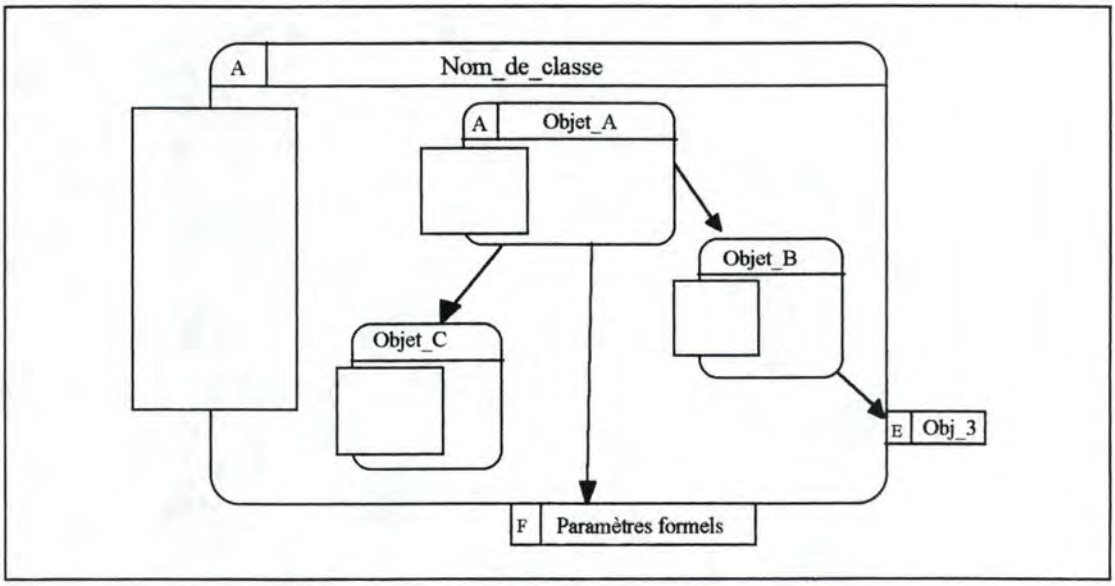


Figure A1.4-6 : Représentation d'une classe.

4.8.2. Définition d'une instance

Les *instances* d'une classe peuvent être créées lorsque les différents paramètres sont définis explicitement. Certains tests doivent être faits pour vérifier la correspondance entre les définitions formelles des paramètres de la classe et les paramètres actuels. De plus, ces paramètres doivent être visibles pour les objets internes du parent où ils sont instanciés. Les opérations d'un objet du modèle étudié ou d'un objet d'environnement peuvent être vues comme des paramètres.

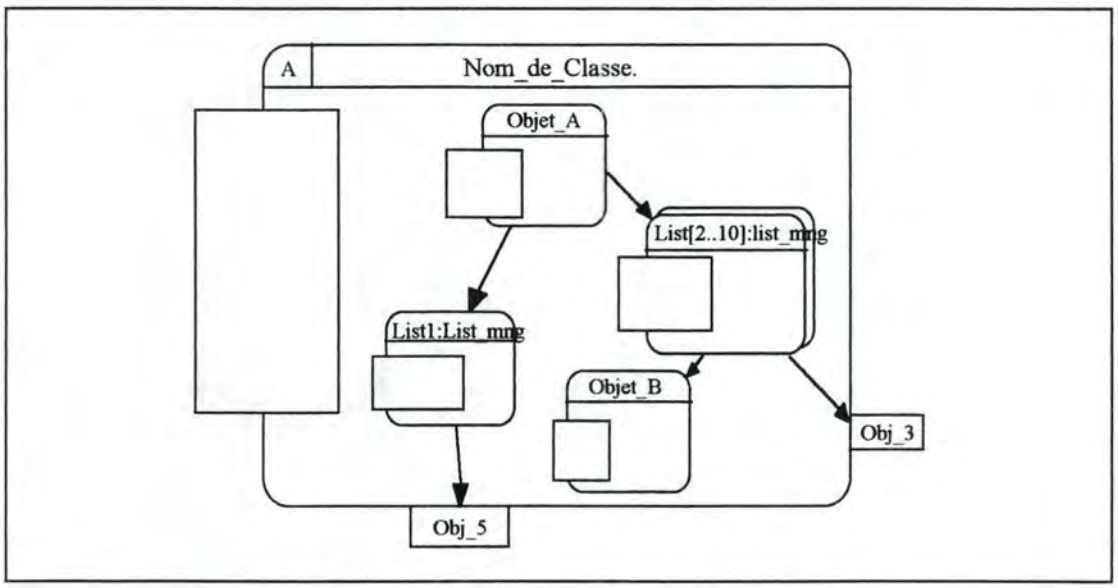


Figure A1.4-7 : Représentation des instances.

A partir de cet arbre de conception, les autres arbres ne sont utilisables que par leur objet racine. Cet ensemble de tous les objets racines, c'est-à-dire tous les arbres de conception HOOD et de toutes les classes, définissent la *configuration du système*, une "super-racine".

4.9. Le noeud virtuel

Le *noeud virtuel* (VN) est un ensemble d'objets HOOD définissant une unité de distribution. Plus tard, cette unité sera placée sur un noeud physique. Les activités de ce noeud consistent en opérations locales et en communications avec les autres noeuds virtuels. Les noeuds virtuels sont définis dans un arbre dont les noeuds terminaux contiennent les objets HOOD définissant la configuration du système.

Le noeud virtuel a un certain nombre de propriétés concernant:

- les relations d'utilisation:
  - un noeud virtuel ne fournit que des opérations contraintes,
  - un noeud virtuel ne peut utiliser que d'autres noeuds virtuels,
  - la relation entre deux noeuds virtuels peut correspondre à des protocoles de communication très divers;
- les relations d'inclusion:
  - un noeud virtuel ne peut être décomposé qu'en noeuds virtuels;
- les relations d'allocations:
  - un noeud virtuel terminal est défini comme étant la composition d'objets alloués à ce noeud,
  - un objet HOOD peut être dupliqué dans plusieurs noeuds virtuels terminaux.

Les noeuds virtuels sont représentés comme des objets. Ils portent un "V" à gauche du nom du noeud, comme pour les objets actifs.

4.10. La formalisation textuelle

L'utilisation d'une notation formalisée permet de supprimer des ambiguïtés, de faciliter l'échange d'informations et de faciliter l'étape finale qu'est l'implémentation.

Cette formalisation n'est pas un processus figé mais est continuellement remise à jour au cours du développement du modèle et de l'identification de nouveaux éléments, relations, etc.

Il apparaît donc:

- un squelette de description d'objet (ODS) pour chaque objet, classe, instance, racine identifié, qu'il soit de l'application ou de l'environnement,
- une description pour chaque type de données et pour chaque constante et données identifiées,
- une description pour chaque ensemble d'opérations,
- une description des structures de contrôle d'objet (OBCS),
- une description des structures de contrôle des opérations (OPCS).

4.11. La vérification de la conception

HOOD définit une série de règles qui permettent de tester la consistance entre les différentes étapes de décomposition.

4.11.1. La visibilité

Les premières règles concernent les interfaces et les possibilités d'accès aux informations. Celles-ci vérifient donc les principes d'encapsulation et de masquage d'information.

4.11.2. Les règles de conception de HOOD

Des règles formelles peuvent être définies pour vérifier la consistance et la complétude des descriptions de conception. Elles peuvent être classées en différentes catégories:

- règles de définitions générales qui donnent les définitions de base et les propriétés des types d'objets HOOD,
- règles des relations d'utilisation qui définissent la manière dont un objet peut être utilisé par un autre,
- règles des relations d'inclusion qui donnent la manière dont un objet parent donné est implémenté par un ensemble d'objets fils. Des règles de décomposition d'objets sont également placées dans cette catégorie,
- règles d'opérations qui donnent les définitions de base et les propriétés des opérations HOOD,

- règles de visibilité et de dénomination pour les objets et les opérations,
- règles de vérification de la consistance des descriptions des objets fils par rapport à la description de l'objet parent,
- règles de consistance interne qui vérifient la consistance interne aux entités HOOD au sein de leur ODS et entre l'ODS et le diagramme correspondant,
- règles des noeuds virtuels qui donnent la définition et les propriétés des noeuds virtuels.

A1.5.

La méthode de SHLAER et MELLOR

Cette méthode a été développée par Sally SHLAER et Stephen J. MELLOR. dans le livre *Object-Oriented Systems Analysis, Modeling the World in Data* [SHL88].

5.1. Pourquoi modéliser l'information?

Suite aux nombreuses difficultés rencontrées lors du développement de logiciels, la modélisation de l'information peut apporter une solution simple et efficace à ces problèmes.

Un *modèle d'information* (Information Model) consiste en une organisation et une notation graphique adaptée pour décrire et définir à la fois le vocabulaire et la conceptualisation du domaine du problème. Le modèle identifie, classifie et permet une certaine abstraction de ce qui est dans le domaine du problème. Il organise l'information au sein d'une structure formalisée.

Les auteurs recommandent de commencer un projet de développement de logiciel par la construction du modèle d'information de l'application. Ensuite, la phase d'analyse se déroulera beaucoup plus facilement étant donné que le problème sera parfaitement connu et spécifié.

Voyons comment construire ce modèle.

5.2. Les objets

Nous allons d'abord déterminer les objets se trouvant dans le domaine de l'application. Un objet est une abstraction d'un ensemble de "choses" du monde réel telles que:

- toutes les "choses" du monde réel de cet ensemble, les instances, ont les mêmes caractéristiques,
- toutes les instances subissent et respectent les mêmes règles.

5.2.1. Identification des objets

L'identification des objets est relativement aisée. La plupart des "choses" peuvent se placer dans les cinq classes suivantes:

- *les "choses" tangibles*,
- *les rôles* joués par des personnes ou des organisations,
- *les incidents*: les objets incidents sont utilisés pour représenter une occurrence ou un événement,
- *les interactions*: elles ont généralement un aspect de "transaction" ou de "contrat" et relient plusieurs objets du modèle,
- *les spécifications*: elles montrent une standardisation ou une définition. De plus, nous avons généralement, un autre objet qui représente les instances des objets qui vérifient les spécifications. Ces "objets d'instance" peuvent représenter quelque chose qui ne soit pas tangible.

5.2.2. Description et dénomination des objets

La description d'un objet est un petit texte qui précise si l'objet est une instance d'un objet qui a déjà été conceptualisé dans le modèle. Chaque objet du modèle doit posséder une description.

Cette description doit suivre une série de règles portant sur:

- la *base de l'abstraction* :
  - si la base de l'abstraction est le *critère d'inclusion,* la description de la classe d'une chose du monde réel insiste sur les aspects communs des instances de cet objet,
  - si la base de l'abstraction est le *critère d'exclusion*, celui-ci est établi s'il y a des instances qui ne sont pas des abstractions de cet objet,
- *le contexte de l'objet.* La relation liant l'objet aux autres objets du modèle est expliquée,
- les *informations générales.* Un certain nombre d'informations générales amplifient la description de l'objet,
- la *déclaration du monde réel vs. déclaration du modèle.* Lors de la description des objets, une déclaration à propos de la réalité formalisée est réalisée, de même qu'une déclaration qui tient compte de la construction du modèle lui-même. Une distinction typographique marque la différence entre les deux aspects: la première lettre des noms des objets sera mise en capitale tandis que la première lettre des références à des instances du monde réel est mise en minuscule,
- les *standards d'écriture technique.* L'utilisation de ces standards facilite et unifie les descriptions des objets,

Un bon choix des noms des objets peut contribuer de manière significative à la lisibilité et à la compréhension du modèle. Voici quelques conseils pour réaliser cette dénomination:

- utiliser des noms communs lorsque ceux-ci peuvent être bien définis,
- utiliser des noms de tous les jours avec une sémantique étendue de préférence à des noms vagues, techniquement inutiles,
- utiliser le même nombre d'éléments si le nom est composé,
- ajouter des adjectifs aux noms communs courts pour rendre ces noms plus précis,
- utiliser des noms faits sur mesure s'ils montrent l'aspect essentiel de l'objet,
- nommer l'objet par les informations qu'il contient et non par la forme généralement utilisée pour porter l'information. Ainsi on écrira: *Conducteur* au lieu de *Permis de conduire*.
- éviter les mots qui ont un grand nombre de significations ou qui sont trop dépendants du contexte.

5.2.3. Test des objets

Lors de l'identification des objets de l'application, il nous arrive de relever un certain nombre d'éléments que nous pensons être des objets alors qu'ils ne le sont pas. Les tests suivants vont nous permettre de rejeter ces faux objets:

- le *Test d'uniformité* : ce test est basé sur la définition de l'objet. Chaque instance doit avoir le même ensemble de caractéristiques et être le sujet des mêmes règles,
- le *Test de Plus-que-le-Nom* : ce test vérifie que l'objet possède quelque chose en plus que son nom comme caractéristique. C'est-à-dire qu'il a au moins deux attributs,
- le *Test du OU* : si, dans la description de l'objet, le critère d'inclusion utilise le "OU" d'une manière significative, ce n'est probablement pas un objet mais un conglomérat. Il faut donc séparer les différents objets qui ont été ainsi assemblés,
- le *Test de Plus-qu'une-Liste* : si le critère d'inclusion ne donne dans la description de l'objet qu'une liste des instances spécifiques, ce n'est probablement pas un objet. Ce test indique généralement une abstraction insuffisante de l'objet par rapport à ses instances.

5.3. Les attributs

5.3.1. Définition et notation

Un *attribut* est l'abstraction d'une caractéristique élémentaire possédée par toutes les entités qui sont elles-mêmes vues comme des objets. Le but est donc d'obtenir des ensembles d'attributs qui soient **complets**, qu'ils contiennent toute l'information pertinente pour l'objet défini, **partitionnés**, chaque attribut contient un aspect séparé de l'objet, et **mutuellement indépendants**, les valeurs des attributs sont indépendantes les unes des autres

Il existe diverses manières de représenter graphiquement les objets avec leurs attributs. La représentation suivante sera utilisée:

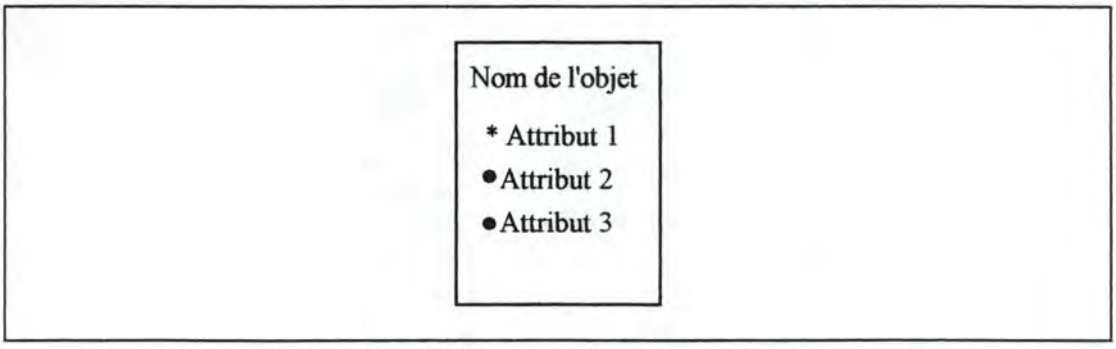


Figure A1.5-1 : Représentation graphique des objets.

5.3.2. Identification, description et classification les attributs

Les attributs peuvent être identifiés en se référant:

- aux instances du monde réel qui sont devenues des objets, c'est-à-dire le type auquel appartiennent les caractéristiques que possèdent *toutes* les instances.
- à la description de l'objet, c'est-à-dire les informations des entités du monde réel devant être connues pour pouvoir dire si celles-ci sont des instances de cet objet.

Chaque objet possède un identifiant. Cet identifiant est un ensemble d'un ou de plusieurs attributs qui permettent de distinguer de manière univoque les instances d'un même objet. Une ou plusieurs astérisques dans la représentation graphique de l'objet indiqueront cette propriété. Voir la figure A1.5-1.

La description de l'attribut montre comment l'attribut formalisé représente la caractéristique du monde réel. Pour chacun de ces attributs, le domaine des valeurs que celui-ci peut prendre est également donné.

Les attributs ainsi identifiés peuvent entrer dans trois catégories différentes:

- *les attributs descriptifs.* Ces attributs fournissent des faits spécifiques à chaque instance de l'objet. La description des attributs descriptifs montre quelle caractéristique du monde réel, possédée par toutes les instances de l'objet, est représentée dans le modèle par cet attribut.
- *les attributs de dénomination.* Ces attributs fournissent des faits sur les étiquettes arbitraires et sur les noms portés par chaque instance de cet objet. La description des attributs de dénomination spécifie la forme du nom, l'organisation qui assigne, enregistre ou contrôle les noms et étend la description vers les attributs qui complètent l'identifiant.
- *les attributs de référence.* Ces attributs fournissent des faits qui indiquent un lien entre une instance d'un objet et une instance d'un autre objet.

La description des attributs référentiels spécifie la relation qui est représentée par cet attribut.

5.4. Les relations

5.4.1. Le concept de relation

Une *relation* est l'abstraction d'un ensemble d'associations qui sont utilisées systématiquement entre différents types d'éléments dans le monde réel. Les relations sont donc décrites en termes d'objets formels modélisant les entités du monde réel qui participent à l'association.

5.4.2. Les relations binaires

Les relations faisant intervenir plus de deux objets seront décrites au point suivant.

Les relations ne faisant intervenir que deux objets peuvent être classifiées en trois types fondamentaux, selon le nombre d'instances de ces objets qui participent à chaque instance de la relation. Nous avons donc les "une-à-une" (1:1), les "une-à-plusieurs" (1:M) et les "plusieurs-à-plusieurs" (M:M). Ces termes indiquent la cardinalité de cette relation. Nous les représenterons différemment comme le montre la figure A1.5-2.

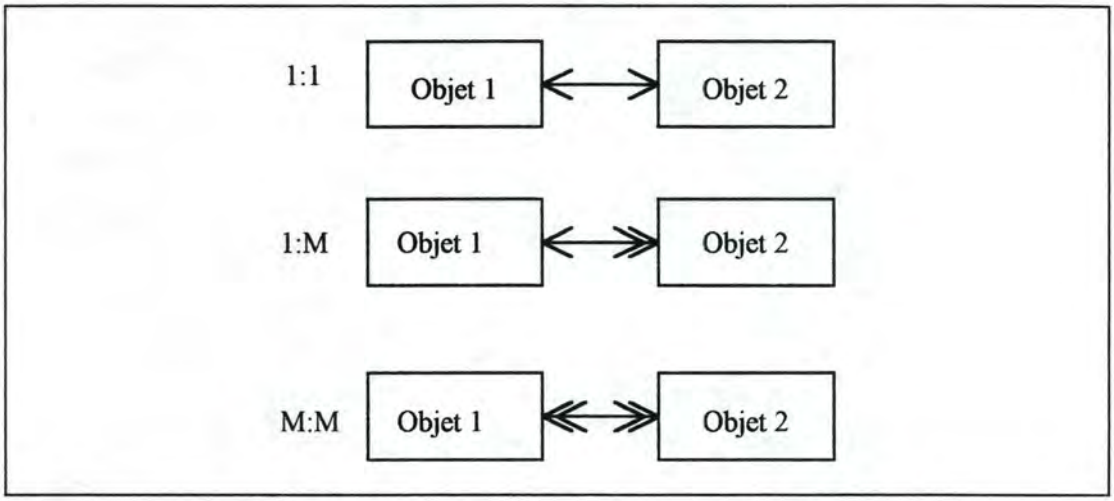


Figure A1.5-2: Les différentes représentations des relations.

5.4.3. Les relations inconditionnelles

La relation (1:1). Chaque instance d'un objet de type A est associée à une et seulement une instance d'un objet de type B. De plus, chaque instance d'un objet de type B doit avoir une instance d'un objet de type A qui lui est associée. Pour modéliser une telle relation, un identifiant d'un des objets sera placé comme attribut de référence dans l'autre objet. Cet attribut sera appelé une "clé étrangère" et sera indiqué par un (R) ajouté au nom de cet attribut. Son domaine de valeur sera donc "Comme *nom de l'attribut et de l'objet d'où il vient*".

La relation (1:M). Chaque instance d'un objet de type A est associée à une ou plusieurs instances d'un objet de type B. De plus, chaque instance d'un objet de type B est associée à une et une seule instance d'un objet de type A. Pour modéliser une telle relation, un identifiant de l'objet dont plusieurs instances participent à la relation sera placé comme attribut de référence dans l'autre objet. Cet attribut sera appelé une "clé étrangère" et sera indiqué par un (R) ajouté au nom de cet attribut. Son domaine de valeur sera également "Comme *nom de l'attribut et de l'objet d'où il vient*".

La relation (M:M). Chaque instance d'un objet de type A est associée à une ou plusieurs instances d'un objet de type B, et chaque instance d'un objet de type B est associée à une ou plusieurs instances d'un objet de type A. La modélisation de telles relations est un peu plus complexe. En effet, les identifiants des deux objets seront utilisés pour construire une table de corrélation. Cette table est nécessaire pour enregistrer la relation, ou corrélation, existant entre deux objets du modèle. La table est représentée à la figure A1.5-3.

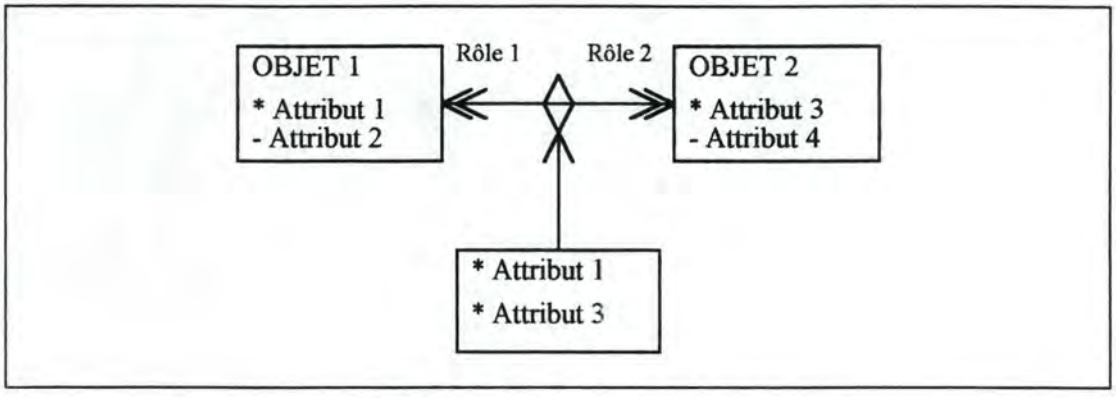


Figure A1.5-3 : Table de corrélation.

5.4.4. Les relations conditionnelles

Les relations conditionnelles sont dérivées des formes inconditionnelles. La seule différence est que certaines instances peuvent ne pas prendre part à ces relations. Leur participation dépend d'une condition.

Il peut y avoir deux types de relations conditionnelles: la relation conditionnelle et la relation bi-conditionnelle possédant une condition de part et autre de la relation. Voici les principales modélisations:

- les relations 1:1c sont modélisées en ajoutant un attribut référentiel à l'objet qui participe toujours à la relation,
- les relations 1c:1c sont modélisées de deux manières différentes: d'une part, *la clé étrangère,* où l'on ajoute un attribut, la clé, à l'autre objet comme dans une relation inconditionnelle 1:1 et de plus cet attribut pourra prendre une valeur spéciale précisant que l'objet n'est pas en relation, d'autre part *la table de corrélation* qui ne reprend dans la table que les instances qui participent à la relation,

- les relations 1c:M ont les propriétés suivantes: chaque instance de type A participe à la relation et est associée à un ou plusieurs objets de type B. Une instance de type B n'est en relation avec aucun ou avec un seul objet de type A, donc toutes les instances de B ne participent pas à la relation. Elles sont modélisées en ajoutant un attribut à l'objet se trouvant du coté "M" de la relation. Cet attribut aura parfois la valeur "nulle" indiquant qu'il ne participe pas à la relation,
- les relations 1:Mc sont caractérisées par le fait que chaque instance de l'objet de type A est associée à aucune ou plusieurs instances de type B tandis que chaque instance de type B est associée à exactement une instance de type A. L'ajout d'un attribut dans l'objet se trouvant du côté "Mc" permet la modélisation de la relation,
- les relations 1c:Mc sont des relations où des instances de chacun des objets peuvent ne pas participer à la relation. L'ajout d'un attribut permet également de modéliser ces relations. Cet attribut peut prendre la valeur "nulle" si l'instance n'y participe pas,
- les relations M:Mc fonctionnant comme les relations inconditionnelles M:M si ce n'est que des instances d'un des objets ne participant pas à la relation peuvent exister. Une table de corrélation sera construite pour garder les instances des objets qui participent à la relation,
- les relations Mc:Mc sont similaires aux relations M:M inconditionnelles si ce n'est que des instances de chacun des deux objets peuvent ne pas participer à la relation. Une table de corrélation sera également utilisée pour modéliser cette relation.

5.5. La structuration des objets

5.5.1. Les sous-types et les super-types

Certains objets ont des éléments communs: attributs et comportements. Tous les attributs communs à un sous-ensemble d'objets sont pris et appelés Sous-Types, ils sont utilisés comme base d'un nouvel objet, appelé Super-Type.

La relation entre les objets sous-types et leur super-type est noté sur le dessin par "Is a". Comme indiqué à la figure A1.5-4.

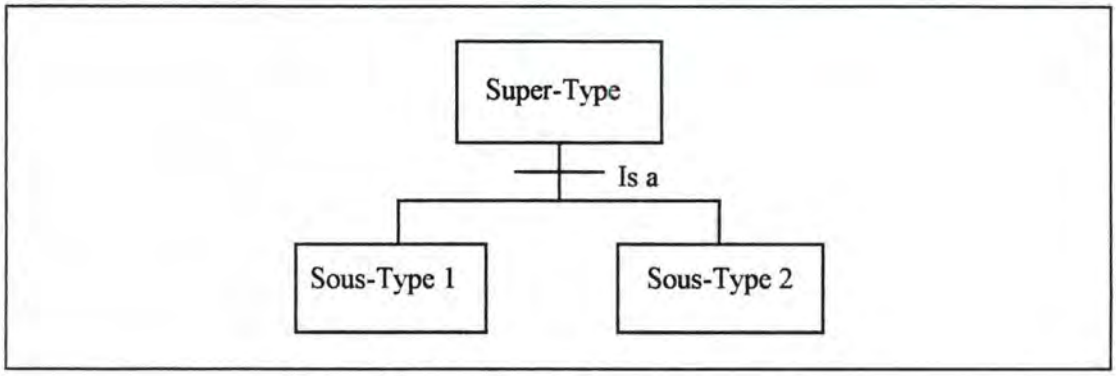


Figure A1.5-4 : Relation super-type et sous-type.

5.5.2. Les objets associatifs

Les objets associatifs sont des objets qui apparaissent lorsqu'il y a de l'information supplémentaire sur les instances de l'association. La table de corrélation comporte donc des données ne se trouvant pas dans les objets eux-mêmes, ce qui va donner la même représentation que celle de la figure A1.5-3, si ce n'est que de nouveaux attributs vont apparaître pour la partie se rapportant à la relation.

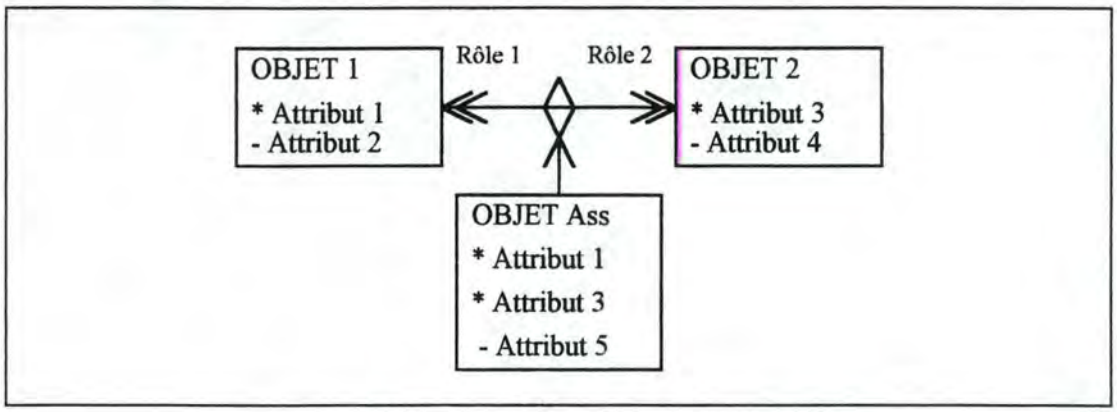


Figure A1.5-5 : Représentation d'un objet associatif.

De plus, une seconde flèche est ajoutée sur le lien entre l'objet associatif et le "diamant" si la cardinalité de la relation est de type M-<rel>. La relation représentée est de type 1-<Rel>. Mais, si le besoin de cet objet associatif dans d'autres relations se fait sentir, il est alors transformé en objet à part entière et le retour à une simple table de corrélation est effectué.

5.6. Représentation du modèle d'information

Nous avons essentiellement un outil graphique et deux outils sous forme de documents pour aider le lecteur à garder une vue cohérente du modèle, c'est-à-dire :

- un diagramme de la structure de l'information,
- un document de spécification des objets,
- un document de spécification des relations.

Ils vont être passés en revue très rapidement. Mais avant cela, il faut souligner qu'un document ne reprenant que les objets avec leurs attributs et les relations avec les objets entrant dans celles-ci ainsi que leurs cardinalités, est généralement produit afin d'obtenir une sorte de feuille de référence rapide.

5.6.1. Le diagramme de la structure de l'information

Ce diagramme est basé sur les différentes formes de schéma entité/association utilisés ces dernières années. Il ne tient compte que de la déclaration des objets, des attributs et des relations du modèle. La représentation graphique a déjà été utilisée dans les figures précédentes du chapitre. Une fois le diagramme terminé, il est intéressant d'en tirer un résumé graphique en ne reprenant que les objets sans leurs

attributs et les relations sans leurs tables de corrélation. Cette vue d'ensemble du diagramme est très utile lors de présentations diverses.

5.6.2. Le document de spécification des objets

Ce document présente l'ensemble du modèle sous forme de textes. Pour chaque objet du modèle, il propose :

- le nom de l'objet,
- une déclaration textuelle de l'objet avec ses attributs, avec indication de l'identifiant,
- une liste des identifiants supplémentaires si nécessaire,
- une description de l'objet,
- une spécification de chaque attribut de l'objet :
    - le nom de l'attribut,
    - la description de l'attribut,
    - les informations diverses sur l'utilité de cet attribut dans le modèle, son rôle dans les relations où l'objet participe si c'est le cas,
    - la déclaration du domaine de l'attribut.

5.6.3. Le document de spécification des relations

Ce document va contenir, pour chaque relation :

- le nom de la relation, du point de vue de chaque objet participant, ainsi qu'une description de la cardinalité et des conditions reliant les différents objets et la relation,
- une description de l'association du monde réel qui est ainsi représentée. Cette description va également justifier la cardinalité et les conditions émises plus haut,
- une description de la manière dont la relation est formalisée par les attributs, c'est-à-dire une mise en relation des attributs appartenant aux différents objets et intervenant dans la relation.

5.7. Rôle du modèle d'information dans le cycle de développement

La modélisation de l'information peut être utilisée lors de la formalisation d'une situation qui exige quelque chose de systématique. Dans le monde de la conception, ce type de modèle est évidement à sa place.

5.7.1. Le processus de développement

Le processus peut être décomposé en quatre étapes:

- l'analyse du problème,
- les spécifications externes,
- la conception du système,

- l'implémentation et l'intégration.

Seules les phases d'analyse et de conception seront détaillées. La phase de spécification externe vise essentiellement à prendre des décisions sur ce que l'application doit faire ou non. La phase d'implémentation et d'intégration transforme les résultats des phases précédentes en un programme correct et accepté par les utilisateurs comme étant ce qu'ils avaient demandé.

5.7.2. La phase d'analyse

La première chose à faire est de construire le modèle d'information afin de déterminer sur quoi le travail va porter. Pour chaque objet, Le cycle de vie peut être déterminé. Chaque objet est créé, a un certain temps de vie, puis est détruit. Ce cycle de vie peut être formalisé sous forme d'un diagramme de transition d'états où les différents états que peut prendre un objet au cours de son cycle de vie sont montrés, ainsi que les différents *événements* provoquant ces changements d'états. De plus, les *actions* sont indiquées. Celles-ci sont associées avec les états à partir desquels elles sont exécutées.

A chaque objet possédant un cycle de vie dans le modèle d'information, un attribut indiquant le statut est ajouté. Il donne l'état de l'objet et son domaine est la liste des états possibles.

Pour les actions qui ont un impact sur les données, un diagramme des flux de données [BOD89] peut être utilisé pour montrer les détails des actions se produisant lors des états. Les flux de données sont représentés par des flèches portant un nom, les processus par des cercles et les accumulateurs par deux lignes parallèles. Ces accumulateurs peuvent êtres vus comme des tables qui sont remplies avec des données qui représentent des instances existantes.

Ces différents modèles sont intégrés de manière à former une phase d'analyse cohérente, comme le montre la figure A1.5-6.

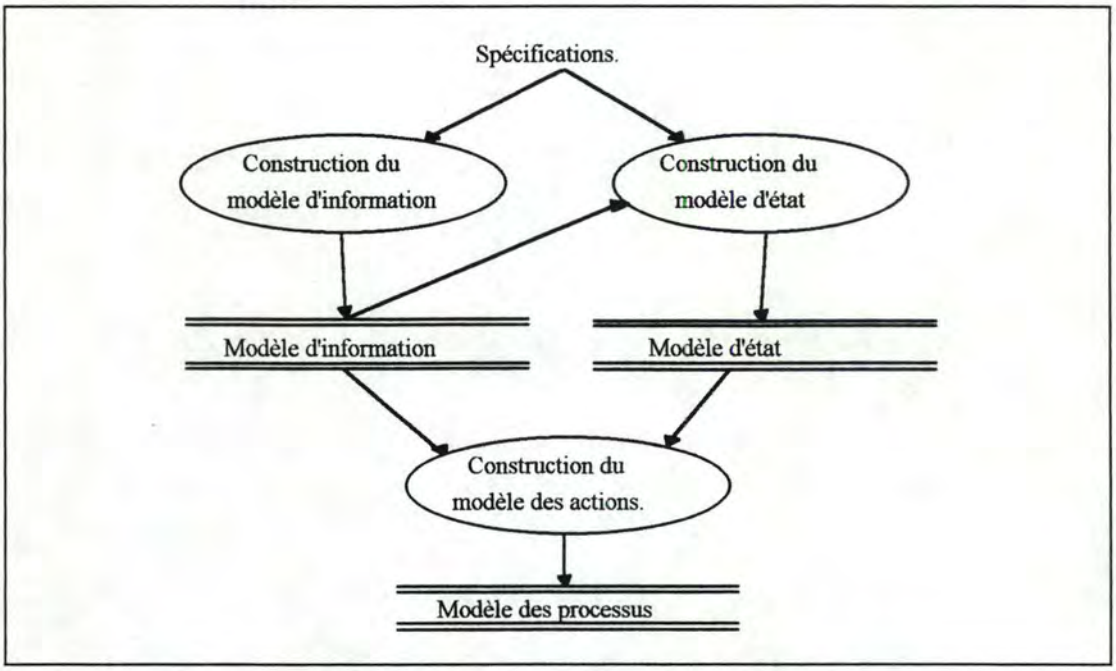


Figure A1.5-7 : Résumé de la phase d'analyse.

5.7.3. La phase de conception

A ce point du développement, une définition claire de ce qui sera mémorisé dans le système et les opérations nécessaires pour agir dessus sont connues.

Dans cette étape, quatre tâches principales apparaissent:

- la conception de l'architecture du logiciel; pour cela, des règles sont définies sur:
  - l'organisation et les accès aux données,
  - la manière dont les opérations vont êtres exécutées,
  - toutes les conventions requises par le programme lui-même, c'est-à-dire les interfaces entre les programmes.
- la conception du contenu du système d'information: les informations qui doivent être mémorisées, disponibles et ce, d'après le diagramme des flux réalisé lors de l'analyse.
- la conception de la structure de données. Cette structure va transformer le résultat de l'étape précédente en structure de données utilisée lors de l'implémentation. Cette structure va tenir compte des moyens disponibles et d'éventuels programmes de gestion de base de données par exemple.
- le partitionnement de l'application en programmes. L'application est divisée en programmes de manière rationnelle. Cette division peut conduire à du code possédant une forte réutilisabilité, du code clair et facile à lire et donc de maintenance facile.

Ces différentes tâches donnent donc un système implémentable.

A1.6. La méthode de WIRFS-BROCK

Cette méthode a été développée par Rebecca **WIRFS-BROCK**, Brian **WILKERSON**, Laureen **WIENER**, dans un livre paru en 1990 : *Designing Object-Oriented Software* [WIR90].

6.1.Le processus de conception

La conception orientée objets est un processus par lequel les spécifications du problème sont transformées en spécifications d'objets.

6.1.1. L'exploration intiale

Dès le départ, le processus de conception est un processus d'exploration et de découverte. Le concepteur cherche les classes, essaye de les organiser selon un certain ordre et définit les différents niveaux d'abstraction du système.

Ce processus est composé d'un certain nombre d'étapes:

- l'identification des classes du système étudié,
- la détermination des opérations, des méthodes pour lesquelles chaque classe est responsable quant à la réalisation et des informations que chacune doit conserver,
- la détermination de la manière dont les objets collaborent entre eux.

Maintenant, nous allons détailler ce processus.

6.2. Les concepts de base de la méthode

Dans ce paragraphe, les différents concepts de base introduits ou redéfinis par les auteurs vont être passés en revue.

6.2.1. Les classes

La première tâche, dans une conception orientée objets est de déterminer les classes qui composent l'application. Une méthode à suivre lors de l'identification de ces classes est proposée. Elle permet d'obtenir un ensemble de règles qui aident à la décomposition de l'application en classes.

6.2.1.1. L'identification des classes

Les spécifications de l'application demandée sont les seuls éléments disponibles au départ. Si elles font défaut, la description des buts de l'application sera effectuée. Un échange entre tous les membres de l'équipe et les utilisateurs permettra de déterminer le plus clairement possible les inputs et les outputs attendus.

Le but recherché est la création de classes d'objets qui peuvent modéliser le domaine de l'application. Pour ce faire, un certain nombre de règles guideront l'identification des classes candidates :

- modéliser les objets physiques.
- modéliser des entités conceptuelles qui forment une abstraction cohérente.
- si plusieurs mots peuvent êtres utilisés pour le même concept, choisir celui qui contient le plus de sens par rapport à l'ensemble de l'application. De cette manière un vocabulaire commun à toute l'équipe travaillant sur le projet est constitué.
- faire attention à l'utilisation des adjectifs. Un adjectif peut indiquer un objet d'un autre type, une utilisation différente de cet objet, où il peut être complètement hors de propos.
- faire attention aux phrases à la voix passive, et à celles où les sujets ne font pas partie du système. Il faut réécrire ces phrases à la voix active car le sujet de cette phrase pourrait être une classe jusque-là non identifiée.
- modéliser les catégories de classes. Ces catégories peuvent devenir plus tard des super-classes abstraites, mais à ce niveau de modélisation, elles sont considérées de manière individuelle.
- modéliser les interfaces connues avec le monde extérieur de l'application.
- modéliser les valeurs des attributs des objets, mais pas les attributs eux-mêmes.

Le résultat de cette procédure est donc de fournir une liste de classes appartenant à l'application. Certaines classes sont manquantes, d'autres seront rejetées plus tard. De même, les différentes parties de l'application ne sont pas détaillées dans la même proportion. En effet, les parties les mieux connues sont étudiées généralement de manière plus approfondie. Pour celles qui le sont moins, l'information manquante est recherchée auprès de gens qualifiés. Mais le processus n'est pas arrêté pour autant puisque cette partie un peu plus difficile peut être vue comme une boite noire, ce qui ce passe à l'intérieur ne doit pas nécessairement être connu. Néanmoins, son interface avec l'application doit pouvoir être décrite.

Lorsque les classes candidates sont identifiées, leurs noms sont écrits sur des fiches. Au dos de ces fiches, est notée une brève description de cette classe, ce qui pourra aider l'écriture de la documentation de celle-ci et expliciter les raisons qui ont conduit à son identification en temps que classe. Ces fiches en papier sont relativement petites et faciles à manipuler. Elles permettent de les organiser et de les réordonner selon les différentes applications ou les différents points avancés.

6.2.1.2. L'identification des classes abstraites

Après la détermination des classes candidates de l'application, les classes abstraites sont recherchées afin d'en identifier la structure ainsi que les classes non encore observées.

Une classe abstraite provient d'un ensemble de classes qui partagent un attribut intéressant. Un tel attribut implique le partage d'un comportement entre celles qui le possèdent. Une super-classe dont les sous-classes vont hériter le comportement est créée. Elle va contenir cet attribut. Le but est donc de rechercher le plus de super-classes possible.

6.2.1.3. Le regroupement des classes

Les super-classes candidates peuvent être identifiées en regroupant les classes connexes. De plus, une classe peut apparaître dans plus d'un groupe. Lorsque le groupe est identifié, un nom est donné à la super-classe en fonction de ce qu'elle représente.

Un certain nombre de super-classes ont déjà été identifiées lorsque les classes ont été placées en catégories. La plupart des classes représentant ces catégories sont, en fait, des super-classes abstraites des classes de cette catégorie.

6.2.1.4. La mémorisation des super-classes

Lorsque les super-classes ont été identifiées, elles sont mémorisées sur des fiches, une classe par fiche. Les sous-classes sont également écrites sur les lignes en dessous du nom. Toutes les fiches de classe déjà réalisées vont être reprises et leurs super-classes et leurs sous-classes y sont ajoutées. L'aspect de ces fiches est représenté à la figure A1.6-1.

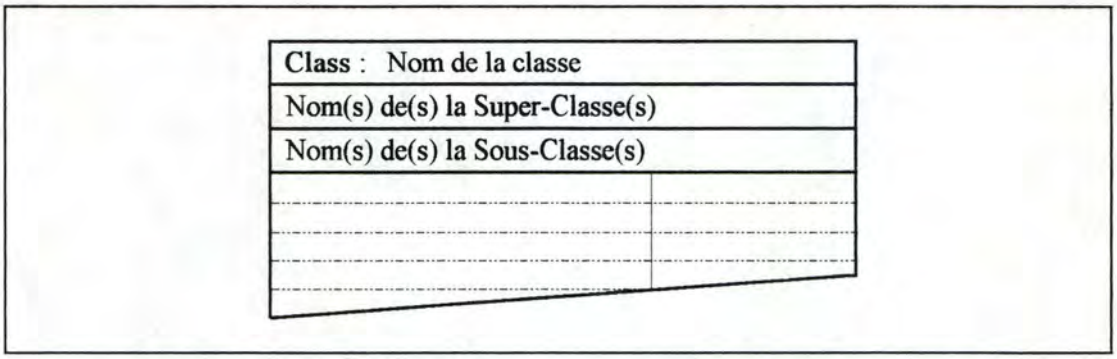


Figure A1.6-1: Fiche d'une classe avec ses super-classes et ses sous-classes.

6.2.2. Les responsabilités

6.2.2.1. La notion de responsabilité

Les responsabilités renferment deux aspects clés:

- la mémoire de l'objet,
- les actions que peut effectuer un objet.

Les responsabilités sont donc les services que l'objet fournit pour tous les contrats qu'il réalise. Lorsqu'une responsabilité est assignée à une classe, le fait que ces responsabilités soient remplies par toutes les instances de cette classe est posé. L'ensemble des services d'un contrat particulier est sous la responsabilité du serveur de ce contrat. Ces responsabilités représentent en fait les seuls services visibles disponibles. Les services se trouvant derrière ces responsabilités sont cachés vis-à-vis des autres objets. Le contrat et les responsabilités disent clairement *ce que le service fait*, mais pas *comment il le fait*.

6.2.2.2. L'identification des responsabilités

Les responsabilités peuvent êtres identifiées en utilisant les spécifications de l'application et en étudiant les classes déjà identifiées. Dans les spécifications, tous les verbes peuvent indiquer une responsabilité pour le sujet de la phrase. De même, l'information qui doit être gardée ou manipulée indique des responsabilités.

Les classes déjà identifiées sont évidement indispensables. Si elles ont été identifiées, cela signifie qu'elles sont nécessaires et qu'elles ont au moins une responsabilité. Le nom choisi, le rôle joué ou une responsabilité déjà identifiée sur une autre classe suggère généralement quelles responsabilités sont présentes dans cette classe.

6.2.2.3. L'attribution des responsabilités

Chaque responsabilité est attribuée à la classe ou aux classes auxquelles elle doit logiquement appartenir. Si certaines responsabilités sont difficiles à attribuer, voici quelques règles qui peuvent aider:

- *répartir de manière égale l'intelligence du système.* Une classe comprend plus ou moins d'intelligence en fonction de ce qu'elle sait et de ce qu'elle peut faire. Une juste répartition aide à la compréhension de la classe examinée.
- *donner les responsabilités de la manière la plus générale possible*, de façon à ce que des responsabilités partagées par différentes classes puissent être identifiés plus facilement.
- *garder, si possible, les opérations avec les informations qu'elles manipulent.* Ce point est à la base de l'encapsulation qui demande que les éléments semblables restent ensemble. De cette manière, si les informations d'un objet changent, il n'y a pas d'envoi de message.

- *garder l'information à propos d'une chose à une seule place.* Si plus d'un objet doit connaître la même information pour réaliser une action, il y a trois possibilités pour résoudre ce dilemme:
  - un nouvel objet est spécialement créé pour jouer le rôle de stockage de cette information. Celle-ci est accessible aux autres objets par l'intermédiaire de messages que ceux-ci lui envoient.
  - l'utilisation de l'information peut être d'importance variable entre les objets qui en ont besoin, l'information sera donc assignée à l'objet dont la responsabilité principale est de garder cette information.
  - il peut être bon de rassembler les différents objets qui ont besoin de cette information en un seul objet.
- *partager les responsabilités entre les objets liés.* Certaines responsabilités semblent être une responsabilité agrégée. Il serait donc intéressant de diviser cet agrégat ou de le répartir sur plusieurs objets.

6.2.2.4. Les relations entre classes

D'autres responsabilités peuvent être découvertes par l'analyse des relations existant entre les différentes classes. Trois relations sont particulièrement utiles:

- la relation "*est de type*" ("is kind of"). Cette relation indique le plus souvent une relation de sous-classe à super-classe. Lorsque c'est le cas, les responsabilités sont assignées aux super-classes.
- la relation "*est analogue à*". Lorsqu'une classe a, avec une autre partie du système, une relation qui est analogue à celle qu'a une autre classe, alors, les deux classes partagent peut-être la même responsabilité ou une analogue.
- la relation "*est une partie de*". Le fait qu'une classe soit composée d'instances d'autres classes ne signifie pas qu'elle ait exactement le même comportement que ses éléments.

Le processus d'attribution des responsabilités n'est pas un processus linéaire mais, comme dans toute l'approche orientée objets, un processus itératif. En général, les difficultés proviennent d'une classe qui n'a pas encore été identifiée ou d'une responsabilité qui peut être assignée à plus d'une classe. Il faut alors recourir à des choix arbitraires.

Les responsabilités vont également figurer sur les fiches des classes.

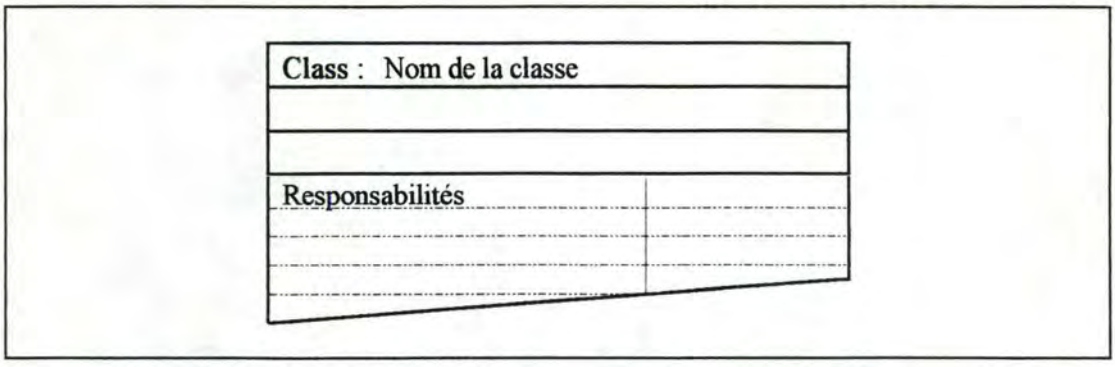


Figure A1.6-2: Fiche d'une classe avec ses responsabilités.

6.2.3. Les collaborations

La manière dont les classes interagissent entre elles va maintenant être déterminée.

6.2.3.1. La notion de collaboration

Les *collaborations* représentent les demandes d'un client à un serveur en vue de remplir les responsabilités du client. La collaboration est donc la matérialisation du contrat entre un client et un serveur. Un *objet collabore* avec un autre si, pour remplir ses responsabilités, il a besoin d'envoyer un message à l'autre objet.

Les collaborations sont très importantes car elles révèlent les flux de contrôle et d'information pendant l'exécution de l'application. Ces flux peuvent aider à définir certains sous-systèmes, utiles pour la suite de la conception.

6.2.3.2. L'identification des collaborations

Pour identifier ces collaborations, un certain nombre de questions doivent être posées et ce, pour chaque responsabilité de chaque classe; la classe est-elle capable de remplir cette responsabilité toute seule? Si non, de quoi a-t-elle besoin et auprès de quelles classes peut-elle obtenir ce dont elle a besoin?

Chaque responsabilité partagée entre différentes classes représente une collaboration entre ces classes.

Il est également intéressant d'identifier les collaborations dont trois types semblent utiles:

- la relation "*fait partie de*". Ces relations sont de deux types: la relation entre la *classe agrégée* et les objets qui la composent et la relation entre la *classe contenant* et les éléments qu'elle contient.
- la relation "*a la connaissance de*". Les classes peuvent parfois connaître quelque chose d'une autre classe sans y être liées d'une autre manière.
- la relation "*dépend de*". Les classes sont souvent attachées entre elles.

Ces collaborations vont aussi se retrouver sur les fiches des classes.

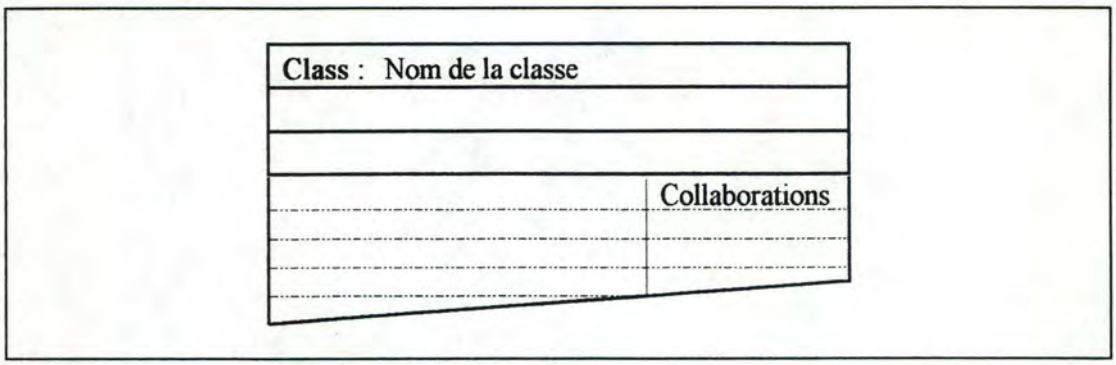


Figure A1.6-3: Fiche de classe avec les collaborations.

La phase d'exploration étant terminée, l'ensemble des fiches actuellement disponibles reprend:

- les classes, super-classes et sous-classes,
- les responsabilités,
- les collaborations.

Dans la phase suivante, l'analyse sera poursuivie par la structuration des classes.

6.2.4. Les hiérarchies

Pour réaliser l'analyse, une compréhension plus globale de notre conception devient nécessaire. Il en va de même des relations d'héritages existantes.

6.2.4.1. La notion de hiérarchie

L'utilisation d'un certain nombre d'outils va aider à acquérir cette compréhension globale.

a) Les graphes de hiérarchies

Ces graphes donnent une représentation de la relation d'héritage entre les classes en relations. Les classes sont symbolisées par des rectangles contenant le nom de la classe. L'héritage est indiqué par une ligne entre la super-classe et la sous-classe et en positionnant sur la page, la super-classe au dessus de sa sous-classe.

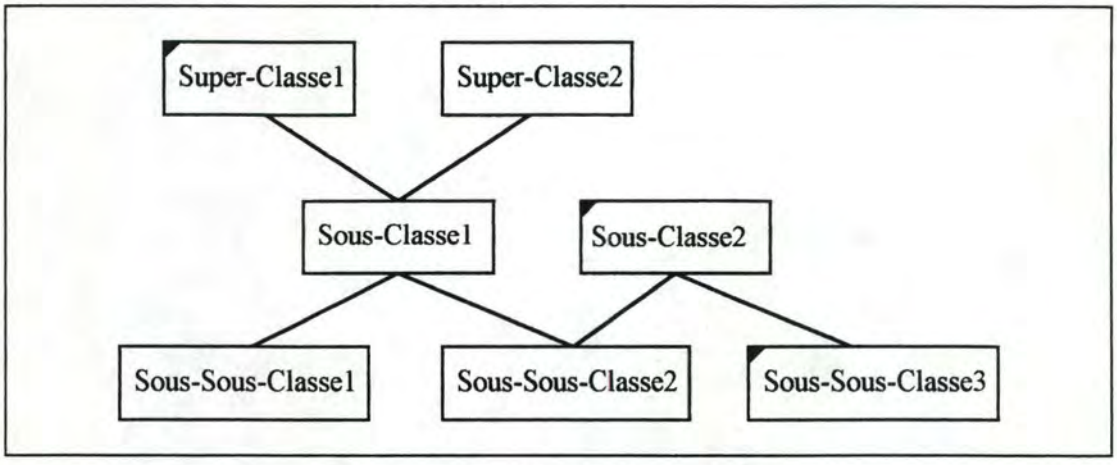


Figure A1.6-4 : Graphe de hiérarchies.

Pour ce type d'analyse, il faut faire en plus la distinction entre les classes abstraites et les classes concrètes:

- une classe abstraite est une classe qui ne sera jamais instantiée, elle se trouve donc dans la partie haute du graphe,
- une classe concrète est conçue pour être instantiée, elle peut cependant faire partie du processus d'héritage.

Le coin supérieur droit des classes abstraites est noirci et cette différence est notée sur les fiches de chaque classe.

b) Les diagrammes de Venn.

Ces diagrammes permettent de nous montrer quelles sont les responsabilités partagées par les classes. La figure suivante donne le diagramme de Venn de la figure A1.6-4.

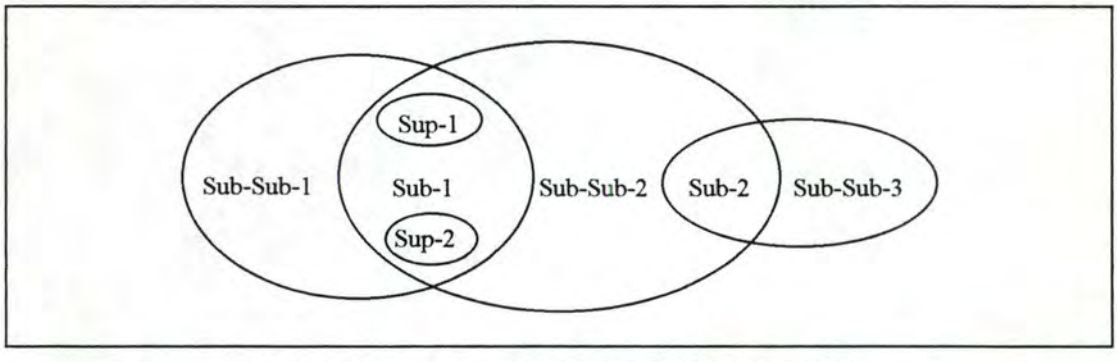


Figure A1.6-5: Diagramme de Venn de la figure A1.6-4.

6.2.4.2. La construction de bonnes hiérarchies

Après la réalisation des graphes de hiérarchies et les diagrammes de Venn, il est possible d'analyser les relations d'héritage entre classes, d'identifier et de résoudre les problèmes de notre conception. L'analyse se base sur les règles suivantes:

- modéliser une hiérarchie "*est de type*"; chaque classe doit être un type spécifique de ses super-classes.
- placer les responsabilités communes le plus haut possible; si un ensemble de classes possède une responsabilité commune, alors elles doivent l'hériter d'une super-classe commune. Si cette super-classe n'existe pas, il faut la créer et déplacer la responsabilité.
- s'assurer que les classes abstraites n'héritent pas de classes concrètes. Les classes abstraites, par leur nature, possèdent des responsabilités qui sont indépendantes de l'implémentation. Elles ne peuvent donc pas hériter de classes concrètes qui peuvent dépendre de l'implémentation.
- éliminer les classes qui n'apportent pas de fonctionnalités. Les classes qui n'ont pas de responsabilités peuvent être éliminées. Les classes abstraites qui ne définissent pas de responsabilités peuvent également êtres éliminées.

6.2.4.3. L'identification des contrats

Un *contrat* définit un ensemble cohérent de demandes, de responsabilités, qu'un client peut faire à un serveur. Le serveur garantit qu'il répondra à ces demandes. Les responsabilités trouvées dans la phase d'exploration sont à la base de la détermination des contrats que possède une classe.

Celle-ci peut posséder un ou plusieurs contrats distincts. Chaque responsabilité peut prendre part à au moins un contrat, mais toutes les responsabilités ne doivent pas faire partie d'un contrat. Quelques responsabilités représentent le comportement de la classe elle-même, ce sont donc des *responsabilités privées*.

Pour déterminer quelles responsabilités sont comprises dans quel contrat, il y a lieu de tenir compte des règles suivantes:

- grouper les responsabilités utilisées par les mêmes clients ainsi que les responsabilités utilisées ensemble.
- maximiser la cohérence des classes. Tout comme un contrat doit être composé de responsabilités cohérentes, une classe doit posséder un ensemble cohérent de contrats.
- minimiser le nombre de contrats.

Une manière d'appliquer ces règles est de commencer la définition des contrats par le haut de la hiérarchie. Les nouveaux contrats seront définis seulement sur les sous-classes en ajoutant une nouvelle fonctionnalité.

6.2.5. Les sous-systèmes

Deux outils sont ajoutés pour aider à simplifier l'application.

- les graphes de collaborations dans lesquels les sous-classes sont graphiquement insérées à l'intérieur de la super-classe,

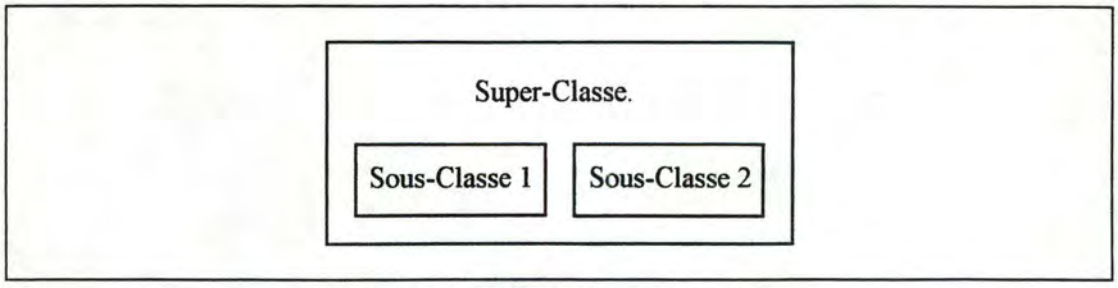


Figure A1.6-6: Graphe de collaborations.

- les contrats sont indiqués par de petits demi-cercles à l'intérieur de la classe à laquelle le contrat est attaché.

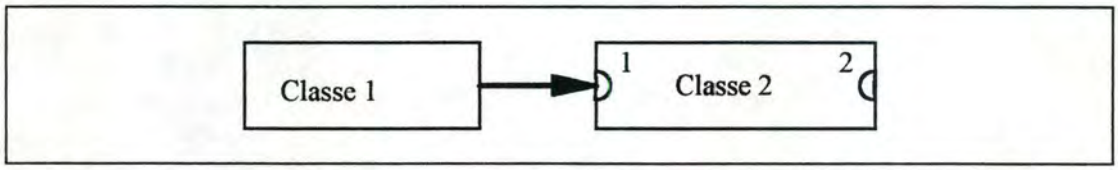


Figure A1.6-7: classe avec deux contrats et une collaboration.

6.2.5.1. La notion de sous-système

Un sous-système est un groupe de classes ou un groupe de classes et d'autres sous-systèmes qui collaborent entre eux en vue de réaliser un certain nombre de contrats. Vu de l'extérieur, un sous-système est un groupe de classes qui travaillent ensembles pour assurer une responsabilité clairement délimitée. Le sous-système n'est pas seulement un rassemblement de classes, il fournit également une bonne abstraction du problème.

Les sous-systèmes sont représentés par des fiches, on y écrit le nom, une petite description au dos de la fiche ainsi que chaque contrat requis par des clients externes au sous-système. Voir la figure A1.6-8.

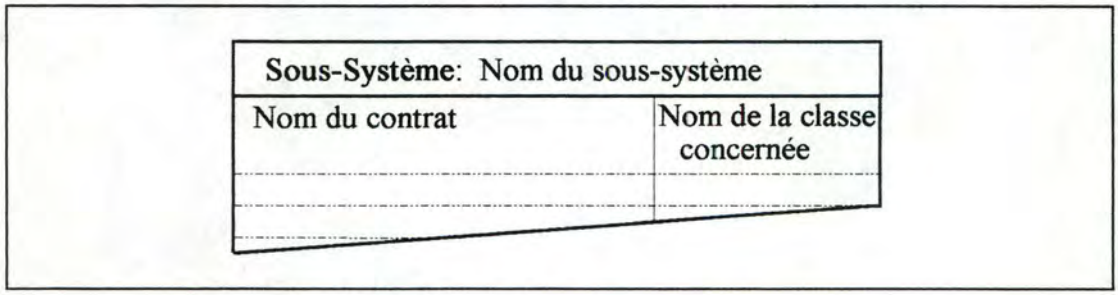


Figure A1.6-8: Fiche de sous-système.

Dans les graphes de collaboration, les sous-systèmes sont représentés par des rectangles dont les coins sont arrondis. De plus, cette représentation encapsule les classes et sous-systèmes, dessine une flèche depuis le contrat du sous-système vers les classes qui réalisent réellement le contrat. Voir la figure A1.6-9.

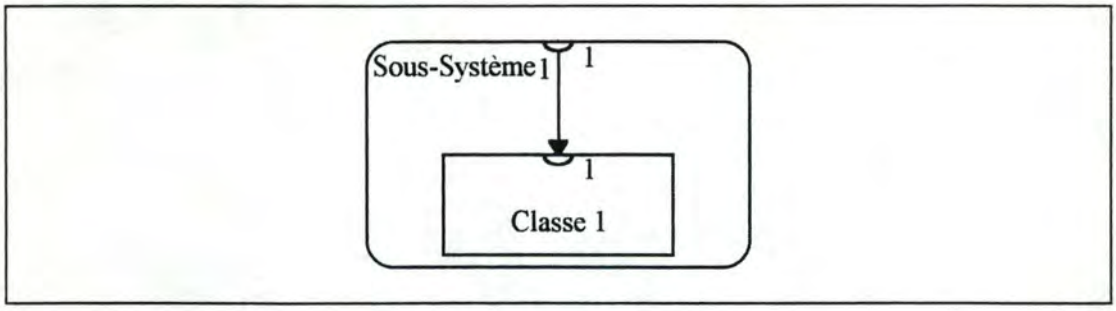


Figure A1.6-9: Un sous-système.

6.2.5.2. L'identification des sous-systèmes

On peut trouver les sous-systèmes en dessinant le graphe des collaborations et en repérant les classes fortement liées. Ces sous-systèmes permettent également de simplifier les graphes de collaboration, pour cela, un certain nombre de règles sont à suivre:

- minimiser le nombre de collaborations qu'une classe a avec les autres classes ou sous-systèmes,
- minimiser le nombre de classes et de sous-systèmes auquels le sous-système fait appel pour remplir une responsabilité,
- minimiser le nombre de contrats différents réalisés par une classe ou un sous-système.

Le graphe de collaboration ainsi obtenu est plus clair et plus facile à lire.

6.3. Les protocoles de représentation

Le but de cette étape est de vérifier le modèle construit. Pour cela, un certain nombre d'étapes seront suivies:

- construire le protocole pour chaque classe,
- écrire une spécification conceptuelle pour chaque classe et pour chaque sous-système,
- écrire une spécification conceptuelle pour chaque contrat.

Un protocole est donc un ensemble d'éléments ou de signatures auxquels une classe doit répondre. Il faut donc que le protocole soit le plus général possible. On va également définir un certain nombre de valeurs par défaut .

6.3.1. La construction du protocole pour chaque classe

Pour cela, les spécifications conceptuelles des méthodes que chaque classe va implémenter vont être définies et un certain nombre de règles respectées:

- utiliser un nom simple pour chaque opération logique,
- associer une opération logique à chaque nom de méthode,

- montrer de manière explicite dans la hiérarchie d'héritage que des classes remplissent une même responsabilité spécifique,
- rendre les signatures des méthodes utilisables de manière générique (pour la réutilisation),
- donner un certain nombre de valeurs par défaut pour tous les paramètres identifiés (toujours pour une réutilisabilité plus aisée).

6.3.2. La spécification des sous-systèmes

De manière similaire, chaque sous-système aura un document. La rédaction de celui-ci observera les règles suivantes:

- écrire le nom du sous-système en début de page et donner la liste de toutes les classes et sous-systèmes encapsulés,
- inclure les références des positions du sous-système dans les graphes de collaborations,
- décrire le sous-système, son but,
- donner la liste des contrats pour lesquels le sous-système est le serveur,
- pour chaque contrat, identifier la classe ou le sous-système qui remplit effectivement ce contrat au sein du sous-système.

Sous-Système : *Nom du sous-système*

Classes : *Nom des classes et des sous-systèmes*

Graphes de collaborations : page *numéro de page*

Description : *Description du sous-système*

Contrats

#. nom du contrat

serveur: *Nom des classes et des sous-systèmes*

- page # -

Figure A1.6-10: Spécification d'un sous-système

6.3.3. La formalisation des contrats

Pour chacun des contrats:

- donner un nom et un numéro unique au sein de la classe ou du sous-système concerné,
- déterminer le ou les serveur(s), le ou les client(s),
- donner une description du contrat.

6.3.4. Le résultat

A la fin de la conception, différents éléments ont été produits:

- un ou plusieurs graphes de hiérarchie montrant l'héritage au sein de l'application,
- un ou plusieurs graphes des collaborations montrant les différents chemins de communication au sein de l'application,
- un ensemble de spécifications formelles de contrats montrant les classes serveurs et clients ainsi que les services couverts par le contrat,
- une spécification pour chaque classe et sous-système.

La phase de conception orientée objets de notre application est ainsi terminée.

ANNEXE 2 :

ENONCÉ COMPLET

DU CAS ÉTUDIÉ.

CADRE GENERAL

Une société de service informatique veut développer un logiciel de gestion hospitalière répondant au besoin de l'hôpital MORTSUBITE.

La société envisage plusieurs autres contrats dans le domaine de la gestion hospitalière; des négociations sont déjà engagées avec d'autres hopitaux de tailles et de besoins différents de ceux de l'hôpital Mortsubite.

Ce premier contrat avec Mortsubite couvre l'informatisation de certaines fonctions de base, et constitue donc une première étape d'une informatisation plus complète de toute la gestion de Mortsubite.

Des qualités essentielles du logiciel à développer sont donc:

- Généralité: la plus grande indépendance possible par rapport aux données particulières de Mortsubite.
- Flexibilité: adaptation facile à des besoins spécifiques d'un autre hôpital.
- Extensibilité: possibilité d'automatisation de nouvelles fonctions en relation avec celles qui sont réalisées, sans devoir changer ces dernières.

On commencera par évoquer les grandes lignes de fonctionnement de l'hôpital Mortsubite, puis on précisera la définition des concepts apparus ainsi que les relations entre ces concepts (schèma conceptuel), pour enfin définir les différentes fonctions à automatiser.

CAS DE L'HOPITAL MORTSUBITE

L'hôpital Mortsubite comprend 200 lits d'hospitalisation partagés entre 12 services de soins. Le nombre de lits dans chaque service de soins varie entre 8 et 24. Chaque service de soins contient un certain nombre de chambres de différents types. Ainsi nous avons les types suivants:

- chambre à quatre lits,
- chambre à deux lits,
- chambre particulière.

On peut considérer que pour un hôpital donné la configuration des chambres et des lits est fixe.

Dans chaque service de soins un poste d'infirmières assure les soins quotidiens.

1. L'accueil

L'admission passe par le service d'accueil qui reçoit les informations nécessaires concernant le malade.

Ce service attribue au patient un lit dans le service de soins correspondant. Le type de la chambre (1,2, ou 4 lits) correspondra au souhait du patient dans la mesure du possible. Il n'y a donc refus d'admission que dans le cas où le service demandé est complétement occupé.

2. L'hospitalisation:

2.1 Mouvements internes et externes: pendant son hospitalisation, le malade peut changer de service et/ou de lit une ou plusieurs fois (transfert en chirugie, retour au service, changement de chambre dans le même service, ...)

2.2 Prestations: pendant son hospitalisation à l'hôpital MORTSUBITE le malade reçoit un certain nombre de soins et un certain nombre de médicaments.

Ces prestations font toujours l'objet d'une prescription d'un des médecins de l'hôpital.

Les infirmières du service de soins assurent le suivi de ces prescriptions.

Après une hospitalisation d'une durée quelconque, le malade quitte l'hôpital. Pour simplifier les choses, on suppposera qu'un malade peut sortir n'importe quel jour de la semaine avec accord du médecin et qu'un malade ne peut pas effectuer plusieurs mouvements sur une même journée.

3. La Facturation

La comptabilité facture toutes les prestations ainsi que tous les jours d'hospitalisation. Le prix d'une prestation dépend de son type, tandis que le prix pour une journée d'hospitalisation dépend lui du type de la chambre.

Pour chaque élément facturé (journée d'hospitalisation ou prestation) une partie est supportée par l'organisme assureur, et le reste par le patient.

Le service de comptabilité prépare mensuellement les factures concernant les malades sortant durant le mois écoulé et envoie ces factures aux patients et aux organismes assureurs. On ne facture donc, que les hospitalisations terminées.

L'hôpital dispose de ses propres tarifs qui donnent :

- le prix plein unitaire pour chaque type de chambre,
- le prix plein unitaire pour chaque type de prestation.

où prix plein = montant à payer par le patient
+ montant supporté par l'organisme assureur.

On dispose aussi, et pour chaque organisme assureur connu, de l'adresse de cet organisme, ainsi que des prix unitaires remboursés par cet organisme pour chaque type de chambre ainsi que pour chaque type de prestation.

!!!! Tous les tarifs ainsi que les parties supportées par les organismes assureurs peuvent changer.

4. Des statistiques.

L'hôpital souhaite disposer de certaines statistiques on_line comme :

- le taux d'occupation des lits de l'hôpital,
- le taux d'occupation des lits d'un service donné,
- le taux d'occupation des lits d'un type de chambre donné.

L'administration souhaite aussi pouvoir calculer annuellement les statistiques suivantes:

- le taux d'occupation de tout l'hôpital réparti sur les mois de l'année. (**Fig 1**)
- le taux d'occupation par service réparti sur les mois de l'année. (**Fig 2**)
- la durée moyenne d'une hospitalisation.

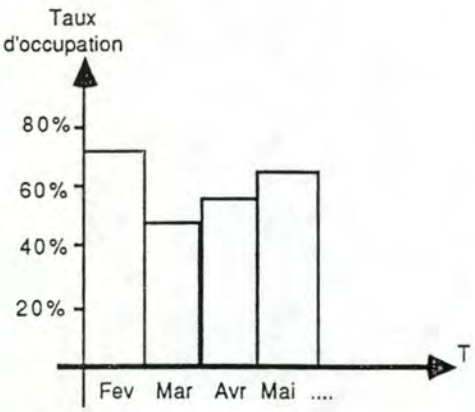


Fig 1

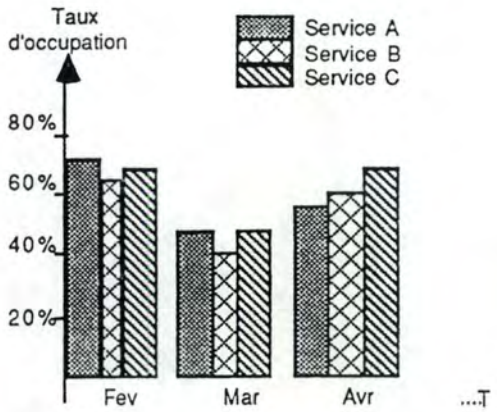
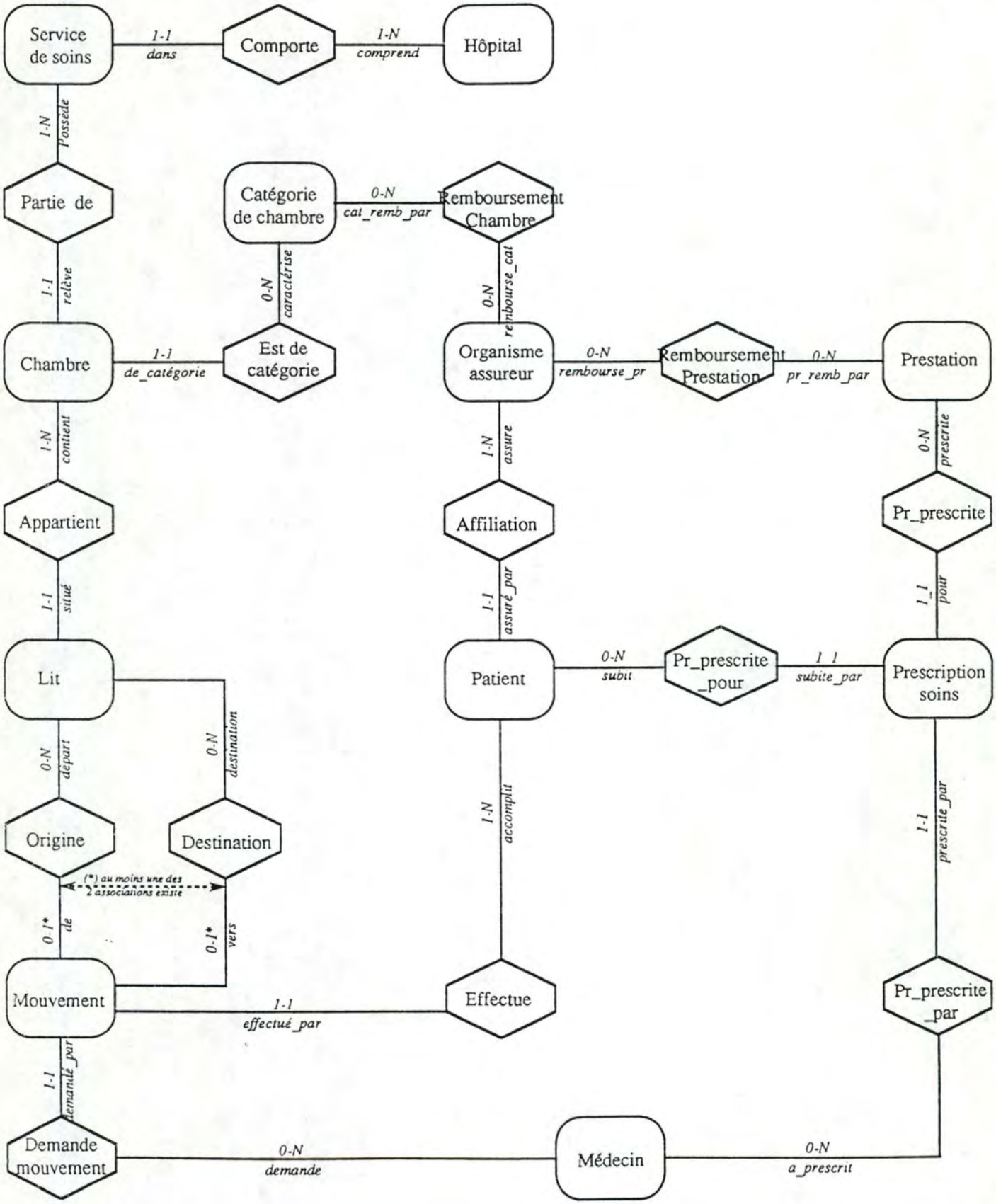


Fig 2

EXTENSIONS PREVUES

- Calcul périodique de certaines statistiques intéressantes.
- Extension à la gestion des prestations et des médicaments.
- Gestion de stock de la pharmacie MORTSUBITE.
- Gestion de réservations préalables pour l'entrée dans certains services (la maternité par exemple)
- Extension de la facturation pour tenir compte des prestations et médiacments.
- Gestion des versements effectués par les patients et les organismes assureurs.
- Extension à la gestion des régimes alimentaires suivi par les malades.
- Protection d'une partie des informations (confidentielles) concernant le patient.
- Une interface d'interrogation de la BD plus élaborée.


Service de soins
1-1
dans
Comporte
1-N
comprend
Hôpital
1-N
Possède
Partie de
Catégorie de chambre
0-N
cat_remb_par
Remboursement Chambre
1-1
relève
0-N
caractérise
0-N
rembourse_cat
Chambre
1-1
de_catégorie
Est de catégorie
Organisme assureur
0-N
rembourse_pr
Remboursement Prestation
0-N
pr_remb_par
Prestation
1-N
contient
1-N
assure
0-N
prescrite
Appartient
Affiliation
Pr_prescrite
1-1
situé
1-1
assuré_par
1_1
pour
Lit
Patient
0-N
subit
Pr_prescrite _pour
1 1
subite_par
Prescription soins
0-N
départ
0-N
destination
1-N
accomplit
1-1
prescrite_par
Origine
Destination
(*) au moins une des 2 associations existe
0-1*
de
0-1*
vers
Mouvement
1-1
effectué_par
Effectue
Pr_prescrite _par
1-1
demandé_par
Demande mouvement
0-N
demande
Médecin
0-N
a_prescrit

SCHEMA CONCEPTUEL

Définition des entités et des associations

ENTITES

ENTITE : **HOPITAL**

DEFINITION:

Organisme assurant des soins médicaux à des malades lors d'une ou de plusieurs hospitalisations au sein de cet hôpital.

IDENTIFIANT(S):

nom_hop : le nom de l'hôpital.

ATTRIBUTS:

date_der_fact : la date de la dernière facturation effectuée.

date_der_stat : la data de la dernière fois qu'un calcul de statistiques périodiques a été fait.

ENTITE : **PATIENT**

DEFINITION :

Toute personne hospitalisée (ou ayant été hospitalisée) dans l'hôpital mortsubite.

IDENTIFIANT(S) :

nr_dos : le numéro de dossier du patient, ce numéro est un identifiant attribué une fois pour toutes à la première hospitalisation de la personne **(id1)**

nom_pat : le nom du patient

pre_pat : son prénom

dnais : sa date de naissance **(id2)**

ATTRIBUTS:

a- **adr_pat** : l'adresse du patient

b- **phone** : son numéro de téléphone

c- **sexe** : M ou F.

d- **etat_civ** : son état civil (célibataire, marié, veuf, ...)

e- **date_sor** : la date du dernier mouvement du type "sortie" effectué par ce patient.

f- **duree_sej:** la durée de la dernière hospitalisation accomplie par le patient.

CONTRAINTES :

-Les attributs (a,b,c,d) sont obligatoires pour un patient "malade".

-Les attributs (e,f) sont obligatoires pour un patient "ancien malade".

ENTITE : **PRESTATION**

DEFINITION :

Type de soin ou de médicament pouvant être donné à un malade et donnant lieu à une facturation.

IDENTIFIANT(S):

code_prest : le code de la prestation.

ATTRIBUTS:

nom_prest : le nom de la prestation.

prix_plein_prest : le prix plein unitaire de la prestation.

ENTITE : **MEDECIN**

DEFINITION :

Toute personnne ayant droit de prescrire une prestation, ou un mouvement.

IDENTIFIANT(S):

nr_med : le numéro du médecin.

ATTRIBUTS:

nom_med : le nom du médecin;

pre_med : le prénom du médecin;

adr_med : l'adresse du médecin.

ENTITE : **LIT**

DEFINITION :

Tout lit d'hospitalisation pouvant être attribué à une personne pendant tout (ou une partie de) son hospitalisation dans mortsubite.

IDENTIFIANT(S):

nr_lit : un numéro identifiant le lit dans l'hôpital.

ENTITE : **CHAMBRE**

DEFINITION :

Partie de l'hôpital contenant de lits d'hospitalisation et faisant partie d'un service de soins.

IDENTIFIANT(S):

nr_ch : un numéro identifiant la chambre au sein de l'hôpital.

ENTITE : **CATEGORIE_DE_CHAMBRE**

DEFINITION :

Classe de chambres ayant toutes le même prix unitaire de séjour pour un lit.

IDENTIFIANT(S):

cat : le nom de cette catégorie (ch_4_lit, ch_particulière, ...).

ATTRIBUTS:

prix_plein_lit_j : le prix plein d'un séjour d'une journée d'hospitalisation dans un lit d'une chambre de cette catégorie.

ENTITE : **SERVICE_DE_SOINS**

DEFINITION :

Unité fonctionnelle de mortsubite comprenant des chambres qui contiennent de lits d'hospitalisation.

IDENTIFIANT(S):

ser : le nom du servive

ENTITE : **MOUVEMENT**

DEFINITION :

Changement d'attribution de lit à une personne ; un mouvement peut donc être d'un des 3 types :
- entrée: mouvement dont l'origine est extérieur à l'hôpital, et la destination est un lit d'hospitalisation.
- sortie: mouvement dont l'origine est un lit d'hospitalisation et la destination est extérieure à l'hôpital.
- transfert: mouvement dont l'origine et la destination sont des lits d'hospitalisation.

IDENTIFIANT(S):

nr_mv : un numéro identifiant le mouvement.

ATTRIBUTS:

date_mv : la date effective du mouvement.

type_mv : (entrée, sortie, ou transfert).

CONTRAINTES

- Un mouvement de type *entrée* est associé à un seul "lit" par "destination".
- Un mouvement de type *sortie* est associé à un seul "lit" par "origine".
- Un mouvement de type transfert est associé à deux "lit"s differents par "origine" et "destination".

ENTITE : **ORGANISME_ASSUREUR**

DEFINITION :

Association qui rembourse une partie des frais d'hospitalisation des personnnes affiliées.

IDENTIFIANT(S):

nr_oa : un numéro identifiant l'organisme assureur.

ATTRIBUTS:

nom_oa : le nom de l'organisme assureur.

adr_oa : l'adresse de l'organisme assureur.

ENTITE : **PRESCRIPTION_SOIN**

DEFINITION :

Exprime la prescription par un médecin d'un soin ou d'un médicament à administrer à un malade.

ATTRIBUTS:

nr_pr : numéro d'ordre identifiant une prescription et attribué lors de la création de cette association.

date_pr : la date prévue pour l'exécution de la prestation.

état pr : l'état actuel de la prescription (en attente ou exécutée).

ASSOCIATIONS

ASSOCIATION : **PR_PRESCRITE**

DEFINITION:

Associe une prescription de soins à la prestation concernée.

ASSOCIATION : **PR_PRESCRITE_POUR**

DEFINITION:

Associe une prescription de soins au patient qui va la subir.

ASSOCIATION : **PR_PRESCRITE_PAR**

DEFINITION:

Associe une prescription de soins au médecin qui la prescrite.

ASSOCIATION : **AFFILIATION**

DEFINITION :

Associe un patient à l'organisme assureur auquel il est affilié.

ATTRIBUTS:

nr_tit : le numéro de matricule du titulaire de l'assurance.

type aff : le type de l'affiliation (vipo, salarié, ...).

ASSOCIATION : **DEMANDE_MOUVEMENT**

DEFINITION :

Associe un mouvement au médecin demandeur de ce mouvement.

ASSOCIATION : **EFFECTUE**

DEFINITION :

Associe un mouvement au patient qui l'a effectué.

ASSOCIATION : **DESTINATION**

DEFINITION :

Associe un mouvement (du type entrée ou transfert) au lit de sa destination ç.à.d. le lit ou le patient se trouvera après le mouvement.

ASSOCIATION : **ORIGINE**

DEFINITION :

Associe un mouvement (du type transfert ou sortie) au lit de son origine ç.à.d. le lit où le patient se trouvait avant le mouvement.

ASSOCIATION : **APPARTIENT**

DEFINITION :

Associe un lit à la chambre où il se trouve.

ASSOCIATION : **PARTIE_DE**

DEFINITION :

Associe une chambre au service de soins dont elle fait partie.

ASSOCIATION : **COMPORTE**

DEFINITION :

Associe un service de soins à l'hôpital dont il fait partie.

ASSOCIATION : **REMBOURSEMENT_CHAMBRE**

DEFINITION :

Associe une catégorie de chambre à un organisme assureur qui rembourse une partie du prix unitaire de séjour dans cette catégorie de chambre.

ATTRIBUTS:

prix_remb_lit_j : le prix remboursé par l'organisme assureur pour un séjour d'une journée dans un lit d'une chambre de cette catégorie.

ASSOCIATION : **REMBOURSEMNT_PRESTATION**

DEFINITION :

Associe une prestation à un organisme assureur qui rembourse une partie du prix unitaire de cette prestation.

ATTRIBUTS:

prix_remb_prest : le prix remboursé par l'organisme assureur pour cette prestation.

ASSOCIATION : **EST_DE_CATEGORIE**

DEFINITION :

Associe une chambre à la classe de chambres dont elle fait partie.

GLOSSAIRE

1- Un **patient** est toute personne admise (au moins une fois) à l'hôpital mortsubite, et ayant donc un numéro de dossier dans cet hôpital.

2- **Malade** est tout patient ayant actuellement un lit attribué.

3- **Ancien malade** est tout patient n'ayant pas actuellement un lit attribué.

4- Un patient malade P **occupe un lit** L si le mouvement le plus récent effectué par ce malade a le lit L comme destination. On dit donc que le lit L **est occupé par** le patient P.

5- Un malade **se trouve dans la chambre** C si ce malade occupe un lit appartenant à cette chambre.

6- Un malade **se trouve dans le service** S si ce malade se trouve dans une chambre C faisant partie du service S.

7- Une **chambre est pleine** si tous les lits appartenant à cette chambre sont occupés.

8- Un **service est plein** si toutes les chambres appartenant à ce service sont pleines.

9- Deux mouvements MV1 et MV2 effectués par un patient P sont des **mouvements successifs** si le patient P n'a effectué aucun autre mouvement entre MV1 et MV2.

10- Un **séjour** du patient P **dans le lit** L est la période écoulée entre deux mouvements successifs MV1 et MV2 de ce patient:

MV1 ayant L comme destination et,
MV2 ayant L comme origine et ,
la date du mouvement MV1 est le début du séjour et
la date du mouvement MV2 est la fin du séjour.

11- Deux **séjours** du patient P sont **successifs** si le début du deuxième séjour est égal à la fin du premier séjour.

12- Une **hospitalisation** d'un patient P est la période écoulée entre D1 et D2 (D1 < D2) dont:
-D1 est la date d'un mouvement "entrée" effectué par le patient P;
-D2 est

soit la date du mouvement "sortie" le plus proche de D1, on parle alors d'une *hospitalisation terminée*,

soit la date du jour s'il n'existe aucun mouvement "sortie" postérieure à D1, on parle alors de l'*hospitalisation courante*.

DESCRIPTION DES FONCTIONS

FONCTION **ADMISSION.**

OBJECTIFS.

Introduire et/ou mettre à jour les informations concernant un patient lors de son admission pour une hospitalisation.

MESSAGES D'ENTREE.

nom_pat : le nom de la personne qui demande l'admission
pre_pat : son prénom
dnais : sa date de naissance
ser : le service de soins demandé
cat : la catégorie de chambre souhaitée
nr_med : le numéro du médecin qui en cas d'admission sera responsable du mouvement d'entrée et de la prescription du régime alimentaire
adr_pat : l'adresse du patient
phone : son numéro de téléphone
sexe: (M ou F).
état_civ : son état civil
nr_oa : identifiant de son organisme assureur.
nr_tit : le numéro du titulaire de l'assurance.
type_aff : son type d'affiliation à l'organisme assureur.

Les données "**ser**", "**cat**", "**nr_med**" identifient des entités de l'hôpital.
Le patient n'a pas effectué un mouvement de sortie à la date du jour.

MESSAGES DE SORTIE.

Message1: "le service est plein".
Message2: "le patient est déjà malade dans mortsubite".
Message3: **<<nr_dos, nr_lit>>**.

ACTIONS SUR LE S.I.

E1: Création d'une occurrence de "*patient*".
E2: Mise-à-jour des attributs d'une occurrence de *"patient"* .
E3: Création des entités et associations: *"affiliation"*, *"effectue"*, *"demande_mouvement"*, *"mouvement"* (du type entrée), et *"destination"*.

REGLES DE TRAITEMENT.

Dénotons par C(i) et E(i) les Conditions et Effets suivants:

C1: L'identifiant introduit correspond à un patient connu (ç-à-d nr_du_dos existant).
C2: L'identifiant introduit correspond à un patient actuellement malade.
C3: Le service désiré est complètement plein.
C4: Les chambres du type souhaité dans le service "ser" sont toutes pleines.
E4: Production du message1.
E5: Production du message2.
E6: Atribution d'un lit dans une chambre du type souhaité.
E7: Attribution d'un lit dans une chambre d'un type autre que le type souhaité.
E8: Production du message3 .

Les règles de traitement sont illustrées par la table suivante :

	R1	R2	R3	R4	R5	R6	R7	R8
C1	V	F	-	-	V	V	F	F
C2	V	V	F	F	F	F	F	F
C3	-	-	V	V	F	F	F	F
C4	-	-	V	F	F	V	F	V
E1							X	X
E2					X	X		
E3					X	X	X	X
E4			X					
E5	X							
E6					X		X	
E7						X		X
E8					X	X	X	X
??		X		X				

FONCTION : **INTRODUCTION_TRANSFERT.**

OBJECTIFS :.

Introduction (si c'est possible) d'un mouvement du type transfert pour un patient malade à l'hôpital.

MESSAGES D'ENTREE.

nr_dos : un numéro de dossier d'un patient malade.
nr_med : le numéro du médecin demandeur du transfert.
nr_lit_or : le numéro du lit d'origine, où le patient est sensé être avant le transfert.
nr_lit_des : le numéro du lit de la destination voulue.

Les données "**nr_dos**", "**nr_med**", "**nr_lit_or**" et "**nr_lit_des**" identifient des entités de l'hôpital.
Le patient n'a effectué aucun mouvement à la date du jour.

MESSAGES DE SORTIE.

Message1: "le patient n'est pas dans le lit d'origine".
Message2: "le lit de la destination est occupé".

ACTIONS SUR LE S.I.

Création des occurrences de: *"mouvement"*, *"demande-mouvement"*, *"effectue"*, et *"destination"* .

REGLES DE TRAITEMENT.

Si le patient se trouve dans le lit d'origine, et si la destination est un lit non-occupé alors il faut mettre le S.I à jour pour ajouter le mouvement demandé; sinon, il faut produire le message adéquat.

FONCTION : **INTRODUCTION_SORTIE**

OBJECTIFS.

Introduction (si c'est possible) d'un mouvement du type sortie pour un patient malade à l'hopital.

MESSAGES D'ENTREE.

nr_dos : le numéro de dossier d'un patient malade.
nr_med : le numéro du médecin demandeur du transfert.
nr_lit_or : le numéro du lit d'origine, où le patient est sensé être.

Les données "**nr_dos**", "**nr_med**" et "**nr_lit_or**" identifient des entités de l'hôpital.
Le patient n'a effectué aucun mouvement à la date du jour.

MESSAGES DE SORTIE.

"le patient n'est pas dans le lit d'origine".

ACTIONS SUR LE S.I.

Création des occurrences de: *"mouvement"*, *"demande-mouvement"*, *"effectue"*, *et "origine"*.

REGLES DE TRAITEMENT.

Si le patient se trouve dans le lit d'origine alors il faut mettre le S.I à jour pour ajouter le mouvement demandé; sinon, il faut produire le message adéquat.

FONCTION : **PRESCRIPTION_SOIN**

OBJECTIFS.

Introduction d'une prescription dictée par un médecin pour un soin à administrer à un patient malade.

MESSAGES D'ENTREE.

nr_dos : le numéro du dossier du malade concerné.
nr_med : le numéro du médecin.
code_prest :le code de la prestation prescrite.
date_pr : la date d'exécution prévue.

Les données "**nr_dos**", "**nr_med**" et "**code_prest**" identifient des entités de l'hôpital.
La date demandée est postérieure ou égale à la "date_du_jour".

ACTIONS SUR LE S.I.

Création d'une occurrence de *"precription_soins"* .

REGLES DE TRAITEMENT.

L'état de la prescription créée est "**en_attente**".

FONCTION : **PRESTATION_D'UNE_PRESCRIPTION**

OBJECTIFS.

Confirmer l'exécution d'une prescription d'un soin.

MESSAGES D'ENTREE.

nr_pr : le numéro de la prescription.

"**nr_pr**" 'identifie une prescription de soins en attente prévue pour la date du jour.

ACTIONS SUR LE S.I.

Changement de l'attribut **"état"** de la prescription concernée.

FONCTION : **ANNULATION_PRESCRIPTION**

OBJECTIFS.

Supprimer une prescription d'une prestation non exécutée (par exemple à cause d'un changement de traitement, de la sortie du malade,....)

MESSAGES D'ENTREE.

nr_pr : le numéro de la prescription concernée.

"**nr_pr**" 'identifie une prescription de soins qui est en attente.

ACTIONS SUR LE S.I.

Suppression de l'occurrence concernée de *"prescription_soin"* .

FONCTION : **RAPPEL_PRESTATION**

OBJECTIFS.

Production d'une liste de rappel de prestations par service, cette liste reprend les prestations prévues pour la date du jour (ou pour des date antérieures) et non encore exécutées.

MESSAGES DE SORTIE.

La sortie est un ensemble de listes de rappels (une par service). Chaque élément de cet ensemble est composé du nom de service concerné et d'une liste des rappels concernant ce service. Cette liste est un ensemble de rappels où chaque rappel est un tuple de la forme: <<nr_dos, nr_pr, date_pr, code_prest, nom_prest>>.

ACTIONS SUR LE S.I.

\

REGLES DE TRAITEMENT.

Pour tout service de l'hopital il faut produire une liste de rappels. Cette liste contient les prescriptions à rappeler concernant les malades séjournant dans ce service. Une prescription est à rappeler si sa date prévue est inférieure ou égale à la date du jour et si elle est en attente.

FONCTION : **DEMANDE_INFO_MALADE_1**

OBJECTIFS.

Consulter la base de données pour connaître des informations concernant un patient malade donné.

MESSAGES D'ENTREE.

nr_dos : le numéro du dossier du patient malade cherché.

MESSAGES DE SORTIE.

- "Message1" qui contient les informations suivantes sur le malade même :

nom_pat : le nom du malade
pre_pat : son prénom
dnais : sa date de naissance
adr_pat : son adresse
phone : son numéro de téléphone
état_civ : son état civil
sexe : (M ou F)

- "Message2" qui contient les informations suivantes concernant l'organisme assureur du malade:

nr_oa : le numéro de l'organisme assureur
adr_oa : l'adresse de l'organisme assureur
nom_oa : le nom de l'organisme assureur
nr_tit : le numéro du titulaire de l'assurance
type_aff : son type d'affiliation (VIPO, ...)

- "Message3" qui contient des informations concernant les mouvements effectués par le malade lors de l'hospitalisation courante. "Message3" est une liste de mouvements dans laquelle chaque ligne représente un mouvement effectué par le malade concerné; une ligne est un tuple de la forme: <<date_mv, type_mv, nr_med, nr-lit_or, nr_lit_des>>.

"Message4" qui contient des informations concernant les prestations subies par ce malade lors de l'hospitalisation courante. "Message5" est une liste de prestations dans laquelle chaque ligne représente une prestation effectuée pour le malade concerné; une ligne est un tuple de la forme: <<nr_pr, date_pr, code_prest, nr_med>>.

ACTIONS SUR LE S.I.

\

REGLES DE TRAITEMENT.

Les informations fournies en sortie concernent le patient malade identifié par le numéro du dossier "**nr_dos**" fourni en entrée.

FONCTION: **DEMANDE_INFO_MALADE_2**

OBJECTIFS.

Consulter la base de données pour connaître des informations concernant un patient malade donné.

MESSAGES D'ENTREE.

nom_pat : le nom du malade
pre_pat : son prénom
dnais : sa date de naissance

MESSAGES DE SORTIE.

- "Message1": Les informations suivantes sur le malade même :
nr_dos : le numéro du dossier du malade cherché.
adr_pat : son adresse
phone : son numéro de téléphone
état_civ : son état civil
sexe : (M ou F).

- "Message2", "Message3" et "Message4", sont identiques aux messages portant le même nom de la fonction précédente.

ACTIONS SUR LE S.I.

\

REGLES DE TRAITEMENT.

Les informations fournies en sortie concernent le patient malade identifié par le triplet fourni en entrée.

FONCTION: **DEMANDE_INFO_ANC_MALADE**

OBJECTIFS.

Consulter la base de données pour connaître un minimum d'informations concernant un patient qui a accompli une hospitalisation. (En cas du retour du patient par exemple,...).

MESSAGES D'ENTREE.

nom_pat: le nom du patient cherché.
pre_pat : son prénom.
dnais : sa date de naissance

MESSAGES DE SORTIE.

nr_dos: le numéro du dossier du patient.

date_sor: la date de sortie de la dernière hospitalisation.
duree_sej: la durée de la dernière hospitalisation accomplie par le patient à l'hopital.

ACTIONS SUR LE S.I.

\

REGLES DE TRAITEMENT.

Les informations fournies en sortie concernent le patient ancien malade identifié par le triplet fourni en entrée.

FONCTION : **SELECTION_MALADES**

OBJECTIFS.

A partir de certains critères de sélection produire une liste des malades respectant ces critères.

MESSAGES D'ENTREE.

Un critère élémentaire défini sur un attribut.
On considérera dans un premier temps les critères élémentaires suivants:

- "malade séjournant actuellement dans le service Y"
- "malade séjournant actuellement dans une chambre de catégorie Z"
- "malade hospitalisé à l'hopital depuis plus de N jours".

MESSAGES DE SORTIE.

Une suite de numéros de dossier "**nr_dos**" de malades respectant le(s) critère(s).

ACTIONS SUR LE S.I.

\

REGLES DE TRAITEMENT.

Un malade est repris dans la suite résultat si et seulement si ce malade respecte les critères données en entrée.
La suite résultat est triée par ordre croissant de nr_dos.

FONCTION : **PRODUCTION_FACTURES**

OBJECTIFS.

Facturer périodiquement les frais de séjours et de prestations concernant les patients sortis durant la période écoulée depuis la dernière facturation. Deux ensembles de factures seront produits : (i) Les factures des patients, chacune de ces factures reprend les frais à charge d'un patient à facturer et (ii) les factures des organismes assureurs, chacune de ces factures reprend les frais à charge d'un organisme pour un patient affilié

MESSAGES DE SORTIE.

- Un ensemble de factures destinées aux patients. Une facture de cet ensemble est composée des parties suivantes:

• L'en-tête : un tuple de la forme <<nr_dos, nom_pat, adr_pat, nom_oa, adr_oa>>.
• La partie "frais de séjour": une suite de lignes ayant chacune la forme:
<< catégorie, prix_plein_lit_j, prix_remb_lit_j, prix_pat_lit_j, date_debut_sej, date_fin_sej, duree_sej, cout_sejour_oa, cout_sejour_pat >>.
• La partie "frais de prestations": une suite de lignes ayant chacune la forme:
< code_prest, nom_prest, date_pr, prix_plein_prest, prix_remb_prest, prix_pat_prest>>.
• Le total dû par le patient.

- Un ensemble de factures destinées aux organismes assureurs des patients à facturer. Une facture de cet ensemble a la même structure et le même contenue qu'une facture destinée au patient, sauf en ce qui concerne la dernière partie qui contient —dans ce cas— le total dû par l'organisme assureur.

ACTIONS SUR LE S.I.

Mise à jour de l'attribut "date_der_fact" de hôpital.

REGLES DE TRAITEMENT.

- Toute "hospitalisation" terminée entre la date de la dernière facturation et la date du jour est une "hospitalisation à facturer". Le patient concerné est le "patient à facturer".

- A chaque "hospitalisation à facturer" correspond une et une seule facture dans l'ensemble des factures adressées aux patients et une et une seule facture dans l'ensemble des factures adressées aux organismes assureurs; ces deux ensembles ne contiennent que des factures relatives à des "hospitalisations à facturer". Les 2 factures correspondant à une hospitalisation d'un patient sont identiques sauf pour la partie "total".

- Toute paire de mouvements successifs de l' "hospitalisation à facturer" donne lieu à une et une seule ligne dans la partie "frais de séjour" de chacune des 2 factures concernant cette hospitalisation.

- Toute prescription de soins concernant un "patient à facturer", avec un état "exécutée" et une date comprise entre le début et la fin de l' "hospitalisation à facturer" donne lieu à une et une seule ligne dans la partie "frais de prestations" de chacune des 2 factures concernant cette hospitalisation.

- Dans une facture:

"**prix_pat_lit_j**" = "prix_plein_lit_j" - "prix_remb_lit_j"
"**duree_sej**" = "date_fin_sej" - "date_debut_sej".
"**cout_sej_oa**" = "duree_sej" * "prix_remb_lit_j"
"**cout_sejour_pat**" = "duree_sej" * "prix_pat_lit_j"
"**prix_pat_prest**" = "prix_plein_prest" - "prix_remb_prest".

Le total dû par le patient est la somme de tous les "cout_sej_pat" + la somme de tous les "prix_pat_prest".
Le total dû par l'organisme assureur est la somme de tous les "cout_sej_oa" + la somme de tous les "prix_remb_prest".

FONCTION : **INTERROGATION_FACTURE**

OBJECTIFS.

Cette fonction permet de savoir à chaque instant ce que doit un malade donné jusqu'à la date de la demande. (abstraction faite de ce que doit son organisme assureur). Ceci revient à produire une facture concernant un

seul patient et reprenant les frais de son hospitalisation comme si cette hospitalisation se termine le jour même de la demande.

MESSAGES D'ENTREE.

nr_dos : le numéro du dossier du patient malade.

MESSAGES DE SORTIE.

- La structure du message est identique à la structure d'une facture destinée à un patient décrite dans la fonction **PRODUCTION FACTURES** ci-dessus.

ACTIONS SUR LE S.I.

\

REGLES DE TRAITEMENT.

En considérant le malade identifié par "**nr_dos**" comme un "patient à facturer", et considérant son hospitalisation courante comme une hospitalisation qui se termine la date du jour; la facture est parfaitement décrite par les règles de la fonction PRODUCTION FACTURES ci-dessus.

FONCTION : **STATISTIQUES_ON_LINE**

OBJECTIFS.

Calcul on-line de différents taux d'occupation de lits dans l'hôpital mortsubite.

MESSAGES D'ENTREE.

ser: un service de l'hôpital,

ou

cat: une catégorie de chambre de l'hôpital,

ou

"taux_global" : un indicateur qui indique que ce que l'on veut est le taux d'occupation de tout l'hôpital.

MESSAGES DE SORTIE.

Taux_ser: le taux d'occupation du service fourni en entrée,

ou

Taux_cat: le taux d'occupation de la catégorie de chambre fournie en entrée.

ou

Taux_glob: le taux d'occupation global.

ACTIONS SUR LE S.I.

\

REGLES DE TRAITEMENT.

Taux-ser = (100 * nbroc) / nbrtot

où nbroc = le nombre de lits occupés du service "ser".

nbrtot = le nombre total de lits du service "ser".

Taux-cat = (100 * nbroc) / nbrtot
où: nbroc = le nombre de lits occupés appartenant à des chambres de la catégorie "cat" ,
nbrtot = le nombre total de lits appartenant à des chambres de la catégorie "cat".

Taux-glob = (100 * nbroc) / nbrtot
où: nbroc = le nombre de lits occupés dans mortsubite.
nbrtot = le nombre total de lits dans mortsubite.

(Fonction a Option pour 91-92)

FONCTION : **STATISTIQUES_PERIODIQUES**

OBJECTIFS.

Calculer périodiquement quelques statistiques intéressantes concernant les séjours dans mortsubite. Nous envisageons actuellement le taux d'occupation par mois pour tout l'hopital, le taux d'occupation par mois par service, et la durée moyenne d'une hospitalisation.

MESSAGES DE SORTIE.

- Les *taux d'occupation de l'hopital* répartis par mois: une suite de couples <<mois, tauxpm>>.

- Les *taux d'occupation répartis par mois pour chaque service* : un ensemble de suites (une par service). Chaque élément de cet ensemble est composé du nom du service concerné, et d'une suite de statistiques par mois pour ce service: une liste de couples <<mois, tauxpspm>>.

- La durée moyenne d'hospitalisation dans mortsubite.

ACTIONS SUR LE S.I.

Mise à jour de l'attribut "date_der_stat".

REGLES DE TRAITEMENT.

A chaque mois écoulé depuis la date du dernier calcul de statistiques "date_der_stat" correspond un et un seul couple <<mois, tauxpm>> dans la suite des taux d'occupation de l'hopital où:

tauxpm = (100 * nuité) / (jourmois * lithop)
nuité = la somme —pour chaque lit de l'hopital— du nombre de journées d'hospitalisation passées dans ce lit pendant ce mois.
jourmois = le nombre de jours du mois.
lithop = le nombre de lits dans l'hopital.

A chaque service de l'hopital correspond une et une seule suite dans l'ensemble des *taux d'occupation répartis par mois pour chaque service* . Dans cette suite, à chaque mois écoulé depuis la date du dernier calcul de statistiques "date_der_stat" correspond un et un seul couple<<mois, tauxpspm>> où:

tauxpspm = (100 * nuitéser) / (jourmois * litser)
nuitéser = la somme —pour chaque lit du service concerné— du nombre de journées d'hospitalisation passées dans ce lit pendant ce mois.
jourmois = le nombre de jours du mois
litser = le nombre de lits du service "ser".

La durée moyenne est calculée par les formules :

dureemoy = (sommehosp / nombrehosp)
sommehosp = la somme des durées de toutes les hospitalisations terminées entre "date_der_stat" et la date du jour.
nombrehosp = le nombre d'hospitalisations terminées entre "date_der_stat" et la date du jour.

FONCTION 20: **ARCHIVAGE**

OBJECTIFS.

Déterminer, dans l'ensemble des données permanentes, les informations qui ne sont plus nécessaires pour les autres fonctions et supprimer ces informations-là (en gardant une trace sur un support extérieur: papier, bandes...).

MESSAGES DE SORTIE.

Une copie de toutes les informations supprimées.

ACTIONS SUR LE S.I.

Suppression des occurrences de *"mouvement"*, *"effectue"*, *"demande mouvement"*, *"destination"* et, *"origine"* correspondantes à des mouvements archivables.
Suppression des occurrences d'*"affiliation"* archivables.
Suppression des occurrences de *"prescription soins"* correspondantes à des prescriptions archivables.
Suppression des attributs archivables des occurrences de *"patient"* .

REGLES DE TRAITEMENT.

Un attribut, une occurrence d'une entité ou une occurrence d'une association est considéré comme "**archivable**" si il n'est plus nécessaire pour aucune des autres fonctions spécifiées ci-dessus.
