MASTER EN SCIENCES INFORMATIQUES

Proposition d'un environnement d'aide au développement d'applications de gestion hautement interactives

Caucheteur, Gautier; Lens, Jean-François

Award date:
1994

Awarding institution:
Universite de Namur

Facultés Notre-Dame de la Paix
Institut d'Informatique
Rue Grandgagnage, 21 - Namur

Proposition d'un environnement d'aide au développement d'applications de gestion hautement interactives

Caucheteur Gautier, Lens Jean-François

Promoteur : *Prof. François Bodart*

En vue de l'obtention du grade de
Licencié et Maître en Informatique

Année académique 1993-1994

Abstract

The purpose of this thesis is to propose an integrated environment supporting the development of highly interactive business applications. This environment should couple a CASE tool supporting the fonctionnal analysis phase and a 4GL tool through the use of generators. Models and a generation strategy fitted to this kind of tools will be defined for each of the application's components : data, fonctions and the user interface. Relationnal tables will be generated on the foundation of the Entity-Relationship model. A definition of the application's process will make possible to generate Function's declarations. The graphical user interface will be automatiquely deduced from the Fonction Chaining Graph and the two previous models. A specification langage, DSL II, will also be defined to support all these models. Finaly the format of the Powerbuilder's definition files will be presented to illustrate the result of the generation in a particular 4GL.

Resumé

L'objectif de ce mémoire sera de proposer un environnement intégré d'aide au développement d'applications de gestion hautement interactives. Cet environnement couplera un outil CASE supportant l'analyse fonctionnelle et un outil 4GL au travers de générateurs. Des modèles et une stratégie de génération adaptés aux outils 4GL seront définis pour chacun des composants de l'application, à savoir les données, les fonctions et l'interface utilisateur. Des tables relationnelles seront générées sur base du schéma entité-association, la signature des fonctions sera générée sur base du modèle de la statique des traitements et l'interface utilisateur sera générée sur base du graphe d'enchaînement des fonctions et des deux modèles précédents. Un langage de description des spécifications, DSL-II, sera défini pour formaliser ces modèles. Finalement, le format des fichiers de définition de l'outil 4GL Powerbuilder sera finalement considéré afin d'illustrer le résultat de la génération dans un environnement particulier.

Nous remercions vivement toutes les personnes qui ont de près ou de loin contribué à la réalisationde ce travail, en particulier:

Monsieur F. Bodart qui en a assuré la direction;

Messieurs J. Vanderdonckt, J.-M. Leheureux et B. Sacré qui nous ont témoignés de leur disponibilité et qui nous ont donné des conseils judicieux.

Nous remercions également Monsieur R. Welke et tous les membres du 'CIS Department' de la Georgia State University pour leur chaleureux accueil lors de notre stage effectué dans le cadre de ce mémoire.

Derniers remerciements et non des moindres, à nos parents sans qui ces études n'auraient pas été possibles.

« Les hommes demanderont de plus en plus aux machines de leur faire oublier les machines »

Philippe Sollers, *Logiques.*

Table des matières

Partie 2 - Modélisation et stratégie de génération

5 ——————————

6 ——————————

Introduction

Les interfaces graphiques popularisées par le Macintosh d'Apple sont devenues dominantes sur nos postes de travail. La majorité des applications de gestion développées actuellement les ont adoptées. En outre, ces applications sont de plus en plus souvent construites suivant une architecture Client/Serveur de données (l'application frontale se trouve sur un poste de travail et accède au serveur de données). Les applications utilisant ce type d'interface et basées sur cette architecture sont évidemment plus conviviales et plus efficaces mais plus complexes à développer. Les environnements de développement d'applications Client/Serveur, de plus en plus répandus actuellement, ont été conçus pour permettre de développer simplement et rapidement ce type d'applications.

Bien que ces outils peuvent paraitre extrêmement puissant en permettent un prototypage rapide et un développement accéléré des applications, ils peuvent néanmoins inciter le développeur à négliger des activités importantes du processus de développement telle que l'analyse conceptuelle. Une utilisation trop libérale de ce type d'outil peut se révéler une erreur à long terme. En effet, un manque de rigueur dans la conception et le développement ne peut que déboucher sur des applications médiocres. D'autre part, celles-ci ont tendance à être mises trop rapidement en exploitation et beaucoup d'entre elles ne respectent pas les besoins initiaux. Sans être passé auparavant par une phase de conception et sans avoir maintenu un minimum de documentation, la modification de ces systèmes s'avèrera extrêmement difficile.

Afin de profiter pleinement de cette nouvelle vague d'outils, il serait intéressant de concevoir l'utilisation de ces outils avec une méthode d'analyse classique telle que IDA [Bod-Pig,89]. Si une telle méthode reconnaît évidemment l'importance de la phase d'analyse fonctionnelle, elle reconnaît aussi la nécessité d'une systématique dans la réalisation des applications correspondantes. Dans cette perspective, il est naturel d'envisager des générateurs d'applications qui réaliseraient cette dérivation systématique.

L'objectif de ce mémoire sera de proposer un environnement intégré d'aide au développement d'applications de gestion supportant l'analyse conceptuelle et réalisant la génération automatique d'un prototype d'application. Il sera composé d'un outil CASE supportant l'analyse fonctionnelle, de générateurs et d'un outil 4GL. Le couplage de ces outils devrait ainsi permettre de supporter tout le processus de développement d'une application. Il s'agira donc d'identifier les modèles de spécification utiles et de les adapter en fonction des caractéristiques des outils de développement envisagés. Nous dégagerons également une stratégie de génération basée sur les modèles retenus.

Dans une première partie, ce travail va souligner l'émergence des environnement 4GL de développement d'applications C/S tels que Powerbuilder. Leurs caractéristiques essentielles seront mises en évidence. Le prototypage rapide qu'ils induisent ainsi que le danger d'une utilisation trop libérale de ce type d'outil sera également souligné. Cela nous amènera à proposer un environnement intégrant ce type d'outil dans un processus de développement plus rigoureux et ayant pour objectif la génération automatique d'un prototype d'application sur base des spécifications conceptuelles.

Dans la deuxième partie, nous traiterons les aspects modélisation et génération devant être pris en compte par l'environnement. Nous montrerons, premièrement, comment une base de données relationnelle pourra être dérivée sur base d'un schéma E/A. Nous identifierons, ensuite, l'architecture des applications 4GLs de manière à envisager par la suite la modélisation et la génération des différents composants de l'application. Nous verrons que des signatures de fonctions pourront être générées sur base du modèle de la statique des traitements. Une structure type d'interface sera définie de manière à pouvoir réaliser la génération systématique. La génération des OI de l'interface et des scripts sera réalisée sur base du modèle E/A, de la statique et du graphe d'enchaînement des fonctions. Nous modifierons et étendrons ces modèles de manière à adapter l'analyse conceptuelle au style graphique de l'interface et au style événementiel du dialogue. Un langage de spécification, formalisant les modèles retenus, sera défini. Nous terminerons en spécifiant le format des fichiers résultats de la génération pour un outil 4GL particulier, Powerbuilder.

Si la modélisation et la génération des trois composantes d'une application doivent être prises en considération, à savoir la base de données, les traitements et l'interface utilisateur, l'accent sera néanmoins mis sur l'aspect interface, et la gestion du dialogue en particulier. La génération de base de données relationnelle sur base d'un schéma entité-association a déjà été développée, alors que celle des traitements s'avérera réduite si l'on se limite à des modèles de haut niveau sans langage formel. La génération d'interface, et plus précisément celle du dialogue, n'est quant à elle, pas encore très répandue et fait actuellement l'objet de nombreuses recherches.

Partie 1

Couplage d'un outil CASE et d'un outil 4GL

1. Trois évolutions qui se croisent
2. Les environnements de développement d'applications client/serveur
3. Cycle de vie ou prototypage rapide ?
4. Proposition d'un environnement d'aide au développement d'applications de gestion

Les interfaces graphiques, l'architecture Client/Serveur et les langages de quatrième génération ont abouti à la création des environnements de développement d'applications Client/Serveur. Ces environnements promouvant un prototypage rapide ne doivent pas être utilisés de façon trop libérale. C'est pourquoi il est intéressant de coupler l'utilisation de ces environnements et une analyse approfondie de l'application à développer. Nous proposerons une approche qui concilie les avantages d'une approche centrée sur le cycle de vie et ceux du prototypage rapide. Pour la supporter, nous proposerons un environnement d'aide au développement d'applications de gestion hautement interactives qui est un couplage d'un outil CASE et d'un outil 4GL par l'intermédiaire de générateurs.

1

Trois évolutions qui se croisent

Les interfaces graphiques, l'architecture Client/Serveur et les langages de quatrième génération sont des tendances actuelles dans le domaine du développement de logiciels. Ces 3 domaines se sont en quelque sorte croisés et ont donné naissance à des outils 4GL bien adaptés aux exigences actuelles, c'est à dire orienté interface graphique et permettant de développer des applications Client/Serveur.

1. Les interfaces graphiques

Les interfaces du type de MS-Windows sont des interfaces d'utilisation graphique basées sur les concepts de multi-fenêtrage, d'icône, de menu déroulant et sur l'utilisation de la souris permettant à l'utilisateur de manipuler des objets à l'écran [Mei,91].

Ce type d'interface place l'utilisateur au centre, alors que les applications traditionnelles (Dos, grands systèmes) sont centrées sur elles-mêmes. Le concepteur de l'application se concentre d'abord sur l'application, son contenu, sa dynamique et ne considère l'utilisateur que comme un fournisseur de données.

L'approche Windows est diamétralement opposée. Le centre de l'application est l'utilisateur ou plus exactement l'interface utilisateur; l'utilisateur pouvant à tout moment choisir ce qu'il veut faire, par exemple clicker sur la partie de l'écran qui l'intéresse. Avec les interfaces graphiques, on passe d'un ancien type de dialogue contrôlé par l'ordinateur, qui présentait à l'utilisateur des écrans successifs, à un type de dialogue contrôlé par les actions de l'utilisateur traduites sous forme d'événements auxquels l'ordinateur peut répondre [Mei,91].

2. L'architecture Client/Serveur

La raison d'être originelle du traitement Client/Serveur était de sous-traiter une part du travail du système host central auprès de machines plus légères, ceci afin d'optimiser l'utilisation des capacités de traitements disponibles. Dans les anciens systèmes, en effet, c'est le host central qui fait tout le travail. Les terminaux des utilisateurs se bornent à afficher les résultats et à collecter les entrées de données via le clavier afin de les transmettre pour traitement au host. Mais qu'entend-on par Client/Serveur ?

2.1. Modèle Client/Serveur

Le modèle Client/Serveur est un modèle de communication dissymétrique entre 2 logiciels où un logiciel client demande par un message un service à un logiciel serveur qui lui donne la réponse sous forme d'un message (Fig. 1.1) [Mei,91]. Ce modèle permet de répartir des logiciels clients et serveurs sur différentes machines d'un réseau et donc d'optimiser la capacité de traitement des différentes machines. Par abus de langage, on appelle serveur la machine qui supporte le logiciel serveur; on parlera de serveur de fichiers, de serveur de données, ...

Figure 1.1 - Modèle Client/Serveur

Il y a plusieurs manières de répartir les logiciels clients et serveurs, autrement dit plusieurs types de Client/Serveur; parmi ceux-ci, le Client/Serveur de données selon lequel la gestion des données de l'application est réalisée sur le serveur, en l'occurrence par un SGBD (Fig. 1.2).

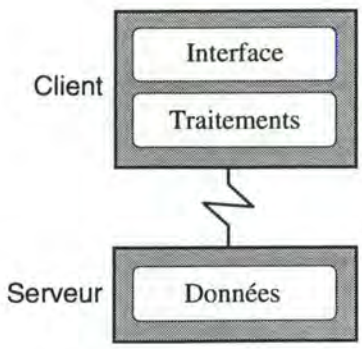

Figure 1.2 - Client/Serveur de données

L'architecture Client/Serveur de données est actuellement la plus répandue. Grâce à celle-ci, les données peuvent être partagées et sont alors disponibles pour plusieurs applications clientes. Seules les requêtes sur les données transitent par le réseau, toutes

les opérations sur les données se font sur le serveur et l'application cliente envoie les requêtes et en reçoit le résultat.

2.2. Serveur de données et PCs

Les réseaux de PCs, assez récents, deviennent de plus en plus populaires parce que les utilisateurs se sentent plus à l'aise sur un PC que sur un terminal[1]. Ces réseaux peuvent communiquer avec d'autres réseaux et donc avec des machines plus puissantes telles que des stations de travail ou des systèmes de type host central. Il est ainsi possible de réaliser des applications sur des PCs qui communiquent avec un SGBD qui se trouve sur une station de travail assez puissante pour supporter les requêtes de plusieurs utilisateurs, autrement dit des applications de type serveur de données. Cette station sera le serveur de données et on appellera l'application qui y accède l'application cliente. Ceci permet de reléguer toute la gestion de l'interface utilisateur et les traitements sur un poste individuel et de communiquer via le réseau avec le SGBD sur un serveur où les données sont partagées entre utilisateurs. On allie donc la puissance des serveurs de données à la convivialité des PCs.

3. Les langages de quatrième génération (4GLs)

Les 4GLs furent inventés à la fois parce qu'il fallait tenir compte de plus en plus de l'importance de l'utilisateur et parce qu'il y avait un besoin d'augmenter la productivité dans la programmation.

Ils sont plus conviviaux, plus simples à utiliser et permettent ainsi notamment à des non-professionnels de la programmation, comme des analystes ou utilisateurs finaux, de développer des applications, ce qui s'avère une des tendances actuelles. Ces 4GLs ont pour objectif d'offrir la possibilité de construire facilement des applications sans utiliser une syntaxe étrangère et de permettre à l'utilisateur d'éviter d'apprendre des mnémoniques, des ponctuations spéciales, ...

Ces 4GLs ont permis une augmentation de la productivité dans la programmation, dans la mesure où une application réalisée avec un 4GL utilise beaucoup moins de lignes de code que les applications créées avec les langages de la génération précédente (les 3GLs comme Cobol, C, Pascal, ...). D'une manière générale, ces 4GLs entraînent une économie de temps, de travail, et permettent l'accès à l'ordinateur à un plus large public.

L'éventail de ces 4GLs est très vaste. Certains sont graphiques, d'autres non, mais tout aussi puissants. Certains sont liés à un SGBD, d'autres sont plus ouverts. Certains sont de simples langages de requêtes, d'autres des outils d'aide à la décision. Certains

[1] Ceci est dû au fait que les applications sur PC y ont une meilleure interface (notamment grâce à l'environnement Windows et son interface graphique).

langages sont très spécifiques tandis que d'autres recouvrent des caractéristiques des différents styles.

Aujourd'hui, on peut trouver un 4GL pour presque tous les types d'application que l'on voudrait développer.

Les deux tendances (interfaces graphiques et architecture Client/Serveur) exposées précédemment s'affirmant dans le domaine des applications de gestion, la nouvelle évolution de ces langages a donné naissance à un nouveau type d'outils destinés au développement de ce type d'application. Ces outils sont généralement regroupés sous le nom d'environnements de développement d'applications Client/Serveur.

2

Les environnements de développement d'applications Client/Serveur

L'utilisation des interfaces graphiques et l'architecture Client/Serveur sont, comme nous l'avons vu, des tendances actuelles dans le domaine du développement de logiciels en général et notamment dans celui des applications de gestion.

Le développement d'applications dans l'environnement Windows est traditionnellement complexe et réservé aux programmeurs professionnels. Ceci est d'autant plus vrai lorsqu'il s'agit d'applications Client/Serveur dans lesquelles la gestion du caractère distribué de l'application n'est pas la partie la plus simple à gérer. Les environnements de développement d'applications Client/Serveur (ou outils 4GL) proposent, pour pallier à ces difficultés, un outil convivial permettant un développement productif. Ces outils sont donc bien adaptés pour développer rapidement des applications de gestion Client/Serveur munie d'une interface graphique. Ils commencent à se multiplier et semblent bénéficier, pour le moment, d'une forte audience commerciale.

Ces environnements se composent en fait de l'outil 4GL proprement dit connecté à un SGBD[2]. L'outil 4GL à proprement parler se trouve sur un PC et est connecté à un serveur de données sur lequel se trouve le SGBD. Ces outils reposent donc sur une architecture Client/Serveur dans laquelle l'outil 4GL est client et le SGBD est serveur.

---

[2] Par la suite nous utiliserons indifféremment les termes outil 4GL et environnement de développement d'applications Client/Serveur pour désigner ce type d'outil. Cette catégorie d'outil n'ayant pas réellement une appellation unique reconnue et bien que la notion de 4GL recouvre de nombreux autres outils et langages, nous y ferons référence en utilsant indifféremment les 2 appelations et ce, pour désigner l'outil 4GL proprement dit et le SGBD auquel il est connecté.

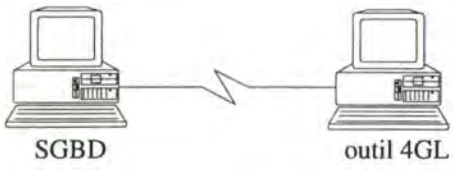

Figure 2.1 - Architecture Client/Serveur des outils 4GL

Ces outils permettent de développer des "applications clientes" munies d'une interface graphique attirante pouvant tourner sur des PCs à l'aide d'un langage propriétaire et d'un environnement de programmation visuelle. Etant donné que ces outils sont construits sur un gestionnaire d'interface, le développement de l'interface utilisateur est rapide et aisé. Toute la gestion de l'écran et des manipulations de l'utilisateur est réalisée par l'outil, ce qui simplifie grandement la programmation.

Il offre aussi des facilités pour la création et la gestion de la base de données située sur le serveur et il permet également le test et le debogage. Ces outils favorisent la création rapide de prototypes et leur test pendant le développement. La rapidité de développement de ces outils permettrait presque au développeur de créer l'application à côté même de l'utilisateur.

Afin de mieux comprendre l'apport de ce type d'outils, décrivons deux caractéristiques fondamentales à l'origine de leur succès: la programmation visuelle et événementielle. Nous présenterons ensuite succinctement l'environnement Powerbuilder de Powersoft que nous utiliserons.

1. Environnement de programmation visuelle

Construire l'interface utilisateur, c'est à dire les fenêtres et leurs constituants, s'effectue à l'aide d'un outil interactif de dessin. La création des fenêtres se fait via des menus simples. Une fois la fenêtre créée, il suffit, à l'aide de la souris, comme dans un logiciel de dessin classique, de sélectionner les objets interactifs (OI) dans une palette et de les disposer dans la fenêtre pour former une présentation élémentaire. Pour améliorer cette présentation, on peut facilement modifier le placement, les dimensions, la couleur ou encore d'autres propriétés de ces objets.

Chaque objet créé, qu'il s'agisse d'une fenêtre ou d'un OI de la palette, est caractérisé par une série de propriétés ou d'attributs (dimensions, couleurs, ...). La valeur de ces attributs est initialisée à la création de l'objet et peut être modifiée de deux manières: initialement lors du dessin de l'interface ou à l'aide d'instructions dans le corps du programme. Lors du dessin de l'interface, la modification des propriétés peut s'effectuer, pour certaines seuelement (placement, position, ...), en manipulant directement les objets tandis que pour d'autres, elle se fait via l'utilisation de boîtes de dialogues permettant de donner des valeurs aux propriétés.

Dans ces environnements, la conception de l'interface se fait donc de manière intuitive et visuelle. Le résultat d'une action est immédiatement visible à l'écran. Le concepteur de l'application voit donc à tout instant l'application telle qu'elle apparaîtra à l'utilisateur lorsqu'elle sera opérationnelle.

Par rapport à la programmation visuelle en général, les objets graphiques manipulés par le programmeur ne sont pas simplement des représentations iconiques, mais bien les objets qui seront finalement créés. C'est le principe de manipulation directe qui est utilisé ici et c'est probablement une des principales raisons pour laquelle la programmation visuelle a plus de succès dans le domaine de la conception d'interface utilisateur que dans la programmation en général [Her,93].

Dans certains outils, la programmation visuelle est également mise à la disposition du programmeur pour réaliser le corps du programme. Des icônes représentant les différentes structures classiques de programmation[2] ainsi que les différentes syntaxes SQL de manipulation de données peuvent être utilisées par le programmeur pour générer le texte correspondant et faciliter ainsi la programmation.

2. Programmation événementielle

Nous savons déjà que l'approche Windows centre l'application sur l'utilisateur ou plus exactement sur l'interface utilisateur. Le dialogue est contrôlé par les actions de l'utilisateur qui sont traduites sous forme d'événements auxquels l'application peut répondre. Ce changement induit des modifications profondes dans la manière de concevoir et de réaliser des applications.

Le type de programmation utilisé par les outils 4GL est appelé événementiel et est à l'opposé de la programmation linéaire traditionnelle. L'utilisateur interagit avec l'application en introduisant ou en modifiant des données dans des champs d'édition, en sélectionnant des commandes dans un menu, en pressant sur des boutons de commande à l'aide de la souris, ... Toutes ces actions sont captées par l'outil[3] et transformées en événements adressés aux OI.

Pour chaque OI, une série d'événements-types est définie. A chacun de ces événements, une partie du code de l'application (généralement appelée script) pourra être associée et sera exécutée dès la génération de l'événement. On pourrait désirer que du code soit exécuté par exemple lorsqu'une fenêtre s'ouvre, lorsqu'un item de menu est sélectionné ou lorsque l'utilisateur quitte un champ particulier. Si pour un événement,

[2] Structures classiques de programmation telles qu'une boucle (while...), une alternative (if... then... else...) ou des choix multiples (case...).

[3] Plus précisément par le gestionnaire de l'interface de l'outil 4GL.

aucun code n'a été écrit, alors il ne se passera rien lorsqu'il sera généré. Pour écrire le code d'une application, il convient donc de déterminer les événements auxquels on souhaite réagir et pour quels objets. Le "maître du jeu" n'est plus le programme mais l'utilisateur qui est à l'origine de ces événements en manipulant l'interface.

En résumé, l'application se construit en concevant l'interface utilisateur dans un premier temps (en choisissant un certain nombre d'OI) et en associant ensuite des scripts aux événements qui peuvent survenir sur les OI de l'interface.

Quant au langage de l'outil 4GL utilisé pour la programmation des scripts, il est orienté-objet concernant l'interface utilisateur et il inclut les structures de programmation classiques (boucles, alternatives, choix multiples) ainsi que le langage SQL, ce qui facilite énormément la manipulation des données. La possibilité d'utiliser des procédures écrites en langage de troisième génération est également prévue.

3. L'environnement Powerbuilder[4]

Powerbuilder est un environnement 4GL doté des caractéristiques et principes présentés précédemment. Il a une approche orienté-objet et on peut le connecter à un des SGBD qu'il supporte (SQL Server, SQLBASE, Oracle, ...).

Le développement d'application se réalise au travers d'un ensemble d'outils, de modules permettant de créer la base de données, de définir les menus et les écrans, d'écrire le code de l'application et de déboguer. Ces modules se trouveront dans le panneau principal de l'environnement sous forme d'icône (Fig. 2.2). Si ces icônes sont affichées séparément dans le menu principal, les modules n'en sont pas moins intimement intégrés et peuvent être exécutés à partir d'un autre.

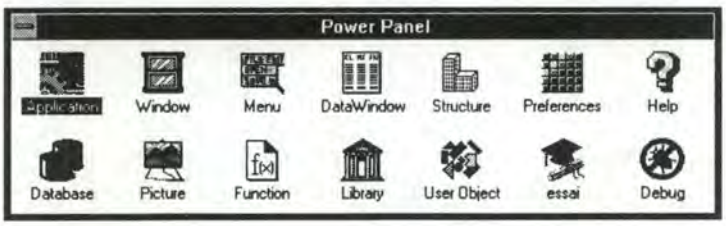

Figure 2.2 - Panneau principal de Powerbuilder

Si l'on désire créer une fenêtre, on lancera le module "Window" (- Fig. 2.2) et on se retrouvera dans un éditeur graphique de construction de l'interface (Fig. 2.3). La construction de celle-ci s'effectuera par manipulation directe en utilisant les OI offerts par la palette d'OI de l'éditeur se trouvant sur le côté gauche de l'éditeur.

[4] Powerbuilder est un produit développé par la société Powersoft.

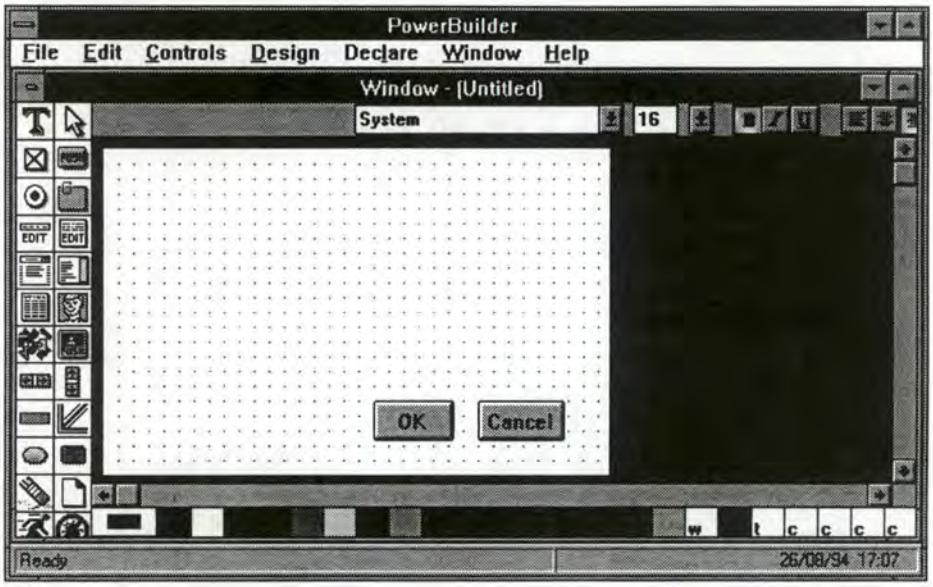

Figure 2.3 - Editeur graphique de construction de l'interface

Pour associer un script à un OI, on le sélectionne et on accède à un éditeur textuel via son icône dans l'éditeur graphique (🗋 - Fig. 2.3). La programmation événementielle est réalisée par l'intermédiaire de cet éditeur textuel et permet d'écrire les différents scripts associés aux objets (Fig. 2.4).

Figure 2.4 - Editeur textuel

Powerbuilder est connecté à un SGBD et offre un module d'interface avec celui-ci (🗊 - Fig. 2.2), ce qui permet de créer, manipuler et même administrer la base de données. Powerbuilder inclut également deux modules facilitant le développement: un module gérant les librairies de l'application (🏛 - Fig. 2.2) ainsi qu'un module de debogage (🐞 - Fig. 2.2).

3

Cycle de vie ou prototypage rapide ?

Les outils 4GL promeuvent le prototypage rapide et une plus grande implication de l'utilisateur dans le développement d'applications. Cette approche du processus de développement d'applications est différente de celle du cycle de vie classique. Sont-elles pour autant incompatibles ?

1. Cycle de vie

Depuis une vingtaine d'années, le développement de logiciel s'est basé sur un modèle communément appelé le modèle du cycle de vie. Bien qu'il peut être caractérisé différemment par certains auteurs, sa philosophie générale est universellement reconnue. Il guide le développement du logiciel à travers une stratégie linéaire comprenant un certain nombre d'étapes distinctes se succédant les unes aux autres :

- l'analyse des besoins consiste à identifier, à partir des besoins du client[1], les propriétés et fonctionnalités que devra supporter l'application;

- la spécification des besoins a essentiellement pour objectif de spécifier le résultat de l'étape précédente de manière précise et formelle;

- la conception consiste à produire des solutions qui satisfont les spécifications;

- l'implantation transforme le résultat de la conception dans un langage de programmation et permet ainsi d'exécuter l'application sur machine;

- la validation permet de vérifier que le système répond bien aux exigences initiales en testant l'application;

[1] Un client voulant développer une application.

- la maintenance modifie le système, durant sa vie opérationnelle, afin de corriger d'éventuelles erreurs, d'améliorer la performance ou encore d'introduire de nouvelles fonctionnalités.

Ce modèle a toutefois été critiqué et ce, essentiellement parce qu'il n'illustrait pas le rôle que jouait l'itération dans le processus de développement de logiciel [Som,92]. Parallèlement à cette critique du modèle du cycle de vie, l'approche du prototypage est devenue assez populaire.

Si le cycle de vie semble approprié pour le développement de systèmes de contrôle en temps réel ou le développement d'applications scientifiques, par exemple, où son approche rationnelle est la meilleure solution pour gérer la complexité de tels projets, il en va autrement pour une majorité d'autres applications [Hek,88], et notamment celles qui nous concernent, à savoir les applications de gestion hautement interactives.

2. Prototypage rapide

Il y a un certain nombre de mésententes sur le terme prototypage. Ceci est dû au fait qu'il y a différents types de prototypage et qu'il existe un éventail très large d'applications de cette technique. Précisons tout d'abord cette notion de prototypage et l'optique dans laquelle il est promu par les outils 4GL.

2.1. Définition

Le prototypage est la pratique de réaliser une première version d'un système, qui ne reflète pas nécessairement toutes les caractéristiques du système final, mais plutôt les plus importantes [Hek,88]. Ceci permet entre autres de tester une application. On exige généralement que cette étape coûte peu et soit rapidement réalisée, d'où le terme de prototypage rapide.

2.2. Deux Types de prototypage

S'il existe différents types de prototypage, on peut néanmoins classer de manière générale les prototypes selon deux types. Il y a d'une part, les prototypes réalisant une première version de l'application qui sont jetés après l'usage et d'autre part, les prototypes qui évoluent cycliquement vers le système final par un processus d'amélioration.

2.3. Différentes applications du prototypage

Le prototypage, selon l'optique choisie, pourra être appliqué à différentes phases du cycle de vie classique. Il pourra également en remplacer certaines, voire toutes. Il pourra par exemple être utilisé comme:

- outil d'aide à l'analyse et à la spécification des besoins. Le prototype peut assister l'analyste dans la recherche des besoins de l'utilisateur, il peut même remplacer le document de spécification;

- outil d'assistance à la conception. Il permettra de vérifier la faisabilité ou l'adéquation d'une solution de conception, ou encore de comparer des solutions alternatives;

- outil expérimental permettant d'étudier les facteurs humains de nouvelles applications afin d'aboutir à une interface utilisateur acceptable;

- méthode de développement dans laquelle le prototype évolue vers le système final.

C'est dans ces deux dernières applications que le prototypage est promu par les environnements de développement 4GL.

2.4. Prototypage de l'interface utilisateur

Le prototypage sera avant tout utilisé pour l'interface utilisateur. La conception d'une interface résulte plus d'un art que d'une science étant donné qu'il n'y a pas de procédures qui peuvent être suivies garantissant une bonne conception; on ne dispose que de quelques lignes de conduite. Ecrire des spécifications est beaucoup moins utile dans le cas de l'interface que dans le cas des fonctionnalités du système par exemple. Il est, en effet, nécessaire d'en visualiser l'apparence pour bien la concevoir.

De plus lorsque le système est destiné à être utilisé par plusieurs utilisateurs, il s'agit de concevoir une interface qui convienne le mieux à chacun. Il est très difficile de détecter les conflits sur papier et de trouver un compromis puisque des propriétés essentielles de l'interface telles que la convivialité et la facilité d'utilisation sont très subjectives et évaluables seulement lorsque le système est opérationnel. Le haut degré d'incertitude et la possibilité de changement sont de bonnes raisons pour considérer une conception expérimentale et adaptative de l'interface.

L'approche prototypage reconnaît ces difficultés et permet à la phase de conception de l'interface d'être un processus itératif impliquant une large participation de l'utilisateur. C'est dans cette perspective que des outils 4GL sont particulièrement bien adaptés à la conception de l'interface utilisateur d'une application.

Les méthodes traditionnelles du développement de logiciel sont, quant à elles, relativement inadaptées en ce qui concerne la conception de l'interface car d'une manière générale, elles n'impliquent pas assez l'utilisateur dans le développement. L'interface étant difficile à concevoir à l'avance, le programmeur la construit autour de son propre modèle conceptuel qui est, dans beaucoup de cas, différent de celui de l'utilisateur.

2.5. Outil 4GL et Prototypage

Le prototypage se concentre souvent uniquement sur l'aspect interface utilisateur de l'application [Hek,88]. Dans beaucoup de cas, elle est conçue et implémentée séparément de la partie fonctionnelle de l'application. Néanmoins une présentation statique n'est pas suffisante pour juger de l'état satisfaisant ou non du système. Intégrer le

prototypage des fonctions avec celui de l'interface permettrait à l'utilisateur de tester l'interactivité et d'observer ainsi les réponses de l'application [Mun,92].

Si les environnements de développement 4GL s'avèrent particulièrement adaptés pour le prototypage des interfaces utilisateur, ils le sont également en ce qui concerne l'aspect fonctionnel du système. La construction interactive de l'interface et les langages de quatrième génération qu'ils offrent favorisent, en effet, une telle approche. En outre, ces outils permettent le test de l'application pendant son développement, sans devoir effectuer des compilations et des liens de modules. Le programmeur peut ainsi facilement développer des portions de l'application et les tester rapidement.

En résumé, ces outils 4GL semblent donc spécialement adaptés à cette optique de prototypage. De plus, ce qui les rend particulièrement efficaces, c'est la haute fidélité du prototype qu'ils permettent de construire. Il est généralement admis qu'au plus celui-ci ressemble au système final, au plus il est bénéfique. La construction de l'interface étant effectuée par manipulation directe des véritables OI (et pas des représentations iconiques) et le langage de prototypage étant celui de développement, le prototype est à tout moment fidèle avec le système et évolue vers l'application finale.

3. Approche mixte

On a vu que l'approche classique du cycle de vie est de plus en plus remise en cause, notamment concernant les applications de gestion dans lesquelles l'interface prend une part de plus en plus importante. A l'opposé, l'approche du prototypage rapide supportée par les environnements de développement 4GL et dans laquelle le prototype évolue vers le système final semble tout à fait approprié au développement de telles applications mais souffre d'une faiblesse.

3.1. Faiblesse du prototypage rapide

Lorsque le prototypage commence trop tôt et lorsque l'analyse des besoins et l'analyse fonctionnelle ont été insuffisantes ou n'ont pas encore été effectuées, cela conduit à des applications qui se révèlent souvent médiocres [Lin,94][Mun,92]. Le développeur n'a pas une idée claire de ce qu'il doit concevoir dans son prototype et procède plus par intuition que rationnellement.

Sans une procédure de développement rigoureuse, le système ne peut être efficace. D'autre part, les systèmes réalisés avec ce type d'outils sont rapidement mis en utilisation et beaucoup d'entre eux ne respectent pas les besoins initiaux. Sans être passé auparavant par une phase de conception et sans avoir maintenu un minimum de documentation, la modification de ces systèmes s'avérera difficile.

Le problème d'une utilisation trop libérale de ce type d'outil est le manque de méthodologie de développement qu'il entraîne. Il peut entraîner les développeurs à

négliger des activités importantes du processus de développement comme l'analyse des besoins, la spécification des traitements ou celle des données.

3.2. Le prototypage rapide et le cycle de vie sont-ils complémentaires ?

Certains auteurs suggèrent la complémentarité du prototypage et des méthodes conventionnelles de développement plutôt que de les considérer comme des approches alternatives [Lin,94][Hek,88].

Afin de profiter pleinement de l'apport de cette nouvelle vague d'outils, il serait profitable d'insérer l'utilisation de ces outils 4GL dans un processus de développement assurant une compréhension de l'existant et une formalisation des besoins. On obtiendrait ainsi un cycle de vie commençant par une étape d'analyse conceptuelle sur base des besoins et adoptant ensuite une optique de prototypage pour le développement de l'application (Fig. 3.1).

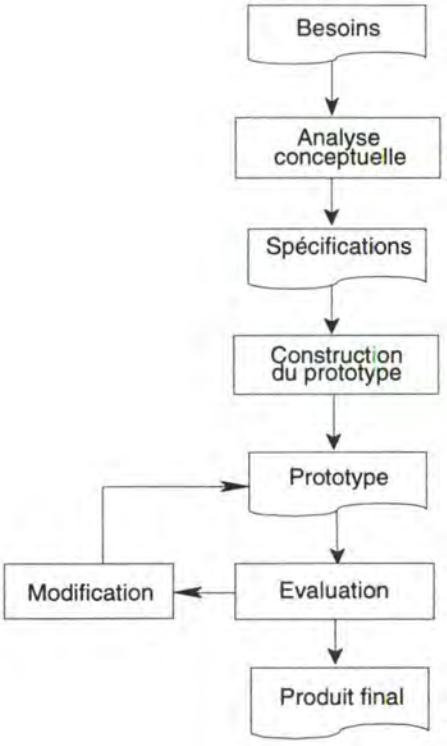

Figure 3.1 - Cycle de vie mixte

Si l'étape d'analyse fonctionnelle du processus de développement des applications de gestion est généralement réalisée à l'aide d'une méthode telle que IDA, il serait également intéressant, à ce stade du processus, de prendre en compte l'interface utilisateur afin de faciliter la construction du premier prototype. Compte tenu des caractéristiques des outils 4GL, il ne s'agira évidemment pas de s'étendre sur des spécifications détaillées de l'interface utilisateur, mais seulement de compléter les

modèles classiques utilisés pour modéliser la sémantique de l'application, en introduisant quelques concepts "orientés interface".

Cette approche mixte, combinant les avantages d'une analyse rigoureuse et d'un prototypage rapide, semble l'approche la plus conseillée pour réellement tirer profit des avantages qu'offrent les outils 4GL.

3.3. Génération automatique

Une des tendances actuelles est la dérivation systématique de l'application sur base des spécifications[2]. Il serait intéressant d'envisager le remplacement de l'étape de construction du premier prototype par une génération automatique de ce prototype sur base des informations sémantiques des spécifications (Fig. 3.2). Pour ce faire, un outil d'aide à la conception[3], supportant la phase d'analyse fonctionnelle, et des générateurs faisant la liaison entre cet outil et l'outil 4GL, devront être conçus.

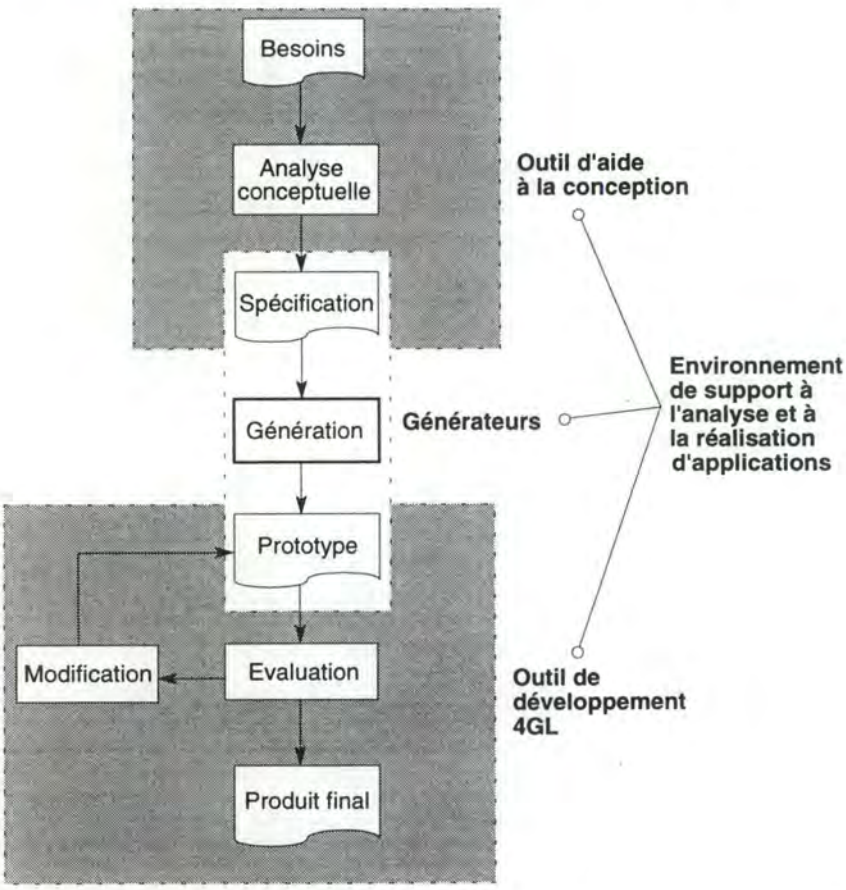

Figure 3.2 - Couplage d'un outil 4GL et d'un outil d'aide à la conception

[2] Cette tendance se justifie par le fait que l'application doit être une image des spécifications fonctionnelles la plus fidèle possible.

[3] Outil que l'on appelle communément outil CASE.

Le couplage, via des générateurs, d'un outil d'aide à la conception et d'un outil 4GL fournira un environnement intégré de support à l'analyse et à la réalisation d'applications que l'on appellera environnement d'aide au développement d'applications de gestion; celui-ci couvre la totalité du cycle de vie (Fig. 3.2). Ce mémoire se propose d'envisager une telle perspective.

Nous allons, dans la section suivante, définir de manière générale cet environnement et les différents aspects à prendre en considération en vue de sa réalisation.

4

Proposition d'un environnement d'aide au développement des applications de gestion

L'environnement d'aide au développement que nous proposons consiste en un couplage d'un outil CASE et d'un outil 4GL pour supporter un cycle de vie mixte; ce couplage sera réalisé au travers de générateurs. Ces derniers se basent sur les liens qui existent entre les composants d'une application et les spécifications fonctionnelles.

1. Couplage d'un outil CASE et d'un outil 4GL

Ce mémoire se propose d'étudier le couplage d'un outil CASE et d'un outil 4GL relié à un SGBD. Le but est d'automatiser et d'accélérer ainsi le développement d'applications de gestion à interface graphique fonctionnant en mode Client/Serveur de données.

L'avantage du couplage est d'automatiser la construction de la BD et d'accélérer le processus de développement de l'application. La génération possède également d'autres avantages.

Les méthodes classiques de développement de logiciel ne traitent pas l'aspect interface utilisateur et il est dès lors difficile de passer d'une phase d'analyse fonctionnelle à une phase de conception de l'interface. Le passage d'une étape à l'autre va exiger un certain effort. Le concepteur devra accéder aux détails des modèles sémantiques afin de choisir les OI.

Une génération automatique de l'interface utilisateur sur base des informations présentes dans les spécifications conceptuelles permettrait d'une part, de faciliter la

liaison entre l'interface et la sémantique de l'application et d'autre part, d'appliquer automatiquement les conventions concernant la sélection et le placement des OI.

2. Composants d'une application 4GL

La première étape de notre travail va consister à sélectionner des modèles adaptés aux applications réalisées avec des outils 4GL (reliés à un SGBD).

Nous avons vu dans un chapitre précédent les caractéristiques des outils 4GL et le type de programmation qu'ils impliquent. Identifions maintenant, les différents composants de l'application à prendre en compte en vue de leur génération.

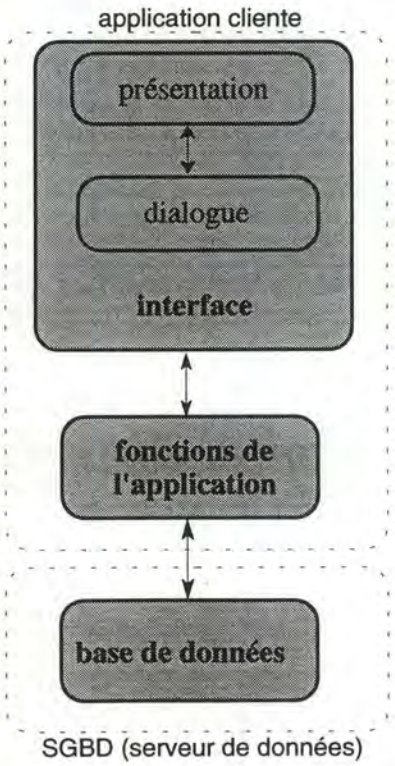

Figure 4.1 - Composants d'une application 4GL

Etant donné que nous nous situons dans un contexte d'interface graphique, la présentation est constituée d'un ensemble d'OI que l'utilisateur manipule à l'aide du clavier et de la souris. Le dialogue animera, quant à lui, la présentation et répondra aux requêtes formulées sur les OI en effectuant des appels aux fonctions de l'application. Dans les outils 4GL, le dialogue est implémenté au travers d'une programmation événementielle.

Les fonctions de l'applications sont séparées de l'interface et stockées dans une librairie.

La base de données relationnelle sera, quant à elle, gérée par un SGBD, auquel est relié l'outil 4GL, qui se chargera des accès physiques à la base de données.

L'utilisateur, à travers la présentation, va demander à l'application d'exécuter un certain nombre de fonctions. Le dialogue répondra aux actions de l'utilisateur en faisant appel aux fonctions de l'application; ces dernières créant des requêtes sur la base de données au SGBD. Ce dernier renverra le résultat à la fonction et le dialogue se chargera de présenter le résultat à l'utilisateur via la présentation.

Le composant base de données se trouvera sur le serveur; l'interface utilisateur, ainsi que les fonctions de l'application constitueront l'application cliente qui se trouvera sur un poste de travail connecté au serveur de données (Fig. 4.1).

3. Liens entre les composants d'une application et les spécifications fonctionnelles

Nous avons vu qu'il était impératif de faire une analyse complète et rigoureuse d'une application. Pour guider le concepteur dans cette analyse, nombre de méthodes de conception ont été élaborées. Etant donné que nous nous concentrons sur les applications de gestion, il semble tout indiqué de profiter de la méthode IDA [Bod-Pig,89]. Elle donne, en effet, un cadre méthodologique de conception des applications de gestion dont nous utiliserons et étendrons quelques concepts.

La méthode IDA traite essentiellement des aspects fonctionnels de la conception que l'on appellera analyse fonctionnelle. Elle aboutira à une solution conceptuelle qui devra satisfaire les besoins de l'organisation préalablement analysés et ceci, indépendamment des moyens d'implémentation. Cette solution, une fois spécifiée et jugée correcte, sera la fondation de la suite du développement. L'application implémentée ne sera tout compte fait qu'une image opérationnelle de spécifications fonctionnelles, c'est pourquoi l'analyse fonctionnelle est si importante.

Afin de spécifier clairement cette solution, un certain nombre de modèles seront utilisés. Ainsi, l'analyse fonctionnelle aboutira à la description des informations utilisées, des traitements qui y sont associés ainsi que leurs conditions de déclenchement et ceci, à l'aide des modèles qui sont à la disposition des concepteurs.

Parmi les modèles de cette méthode, trois d'entre eux nous intéressent particulièrement: le modèle entité-association, celui de la statique et celui de la dynamique des traitements[1]. Le premier sera utilisé pour structurer les données. La statique des traitements sera utilisée pour les différentes phases de l'application et leurs

[1] Un traitement permet de représenter les fonctionnalités d'une application à différents niveaux de détails. Chaque niveau traduisant une préoccupation particulière de l'organisation [Bod-Pig,89]. Dans la nomenclature standard, seuls les traitements interactifs de type "phase" et "fonction" et leurs critères d'identification nous seront utiles. Cela nous permettra ainsi d'identifier, dans un premier temps, les différentes phases d'une application et les fonctions qui les composent dans un deuxième temps.

fonctions. Quant à la dynamique des traitements, nous en utiliserons un dérivé: le graphe d'enchaînement de fonctions d'une phase (GEF).

La méthode IDA se base sur une logique de fonctionnement et ne tient pas compte de l'interface utilisateur. En effet, on y met l'accent sur le caractère fonctionnel des applications et pas sur son interfaçage. De plus, elle est totalement indépendante de toute technique d'implémentation. Or, si l'on désire générer l'interface utilisateur dans un outil 4GL, il faudra tenir compte de l'aspect interface utilisateur et de la programmation événementielle utilisée par les outils 4GL dans les modèles utilisés. Les modèles initiaux de la méthode seront donc modifiés et étendus.

Etablissons maintenant de manière générale les correspondances entre les modèles des spécifications fonctionnelles et les différents composants de l'application identifiés précédemment. Nous verrons ainsi sur quel(s) modèle(s) se base la dérivation des différents composants. Nous expliquerons brièvement la contribution de chaque modèle au travail de génération et plus en détail dans le reste du mémoire.

Le modèle entité-association est utilisé pour modéliser les données de l'application. La base de données sera donc dérivée du schéma entité-association (Fig. 4.2(1)). Il est clair qu'il ne sera pas possible de générer une base de données à partir du schéma entité-association étant donné qu'aucune technologie ne gère des données structurées en entité-association. Il faudra donc transformer ce schéma en une autre structuration des données pour pouvoir utiliser un SGBD et gérer les données. En l'occurrence, il faudra le transformer en un schéma relationnel si l'on considère les SGBD relationnels (SGBDR)[2].

Les fonctions de l'application sont, quant à elles, modélisées à l'aide du modèle de la statique des traitements (Fig. 4.2(2)). Etant donné que nous n'avons pas de langage formel de description des règles de traitement, il ne nous sera pas possible de générer des fonctions directement exécutables. D'autant plus que ces traitements seront spécifiés en fonction du schéma entité-association et non en fonction d'une quelconque base de données relationnelles[3]. Nous envisagerons plutôt une génération de la signature des différentes fonctions, autrement dit des fonctions exécutables, mais vides. Il ne resterait plus qu'à remplir le corps des fonctions avec des instructions sur base des règles de traitement.

---

[2] C'est actuellement la technologie de gestion de données la plus répandue. Parmi les SGBDR, on trouve Oracle, Informix, Ingres, ...

[3] Le fait que les règles de traitement soient en fonction du schéma entité-association complexifie encore plus le travail de génération. Si l'on désire, en effet, avoir des fonctions exécutables, il faudrait modifier toutes les requêtes de données en fonction des transformations réalisées sur le schéma entité-association afin d'obtenir des requêtes de données relationnelles.

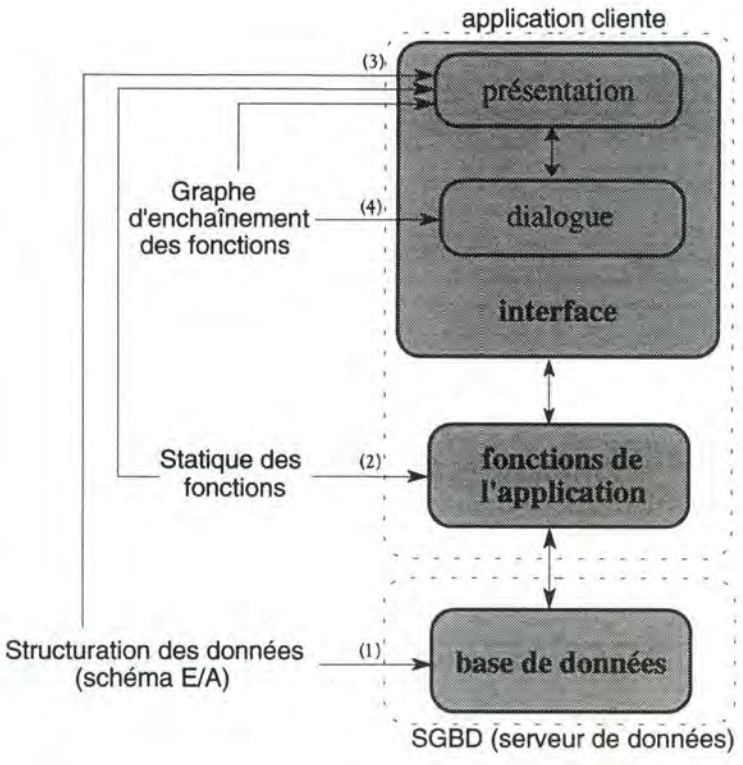

Figure 4.2 - Liens entre les composants d'une application et ses spécifications fonctionnelles

Quant à l'interface, c'est à dire la présentation et le dialogue, elle sera dérivée sur base des trois modèles.

La dérivation de la présentation utilisera les trois modèles, c'est à dire le GEF, la statique des fonctions et le schéma entité-association (Fig. 4.2(3)). Le GEF et la statique nous permettrons notamment d'identifier les données sémantiques d'entrée et sortie devant être saisies ou affichées dans l'interface. Le schéma entité-association fournira le type de ces données ainsi que certaines autres informations qui seront utilisées dans l'étape de sélection des OI.

La génération du dialogue se basera sur le GEF qui fixe précisément les conditions de déclenchement et d'enchaînement des fonctions au sein d'une phase (Fig. 4.2(4)). Le dialogue permettra de déclencher ou non les fonctions que l'utilisateur veut exécuter. Les objets de la présentation pourront également être modifiés par le dialogue en fonction de l'état de l'application.

4. Environnement d'aide au développement d'applications

L'environnement tel que nous l'envisageons se compose de trois outils principaux qui permettent de supporter chacun une partie du cycle mixte de développement que nous avions présenté. Un outil d'aide à la conception pour supporter l'analyse fonctionnelle,

un outil de génération pour supporter la phase de génération et un outil de prototypage pour supporter la phase de prototypage (Fig. 4.3).

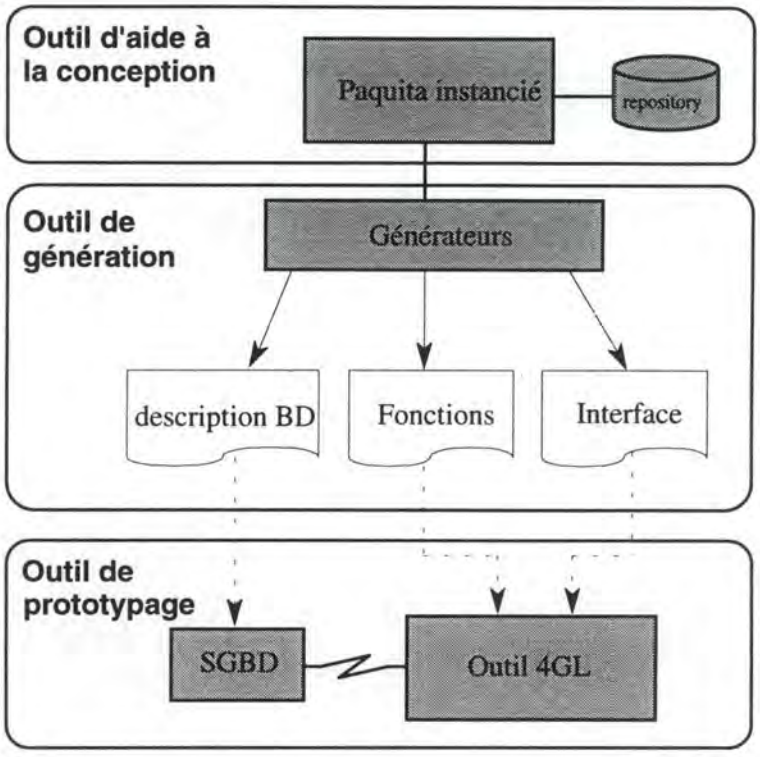

Figure 4.3 - Architecture de l'environnement d'aide au développement d'applications informatiques

L'outil d'aide à la conception permettra de stocker toutes les spécifications fonctionnelles dans un repository[4]. Toutes ces informations seront utilisées par l'outil de génération qui produira un certain nombre de fichiers opérationnels. Ceux-ci pourront alors être récupérés par l'outil de prototypage (Fig. 4.3).

4.1. Outil d'aide à la conception

L'outil d'aide à la conception sera réalisé grâce à l'outil générique Paquita[5]. Celui-ci permet de faire une instanciation des modèles de spécification d'une méthode d'analyse. Le travail de méta-modélisation consiste à définir la sémantique des modèles en question ainsi que l'expression textuelle et graphique de cette sémantique. Ce travail sera réalisé à l'aide de l'instanciateur de Paquita[6].

---

[4] Un repository est une base de données de spécification. C'est donc une base de données où sont stockées toutes les informations utiles au développement d'une application.

[5] Paquita est un outil développé par la société Metsi.

[6] Les définifions seront réalisées grâce aux différents concepts que l'instanciateur met à disposition. Pour de plus amples informations voir le manuel [Paq,89]. Ce dernier expose les différents concepts utilisés dans la méta-modélisation ainsi qu'une démarche de méta-modélisation.

Une fois ce travail réalisé, on aura à sa disposition l'outil Paquita instancié avec un repository qui accueillera les spécifications des différents modèles instanciés. Il sera alors possible de rentrer les spécifications de son application à l'aide des différents modèles. Paquita instancié sera l'outil d'aide à la conception de l'environnement que nous proposons.

4.2. Outil de génération

L'outil de génération se composera d'un certain nombre de générateurs[7] qui permettront d'obtenir une description exécutable de la base de données, la signature des différentes fonctions et les différents écrans minimaux opérationnels[8] récupérables dans l'outil 4GL.

Ces générateurs trouveront toutes les informations nécessaires dans le repository de l'outil d'aide à la conception et y stockeront un certain nombre d'informations. Les générateurs se baseront sur les liens que nous avons identifié entre les spécifications fonctionnelles et les composants d'une application, pour générer la base de données, ses fonctions et son interface utilisateur. Une fois les textes de fonction et d'interface générés, le module de génération se chargera de les placer dans le répertoire de l'outil 4GL. Quant à la description de la base de données, une fois qu'elle sera générée, il faudra l'exécuter pour créer la base de données dans le SGBD.

4.3. Outil de prototypage

L'outil de prototypage ne sera autre que l'outil 4GL utilisé pour développer l'application. Il permettra dans un premier temps de compléter l'application en remplissant le corps des fonctions grâce à son éditeur textuel. On obtient ainsi une application opérationnelle. Ensuite, le prototypage proprement dit pourra commencer. Il faut bien évidemment que l'outil 4GL soit suffisamment ouvert pour pouvoir récupérer les différents éléments de l'application. Powerbuilder, par exemple, semble tout à fait approprié à cette tâche puisqu'il est possible d'y importer des fichiers texte de définition des différents éléments de l'application sous un certain format. De plus, c'est un environnement très bien conçu où tout a été fait pour faciliter le travail de programmation.

4.4. Utilisation de l'environnement

On peut expliquer succinctement comment le développeur utilisera l'environnement, et ceci en considérant les trois composants d'une application: la base de données relationnelle, les fonctions et l'interface utilisateur.

---

[7] Algorithmes de génération implémentant un certain nombre de règles de dérivation des spécifications fonctionnelles.

[8] Par "minimal opérationnel", on entend des écrans qui fonctionnent mais qui doivent encore être améliorés afin de les adapter au mieux à l'utilisateur.

En ce qui concerne les données de l'application, le développeur pourra en spécifier la structuration à l'aide du modèle entité-association instancié dans l'outil d'aide à la conception. Il lui faudra ensuite opérer une génération de la base de données relationnelle sur laquelle l'application travaillera. Il pourra éventuellement la retravailler soit dans le SGBDR, soit à travers l'outil 4GL[9].

Quant aux fonctions de l'application, il pourra en donner une description dans l'outil d'aide à la conception. Il lui faudra alors opérer la génération de la signature des fonctions. Celles-ci seront alors stockées dans le répertoire de l'outil 4GL et il pourra en compléter le corps dans l'outil 4GL.

Le développeur complètera les spécifications fonctionnelles en définissant le graphe d'enchaînement des fonctions. Ainsi, une fois les données et les fonctions entièrement spécifiées, le développeur pourra opérer la génération d'une interface opérationnelle de l'application. Celle-ci sera aussi stockée dans le répertoire de l'outil 4GL et il pourra ainsi la tester et l'améliorer dans l'outil 4GL

[9] Toutes les transformations possibles et imaginables ne peuvent être réalisées à travers l'outil 4GL. Il ne pourra opérer que quelques transformations élémentaires telle qu'ajouter un index.

Partie 2

Modélisation et stratégie de génération

5. Les données de l'application
6. L'application cliente
7. Définition du langage DSL-II
8. Format des fichiers Powerbuilder à générer

Maintenant que les principes de l'environnement de développement d'applications de gestion ont été exposés, nous pouvons en détailler les différents aspects pour les différents composants d'une application: les données, les fonctions et l'interface utilisateur. Comme nous nous intéressons aux applications de type Client/Serveur de données, nous aborderons ces différents composants regroupés de la manière suivante: les données de l'application tout d'abord et l'application cliente composée des fonctions et de l'interface ensuite. Nous y aborderons les modèles nécessaires à la génération des différents composants.

Nous définirons un langage de spécification que l'on appellera DSL-II qui est un sous-ensemble du langage DSL que nous avons étendu afin de stocker dans le repository de l'outil d'aide à la conception tous les éléments nécessaires à la génération. Nous terminerons en présetant pour un outil particulier, Powerbuilder, le format des fichiers à générer.

5

Les données de l'application

Pour générer les données de l'application de la meilleure façon qui soit, il faut pouvoir les définir avec le plus de rigueur possible, et ceci est rendu possible grâce au modèle entité-association. Nous présenterons tout d'abord succinctement ce modèle, nous expliquerons ensuite l'approche de génération de la base de données que nous suivons.

1. Structuration des données - modèle entité association

Une application de gestion a besoin d'un certain nombre d'informations pour fonctionner puisque son objectif est de traiter de façon efficace les informations de l'organisation. Etant donné qu'une application repose sur la qualité de ces informations, il faut absolument qu'elles soient définies rigoureusement pour que leur utilisation soit la meilleure possible. C'est parce que la structuration des informations est capitale que des modèles de structuration de données ont été réalisés[1].

1.1. Modèle entité-association

Le modèle entité-association permet de définir la structuration et la sémantique des données d'une application, et celui présenté dans [Bod-Pig,89] trouve son origine dans un certain nombre de travaux antérieurs. Les concepts utilisés pour exprimer la sémantique des données sont les concepts d'entité, d'association, d'attribut et de contrainte d'intégrité [Bod-Pig,89].

---

[1] Une donnée est la représentation des propriétés d'un concept, d'un objet ou d'un fait; une information est, quant à elle, la signification attachée aux données [Bod-Pig,89].

Le modèle entité-association utilisé est un modèle non-étendu [Bod,89] et nous prenons le parti de ne pas prendre en compte un certain nombre de concepts qui permettent d'enrichir la sémantique des données telle que la relation Is-A. Ceci parce qu'il n'est pas obligatoire de les utiliser pour arriver à une spécification correcte des données. Ce sont des concepts qui facilitent la spécification mais dont il est toujours possible d'avoir un équivalent avec les concepts de base. Il n'est évidemment pas inutile de les avoir à sa disposition mais cela ne changera rien à la génération de la base de données.

Une fois les données spécifiées à l'aide du modèle entité-association, il faudra mettre ce schéma sous forme canonique, c'est à dire vérifiant un certain nombre de règles de complétude et de cohérence [Bod-Pig,89]. Les règles de complétude doivent permettre de vérifier que chaque objet du schéma entité-association possède bien toutes les propriétés prévues dans le modèle entité-association pour le type d'objet dont il est une instance. Les règles de cohérence permettront d'obtenir un schéma entité-association dont on aura éliminé les contradictions dans les spécifications et dont on aura éliminé ou contrôlé la redondance. De cette manière, on aura un schéma entité-association aussi significatif et aussi stable que possible.

1.2. Représentation graphique

Un support graphique est une chose à ne pas négliger dans un modèle dans la mesure où cela permet une excellente visualisation des concepts du modèle. Un des avantages du modèle entité-association est qu'il offre justement une représentation graphique très significative qui permet de bien mettre en évidence la structuration des données. Ce n'est évidemment qu'un support visuel, la représentation doit être complétée par la spécification précise de chaque concept représenté.

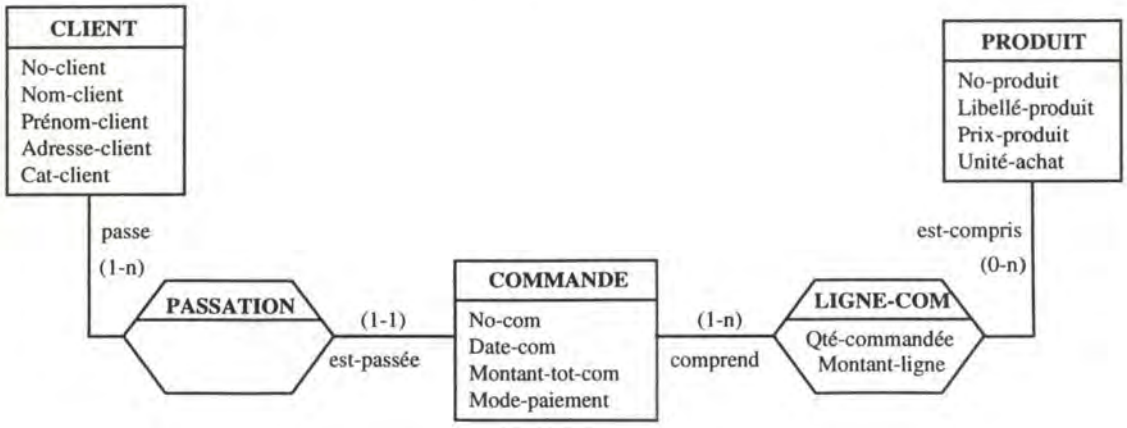

Figure 5.1 - Schéma entité-association

La Fig. 5.1 montre un exemple de schéma entité-association où sont représentés les différents concepts de base. Dans celui-ci, apparaissent très bien les types d'entité CLIENT, COMMANDE et PRODUIT avec leur attributs, ainsi que les types

d'association Passation et Ligne-com qui les relient. Les rôles joués par les différents types d'entité ont chacun un nom et leur connectivité est mises entre parenthèses.

2. Génération de la base de données

Maintenant que la spécification des données à l'aide d'un schéma entité-association sous forme canonique est rendue possible, on peut envisager la génération d'une base de données sur base de ce schéma conceptuel. Ces spécifications se trouvant dans le repository de l'outil d'aide à la conception, il faudra donc tout d'abord extraire le schéma entité-association pour ensuite le traduire en une base de données.

Etant donné que ce domaine est l'objet de beaucoup de recherches et que des outils existent déjà, nous nous contenterons de présenter les points importants qui doivent être considérés lorsque l'on envisage ce problème.

2.1. SGBD relationnel comme cible de la génération

Dans notre environnement, l'outil cible est un outil 4GL et la plupart du temps, ces outils sont rattachables à un SGBD relationnel (SGBD-R). Dans un SGBD-R, les données sont présentées sous forme de tables, celles-ci étant constituées d'un certain nombre de colonnes; une base de données relationnelle est ainsi constituée d'une collection de tables. Il faudra donc générer une base de données relationnelle, or nous n'avons à notre disposition qu'un schéma entité-association. Celui-ci n'est jamais qu'une description abstraite de la base de données, c'est à dire non opérationnelle[2]. Il faudra donc traduire, transformer le schéma entité-association en un schéma relationnel; une transformation permet de supprimer une construction indésirable tout en conservant la sémantique que ce schéma contient[3].

Un schéma entité-association permet, en fait, une description des données plus riche qu'un schéma relationnel; tous les concepts relationnels se retrouvent, en effet, dans un schéma entité-association. Autrement dit, un schéma relationnel est une réduction du schéma entité-association. La transformation reviendra donc à éliminer un certain nombre de concepts du schéma entité-association qui ne sont pas conformes au

[2] Le but des modèles entité-association a été d'offrir des concepts de description d'une base de données moins technique et sémantiquement plus riche que les modèles offerts par les SGBD-R [Hai,89].

[3] Une transformation de schéma est un opérateur par lequel une structure de données A est remplacée par une autre structure de données B qui est sémantiquement équivalente à la structure A [Hai,86]. Considérant le modèle entité-association comme un formalisme de description abstraite d'une base de données, il est possible de définir des transformations de ce schéma.

relationnel, c'est à dire les types d'association, les attributs décomposables et les attributs répétitifs [Hai,89][4].

2.2. Deux approches de génération

Deux approches existent pour générer une base de données relationnelle: la génération assistée de la base de données d'une part, et la génération automatique d'autre part. Dans le premier cas, le concepteur intervient dans le processus de génération alors que dans le deuxième cas, il n'a aucune emprise sur la génération.

2.2.1. Génération assistée d'une base de données

La première approche se fonde sur une démarche transformationnelle en 3 étapes selon le processus suivant : construction du schéma conceptuel des données, transformation de ce schéma en un schéma logique qui sera lui-même transformé en un schéma physique [Hai,86][5]. La première étape de cette approche est réalisée à l'aide du modèle entité-association. La deuxième étape consiste a transformer le schéma entité-association en éliminant les types d'application complexes pour obtenir un schéma ne contenant plus que des types d'association binaires et en ajoutant ensuite, les clés d'accès aux données[6]. Cela permet donc de produire un schéma logique. Ce schéma permettra de produire directement le texte DDL (Data Description Language) de définition de la base de données en faisant un certain nombre de transformations élémentaires; les clés d'accès seront implémentées par des index.

Une telle approche part du principe qu'il n'est pas possible de concevoir directement une base de données efficace une fois le schéma entité-association entièrement spécifié. Il faut que la génération soit la plus souple possible afin d'obtenir la meilleure définition possible de cette base de données. La transformation se fait interactivement en précisant un certain de nombre de critères qui auront une influence sur l'efficacité de la base de données[7]. Cette approche n'est pas simple et prend donc assez bien de temps pour générer une base de données.

---

[4] En réalité, pour passer du schéma conceptuel au schéma relationnel, un certain nombre de transformations élémentaires peuvent être utilisées afin d'éliminer chaque fois un concept non-relationnel (transformations d'un type d'entité, d'un type d'association, d'un attribut, ...).

[5] Cette démarche est basée sur les trois niveaux de conception suivants: conceptuel, logique et physique.

[6] Un schéma entité-association binaire permet d'avoir une continuité dans le formalisme et est bien adapté à la définition des accès. On a ainsi un schéma logique basé sur le formalisme entité-association enrichi de concepts propres à décrire les accès aux données [Hai,91].

[7] Pour ce faire, le concepteur utilise les primitives de transformation à sa disposition lui permettant de rendre un schéma relationnel plus efficace.

2.2.2. Génération automatique d'une base de données

Cette approche prône une dérivation directe de la base de données sur base du schéma conceptuel à l'aide de quelques règles très simples. C'est ce qu'on appelle une approche "draw-and-generate" selon laquelle le développeur n'a aucune emprise sur le processus de génération[8]. C'est en quelque sorte une particularisation de l'approche précédente dans la mesure où on y utilise quelques transformations élémentaires qui étaient à la disposition de l'approche précédente; mais dans ce cas-ci, elles sont réalisées automatiquement. En fait, la génération consiste à exécuter un programme implémentant une stratégie de transformation qui réalise un certain nombre de transformations élémentaires pour produire une base de données relationnelles. Autrement dit, c'est le programme qui guide lui-même les transformations selon sa stratégie; quant aux accès, il produira un ensemble minimal de clés d'accès.

Si l'on n'est pas trop exigeant du point de vue des performances ou si la base de données n'est pas très volumineuse, cette approche est tout à fait valable. Même si cela n'est pas toujours évident, il est toujours possible d'améliorer les performances physiques de la base de données après sa création grâce au SGBD-R[9] ou au travers de l'outil 4GL[10].

2.3. Approche de génération adoptée

Les 2 approches ont leurs avantages et inconvénients. La génération automatique est rapide mais rigide, elle ne produit pas toujours une base de données efficace. La génération assistée est plus longue, mais peut produire une base de données plus efficace.

Etant donné que nous nous situons dans une optique de prototypage, la deuxième approche sera celle adoptée pour notre environnement. On part donc du principe que l'on génére directement une première base de données et si celle-ci s'avère trop peu performante, il sera demandé au DBA de modifier certains paramètres afin d'en améliorer l'utilisation. Si au contraire, elle s'avère correctement utilisable, on peut la laisser telle quelle.

On aura donc un générateur relationnel qui, sur base du schéma entité-association sous forme canonique, génère la structure d'une base de données relationnelle dans le SGBD-R cible.

---

[8] Un outil CASE tel que OBLOG Case suit cette approche. Une fois les spécifications complètes, il est possible de générer totalement la définition d'une base de données.

[9] Les améliorations peuvent être réalisées par le DBA (DataBase Administrator) de l'application, celui-ci s'occupe de la gestion des tables de la base données. Il est sensé connaître suffisamment le SGBD utilisé pour améliorer la base de données en question. Il pourra initialiser quelques paramètres physiques tels que la manière de stocker les tables, le taux de remplissage des pages, la taille des pages, d'éventuels index pour améliorer la performance lors de certains accès, ...

[10] Les modifications permises au travers de l'outil 4GL sont limitées: rajout d'un index, modification de la structure d'une table, gestion des accès sur les tables.

2.4. Générateur relationnel

L'objectif premier du générateur relationnel est évidemment de générer une base de données relationnelle, mais il faut en plus, stocker le schéma relationnel dans le repository de l'outil d'aide à la conception ainsi que les correspondances entre celui-ci et le schéma conceptuel [Bod-Bea,88]. Il serait intéressant de stocker un schéma relationnel standard, c'est à dire indépendant d'un quelconque SGBD-R, et d'établir les correspondances avec le schéma entité-association[11]. Sur base de ce schéma relationnel, il sera alors possible de générer le texte DDL de description de la base de données et il ne restera plus qu'à l'exécuter pour créer la base de données.

La génération de la base de données relationnelles se fera donc en deux étapes principales, une première pour transformer le schéma entité-association en un schéma relationnel standard et une deuxième pour créer les tables de la base de données :

- Extraction du schéma entité-association sous forme canonique du repository de l'outil d'aide à la conception, transformation de ce schéma en un schéma relationnel standard, stockage de ce dernier dans le repository[12] et des correspondances entre les concepts du schéma entité-association et relationnels (Extraction-Transformation-Stockage);
- Génération du code DDL conforme au SGBD-R cible et exécution de ce code dans le SGBD-R en question (Génération-Exécution du code DDL).

2.4.1. Extraction-Transformation-Stockage

Nous verrons dans un premier temps ce que nous retiendrons dans l'extraction du schéma entité-association. Nous aborderons ensuite l'aspect transformation, c'est à dire les transformations élémentaires utilisées. Nous pourrons alors présenter un algorithme de transformation qui ne tiendra pas compte du stockage. Ensuite, nous passerons aux informations à stocker et nous proposerons quelques modifications dans l'algorithme de transformation.

A. Extraction du schéma entité-association

La génération commencera tout d'abord par l'extraction du schéma entité-association du repository, c'est à dire les spécifications du schéma utiles pour la génération de la base de données. Dans ces spécifications, seules les contraintes d'intégrité informelles ne seront pas utilisées. Ceci parce qu'actuellement, peu de SGBD-R supportent de telles contraintes; de plus, il n'y a pas de syntaxe relationnelle standard à ce sujet[13]. Ces contraintes d'intégrité devront donc être implémentées plus tard. Une fois les spécifications extraites, il faudra réaliser la transformation en un schéma relationnel à l'aide d'un algorithme de transformation.

---

[11] Nous verrons plus loin ce que nous entendons par schéma relationnel standard.

[12] Quand on parlera de repository, il s'agira de celui de l'outil d'aide à la conception.

[13] Des tentatives de standardisation des contraintes d'intégrité ont été faites [Date,90].

L'extraction sera réalisée grâce à une procédure d'extraction de spécifications réalisée par l'outil d'aide à la conception. Il faudra définir cette procédure et elle aura pour résultat un fichier dont on aura spécifié le format que l'algorithme de transformation récupérera.

B. Algorithme de transformation

Pour effectuer une transformation d'un schéma conceptuel en un schéma relationnel, l'algorithme de transformation utilisera quelques-unes des transformations élémentaires classiques [Hai,86] que nous présentons dans ce qui suit.

B.1. Transformations élémentaires utilisées

Les transformations élémentaires que l'algorithme utilisera sont celles d'un type d'association complexe, d'un attribut décomposable, d'un attribut répétitif, d'un type d'association "many-to-many" et d'un type d'association simple. Nous les illustrerons en les appliquant au schéma entité-association de la Fig. 5.1 quand cela sera possible.

- Transformation d'un type d'association complexe :

Elle consiste à éliminer un type d'association complexe, c'est à dire un type d'association qui a un degré supérieur à 2 et/ou plusieurs attributs. Cette transformation consiste à remplacer le type d'association en question par un type d'entité et permet ainsi d'obtenir un schéma entité-association qui ne possède plus que des associations simples, c'est à dire binaires (degré égal à 2) et ne possédant pas d'attributs. Par exemple, dans le schéma entité-association de la Fig. 5.2 , le type d'association LIGNE-COM est complexe puisqu'il contient deux attributs. Il est donc remplacé par le type d'entité LIGNE-COM (Fig. 5.2).

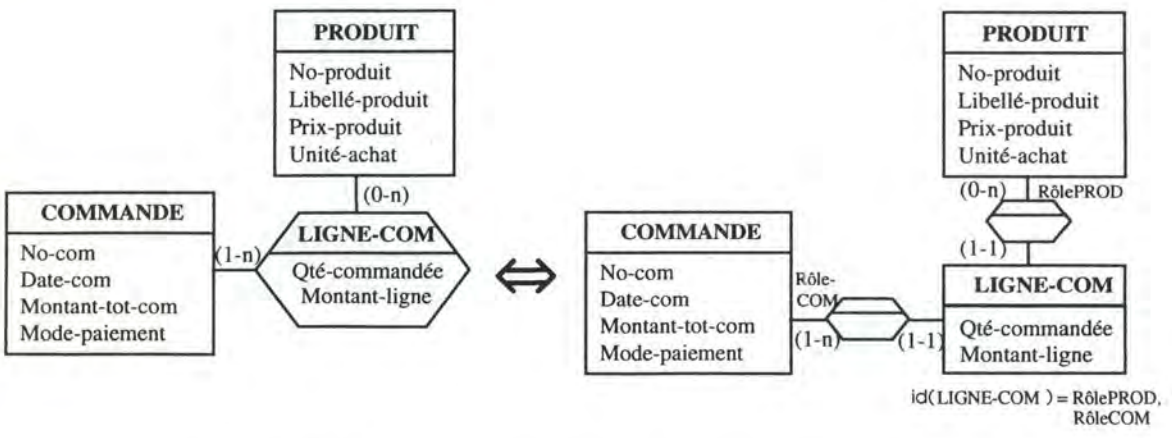

Figure 5.2 - Elimination d'un type d'association complexe

- Transformation d'un attribut décomposable :

Elle permet d'éliminer un attribut décomposable en le remplacant par tous ses composants. Par exemple, si l'attribut Adresse-client de CLIENT était décomposable en

Rue, Numéro-rue, Localité et Code-postal, on le remplacerait par ces 4 attributs-ci (Fig. 5.3).

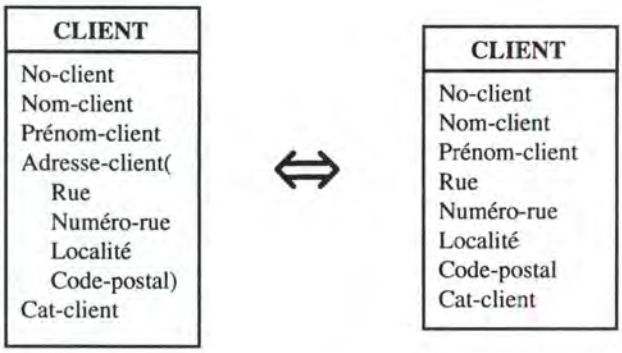

Figure 5.3 - Elimination d'un attribut décomposable

- Transformation d'un attribut répétitif :

Cette transformation permet d'éliminer un attribut répétitif. Cet attribut peut être exprimé à l'aide d'un type d'entités relié au celui qui le contient par un type d'association. Cette transformation consiste donc à remplacer l'attribut répétitif par un type d'entité portant le même nom et ayant pour seul attribut l'attribut en question; le nouveau type d'entité sera relié au type d'entité modifié par un type d'association simple. Le nouveau type d'entité sera identifié par son attribut et le rôle joué par le type d'entité modifé.

Par exemple, si le type d'entité CLIENT avait pour attribut répétitif Téléphone, alors on le remplacerait par un type d'entité TELEPHONE relié à CLIENT (Fig. 5.4). Le type d'entité TELEPHONE est identifié par l'attribut Téléphone et RôleCLIENT.

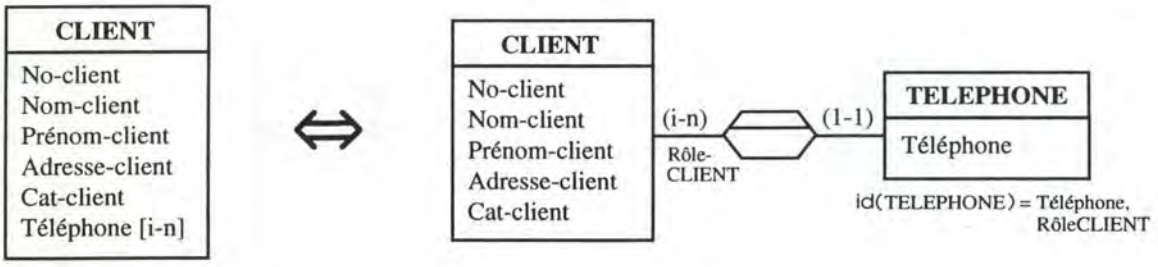

Figure 5.4 - Elimination d'un attribut répétitif

- Transformation d'un type d'association "many-to-many" :

cette transformation consiste à éliminer les types d'associations "many-to-many", c'est à dire un type d'association simple dont les rôles joués par les 2 types d'entité reliés ont une connectivité (1-n). Cette transformation consiste à remplacer le type d'association en question par un type d'entité, celui-ci sera relié aux 2 types d'entité de départ par un type d'association simple. Le nouveau type d'entité est identifié par les rôles joués par les 2 types d'entité de départ.

Par exemple, le type d'association reliant les types d'entité ENTITE-A et ENTITE-B est remplacé par le type d'entité ENTITE-AB, celui-ci est identifié par les rôles RôleA (joué par l'entité ENTITE-A) et RôleB (joué par l'entité ENTITE-B)(Fig. 5.5).

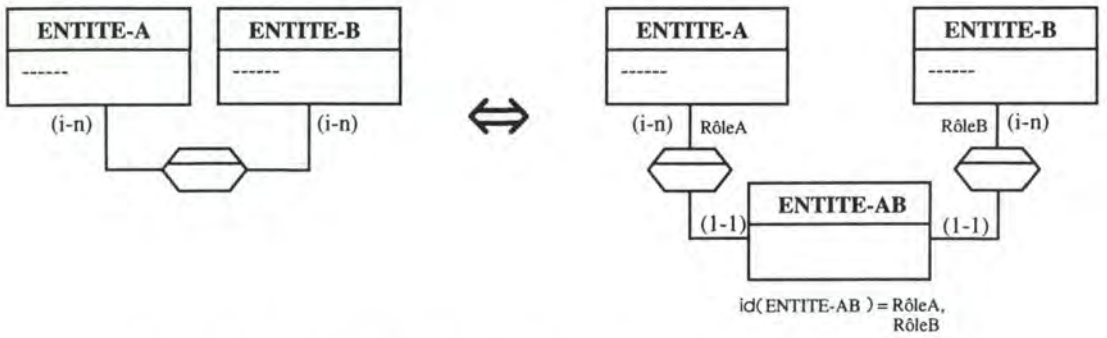

Figure 5.5 - Elimination d'un type d'association "many-to-many"

- Transformation d'un type d'association simple :

cette transformation permet d'éliminer les types d'associations simples, c'est à dire un type d'association binaire qui ne possède pas d'attributs. Cette transformation est fondamentale dans la mesure où elle permet d'obtenir une structuration des données sous forme de tables, c'est à dire en conformité avec la structuration relationnelle des données. Le type d'association est transformé en un attribut référentiel dans le type d'entité qui y joue le rôle (1-1) ou (0-1), on ajoute ensuite une contrainte référentielle au schéma.

Par exemple, on remplacera le type d'association simple reliant CLIENT à COMMANDE par l'attribut référentiel No-client dans le type d'entité COMMANDE plus la contrainte référentielle COMMANDE.No-client **in** CLIENT.No-client. On obtient ainsi les tables CLIENT et COMMANDE telles que décrites à la Fig. 5.6.

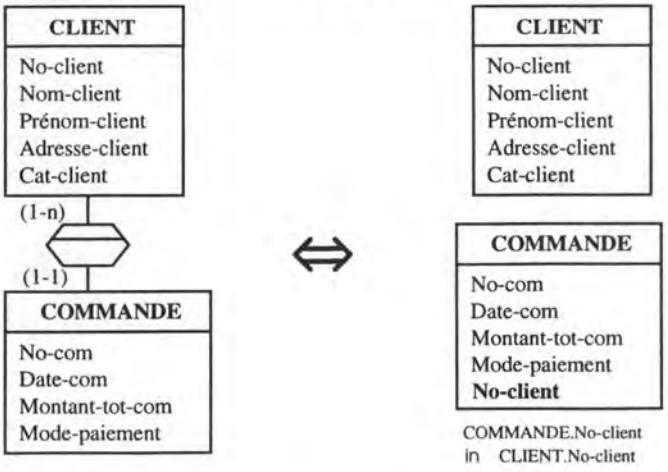

Figure 5.6 - Elimination d'un type d'association simple

B.2. Algorithme de transformation

Maintenant que les différentes transformations élémentaires ont été présentées, expliquons la stratégie de transformation suivie par l'algorithme de transformation, c'est à dire comment celui-ci utilise ces transformations élémentaires pour réaliser un schéma relationnel standard[14].

La première étape consiste à transformer les types d'association complexes; le schéma ne contient alors plus que des types d'associations binaires. La deuxième étape consiste à transformer les attributs non conformes au relationnel, c'est à dire les attributs répétitifs et les attributs composés. La troisième étape consiste à éliminer les types d'association "many-to-many" pour ne plus avoir que des types d'association simples. On peut alors passer à la dernière étape et éliminer tous les types d'association simples afin d'obtenir une structuration des données sous forme de tables. Une fois les tables définies, il ne reste plus qu'à définir l'ensemble minimal des clés d'accès aux tables.

Dans le modèle relationnel, tout identifiant est une clé d'accès, il faudra donc définir une clé d'accès pout tous les identifiants. Ces clés seront exprimées par un index[15], et pour assurer l'unicité des valeurs de l'identifiant ces index seront des index uniques[16], le SGBD-R n'acceptera alors pas que l'on insère dans la table plus d'une ligne possédant les mêmes valeurs pour ces colonnes.

Voici l'algorithme abstrait de transformation d'un schéma entité-association en un schéma relationnel :

// Transformation des **types d'association complexes** :
Pour tous les types d'association complexes :
    Appliquer la transformation de la Fig. 5.2 ;

// Transformation des **attributs répétitifs** et des **attributs décomposables** :
Tant qu'il existe des attributs répétitifs ou des attributs décomposables :
    Pour tous les attributs répétitifs :
        Appliquer la transformation de la Fig. 5.4 ;
    Pour tous les attributs décomposables :
        Appliquer la transformation de la Fig. 5.3 ;

// Transformation des **types d'association "many-to-many"** :
Pour tous les types d'association "many-to-many" :
    Appliquer la transformation de la Fig. 5.5 ;

---

[14] Cette stratégie est la même que celle proposée pour produire un schéma conforme au relationnel dans [Hai,92].

[15] Un index est fait pour accélérer l'accès direct basé sur une valeur pour les champs de l'index, il permet d'accéder aux lignes de la table dans l'ordre (dé-)croissant des valeurs de ces colonnes. Etant donné qu'ils prennent beaucoup de place, on ne créera que les index strictement nécessaires, c'est à dire ceux qui correspondent aux identifiants.

[16] On parlera indifféremment de clé d'accès ou d'index dans ce qui suit.

// Transformation des **types d'association simples** :

Pour tous les types d'association simples :

Appliquer la transformation de la Fig. 5.6 ;

// Définition de l'ensemble minimal des **clés d'accès** :

Pour tous les identifiants :

Définir un index unique ;

Illustrons maintenant cet algorithme à l'aide d'un exemple simple. Reprenons le schéma entité-association que nous avions pris comme exemple précédemment (Fig. 5.1). Ce cas est assez simple et ne permettra donc d'illustrer que les transformations les plus courantes. Voyons maintenant les transformations qui seront appliquées.

L'algorithme transformera tout d'abord le type d'association LIGNE-COM et en fera un type d'entité identifié par les rôles joués par COMMANDE et PRODUIT. Il n'y pas d'attributs répétitifs, ni décomposables; il n'y pas non plus de type d'association "many-to-many", on passe donc ces 2 étapes. L'algorithme n'aura donc plus qu'à transformer tous les types d'association simples, on obtiendra ainsi les tables CLIENT, COMMANDE et ses contraintes référentielles, PRODUIT et LIGNE-COM et ses contraintes référentielles (Fig. 5.7).

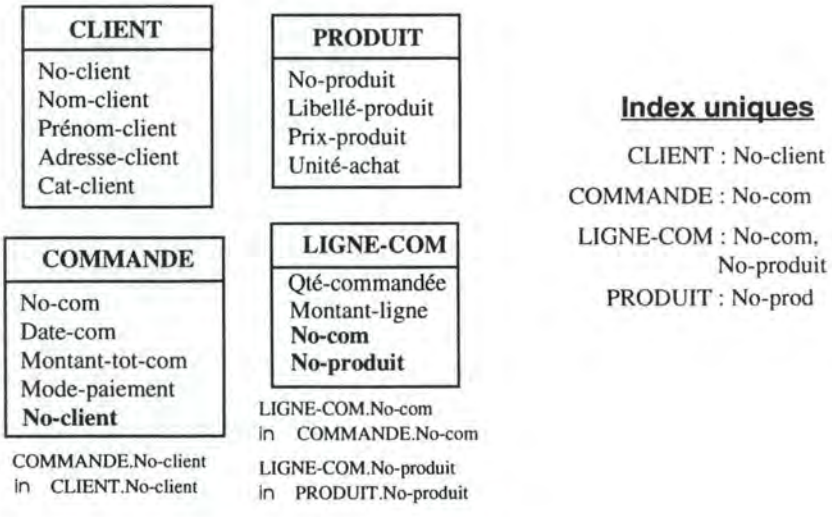

Figure 5.7 - Résultat d'une transformation

C. Stockage du schéma relationnel et des correspondances

Il ne suffit pas de transformer le schéma entité-association en un schéma relationnel, il faut stocker ce dernier dans le repository de l'outil d'aide à la conception et répercuter la transformation en établissant des correspondances entre les concepts des 2 schémas.

Voyons tout d'abord les éléments du schéma relationnel que nous allons stocker dans le repository; pour ce faire, nous nous baserons sur la syntaxe relationnelle standard de définition de données [Date,90]. Pour une table, nous stockerons son nom, la définition

de ses colonnes (leur nom et leur format), la définition de sa Primary Key (qui correspond à l'identifiant) et de ses éventuelles Foreign Key (clé de référence contenant un ou plusieurs attributs de référence). Quant à l'index, nous stockerons son nom, la table sur laquelle il est défini ainsi que les colonnes qu'il comprend; les index définis seront tous uniques (V. ci-dessus).

L'algorithme de transformation que nous venons de présenter ne tenait compte que des transformations. Il faudrait intégrer la transformation et le stockage afin de ne plus avoir qu'un seul algorithme qui réalise la transformation du schéma conceptuel et en stocke les résultats au fur et à mesure. La transformation et le stockage des concepts impliqués dans la transformation ne seront alors plus 2 étapes distinctes, elles se feront en même temps.

L'idée est que lorsqu'un type d'entité n'est plus relié par un type d'association, autrement dit lorsque ce premier est devenu une table, on stocke la table et ses composants (colonnes, Primary Key et éventuellement Foreign Key) dans le repository dans un premier temps, on répercute ensuite cette transformation en établissant les correspondances entre cette table, ses composants et les concepts du schéma entité-association. Il faut encore stocker la définition de l'index unique pour chaque table, mais ici, il n'y a pas de correspondance avec le schéma entité-association à stocker puisque les accès sont des caractéristiques propres au niveau logique et physique.

Il reste maintenant à modifier quelque peu l'algorithme de transformation précédent dans sa phase finale pour qu'il tienne compte du stockage. Lorsque l'algorithme arrivera à l'étape de transformation des types d'association simples, à chaque transformation appliquée, il regardera les 2 types d'entité impliqués dans la transformation : pour celui (ceux) qui est (sont) devenu(s) une (des) table(s), il en stockera la définition [nom de la table, ses colonnes et leur format, sa Primary Key et ses éventuelles Foreign Key], définira et stockera un index unique, et il stockera les correspondances avec le schéma entité-association.

Les modifications de l'algorithme précédent commence à partir de la transformation des types d'entité simples :

..... Transformation des types d'association complexes, des attributs répétitifs et décomposables
..... et des types d'association "many-to-many" réalisée

*// **Transformation** des types d'association simples ET **stockage** :*

Pour tous les types d'association simples :

Appliquer la transformation de la Fig. 5.6 ;

Pour tout type d'entité impliqué dans la transformation qui n'est plus relié à un autre type d'entité par un type d'association :

Stocker la table et ses composants ;

Définir et stocker son index unique (A) ;

Stocker les correspondances avec le schéma entité-association (B) ;

(A): l'index unique est défini sur base de l'identifiant et est ensuite stocké; il n'y a pas de correspondance avec le schéma entité-association puisque c'est un concept qui lui est étranger;

(B): pour établir les correspondances, il faudra regarder l'origine de la table (c'est à dire un type d'entité, un type d'association ou un attribut répétitif) et en fonction de cela, on connaîtra les correspondances à établir et à stocker. Voici une table de correspondance afin de montrer les différentes liens qui peuvent exister entre les concepts du schéma entité-association et les concepts du schéma relationnel (Table 6.1).

Concepts du schéma entité-association	Concepts du schéma relationnel
Type d'entité	Table
Identifiant d'un type d'entité	Primary key
Attribut simple	Colonne
Attribut répétitif	Table
Type d'association simple	Foreign Key
Type d'association "many-to-many"	Table
Type d'association complexe	Table

Table 6.1 - Correspondance entre les concepts du schéma entité-association et les concepts du schéma relationnel

Un type d'entité se retrouvera sous forme d'une table dans le schéma relationnel et son identifiant en deviendra la Primary Key. En ce qui concerne les attributs, il faut faire la distinction entre les attributs simples, répétitifs et décomposables. Un attribut simple deviendra une colonne d'une table; un attribut répétitif correspondra après sa transformation (Fig. 5.4), à une table; seuls les composants d'un attribut décomposable ,en tant qu'attributs simples, correspondront à des colonnes. Un type d'association simple sera éliminé et la référence sera faire grâce à un ou plusieurs attributs de référence, c'est à dire une Foreign Key. Un type d'association "many-to-many" correspondra après transformation (Fig. 5.5), à une table; un type d'association complexe correspondra aussi, après transformation (Fig. 5.2), à une table.

2.4.2. Génération-exécution du code DDL

Le schéma relationnel standard défini dans l'étape précédente contient toutes les informations nécessaires à la création d'une base de données, et la deuxième étape de la génération de la base de données consistera à générer le code DDL conforme au SGBD-R cible sur base de ce schéma et à exécuter ce code afin de créer la base de données relationnelles.

Etant donné que ne sont générés que des concepts relationnels standards, la génération du code DDL sera identique pour tous les SGBD-R cible. Le module de génération

exécutera une procédure d'extraction du schéma relationnel (commune à tous les modules), générera le code DDL et exécutera ce code.

A. Extraction du schéma relationnel

L'extraction du schéma relationnel sera réalisée grâce à une procédure d'extraction réalisée par l'outil d'aide à la conception, cela nous permettra d'avoir tous les éléments du schéma selon le format le mieux adapté à la génération du code DDL. Il faudra donc définir cette procédure dans l'outil et elle aura pour résultat un fichier que l'algorithme de génération de code DDL récupérera.

Le fichier comprendra la définition complète des tables et celle des index. La définition des tables comprendra leur nom, leurs colonnes et leur format[17], leurs Primary Key et Foreign Key. La définition des index comprendra leur nom, la table sur laquelle il est défini et les colonnes qu'il comprend. Cette prcédure d'extraction sera le même pour tous les modules de génération-exécution de code DDL.

B. Génération du code DDL

Le code DDL se compose d'un certain nombre d'instructions de création SQL qui permettront de créer la base de données dans son entièreté : les instructions de création d'une table (Create Table ...) et celles de création d'un index (Create [Unique] Index). Il faudra donc analyser le fichier d'extraction et générer une instruction de création d'une table pour chaque table ainsi qu'une instruction de création d'un index unique (puisque la transformation ne définit que des index uniques) pour chaque index. Voici un algorithme abstrait très général de génération du code DDL :

*// Génération des **instructions de création d'une table** :*

Pour toutes les tables du schéma relationnel :

Générer une instruction de création d'une table (A) ;

*// Génération des **instructions de création d'une table** :*

Pour tous les index du schéma relationnel :

Générer une instruction de création d'un index unique (B) ;

(A): toutes les informations utiles à la génération de l'instruction de création se trouvent dans le fichier d'extraction du schéma relationnel. Les Primary Key et Foreign Key seront définies lorsque le SGBD-R les supporte[18]. A noter que lorsque l'attribut d'une colonne est obligatoire, il faudra rajouter la caractéristique NOT NULL à la définition de la colonne.

(B): toutes les informations utiles à la génération de l'instruction de création se trouvent aussi dans le fichier d'extraction du schéma relationnel.

---

[17] Pour obtenir le format d'une colonne, il suffit de regarder le format de l'attribut qui lui correspond; il faudra aussi spécifier s'il est obligatoire ou non.

[18] La plupart des SGBD-R actuels supportent ces clés.

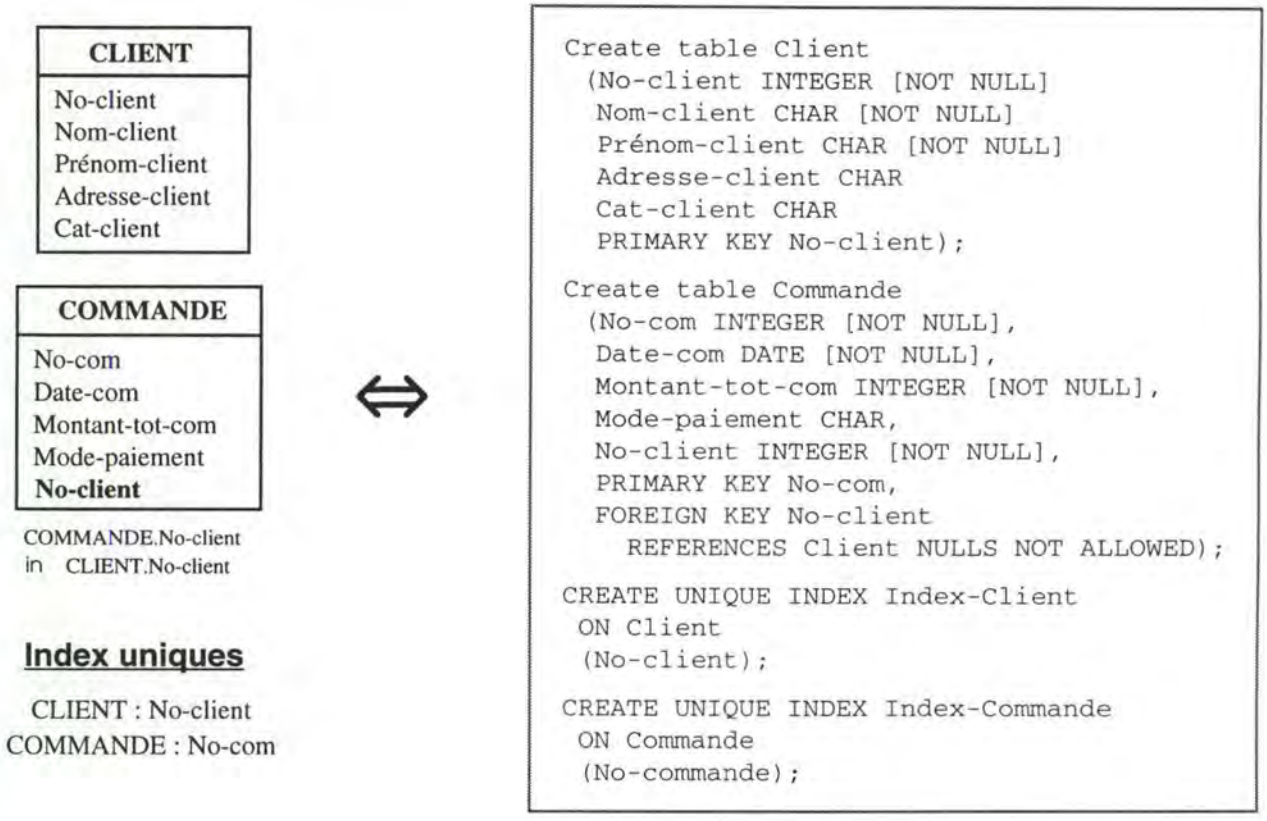

Figure 5.8 - Instructions de création d'une table et de son index unique

Afin d'illustrer la génération du code DDL, prenons deux des tables de la Fig. 5.7 (CLIENT et COMMANDE) ainsi que leur index unique et montrons les deux instructions de création de table même et leur index (Fig. 5.8).

C. Exécution du code DDL

Une fois le code DDL généré, le générateur se chargera de l'exécuter directement dans le SGBDR. Pour ce faire, il faudra déclarer l'adresse du SGBD-R au générateur.

Le générateur relationnel exécutera donc tout d'abord la procédure d'extraction du schéma entité-association, l'algorithme de transformation et de stockage ensuite, l'algorithme de génération et d'exécution du code DDL pour terminer. Ainsi le schéma relationnel standard et les correspondances avec le schéma entité-association seront stockés et la structure de la base de données relationnelle sera créée.

6

L'application cliente

L'application cliente est exécutée sur un poste de travail, et elle accède à la base de données par l'intermédiaire du SGBD situé sur le serveur. Elle est constituée de l'interface et des fonctions de l'application.

Nous allons tout d'abord établir l'architecture des applications 4GL, et nous traiterons ensuite séparément la modélisation et la génération des différents composants identifiés dans l'architecture.

1. Architecture des applications 4GL

Avant d'aborder l'architecture physique, mettons en évidence les composants logiques de l'application.

1.1. Différents niveaux d'abstraction de l'interaction

On peut considérer l'interaction de l'application avec l'utilisateur à différents niveaux d'abstraction [Cou,91]. On peut tout d'abords envisager une interaction à deux participants: l'utilisateur et l'application (Fig. 6.1).

Figure 6.1 - Interaction à deux participants

On peut affiner cette perspective et considérer l'application comme un groupe de trois composants : le composant fonctionnel, le composant de gestion du dialogue et le composant présentation. On peut alors considérer qu'on passe d'une interaction de deux participants à une interaction de quatre participants (Fig. 6.2).

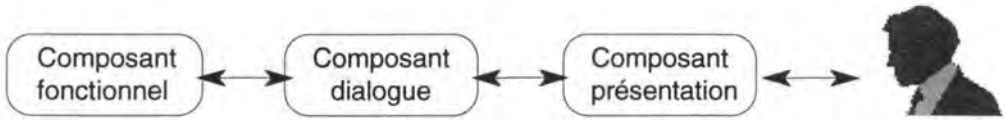

Figure 6.2 - Interaction à quatre participants

On peut encore préciser cette perspective en considérant les interfaces graphiques telles qu'on les a définies dans le premier chapitre. De façon simple on considère qu'elles sont constituées d'un ensemble d'objets intéractif (OI) que l'utilisateur va manipuler. On peut alors voir le composant présentation comme une collection d'OI qui agissent comme des partenaires pour réaliser l'interface utilisateur (Fig. 6.3).

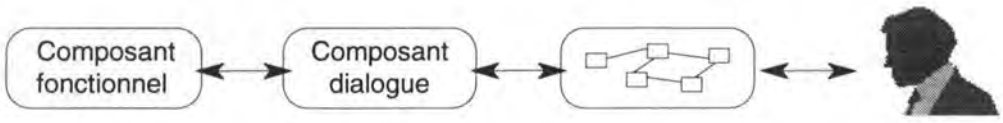

Figure 6.3 Interaction à participants multiples

Le *composant présentation* assure la représentation externe des concepts sémantiques de l'application et prend en compte les actions de l'utilisateur.

Le *composant dialogue* contrôle la présentation et le séquencement des interactions avec l'utilisateur; par exemple, il détermine quand un OI est accessible à l'utilisateur. Il gère la dynamique de l'interface et agit comme un médiateur, un arbitre entre le composant fonctionnel et la présentation. Selon que le contrôle sera laissé à l'utilisateur ou géré par le système son rôle sera différent (V. plus loin).

Le *composant fonctionnel*, quant à lui, encapsule les fonctions de l'application.

Après avoir fait apparaître de manière logique les différents intervenants dans l'interaction, voyons maintenant de manière concrète et précise comment ces composants logiques sont organisés dans une architecture informatique, compte tenu des outils 4GL.

1.2. Architecture de l'application

Nous allons voir que la modularisation logique établie ci-dessus est conservée de manière concrète dans les applications 4GL. La gestion de la présentation, du dialogue et du corps fonctionnel sont indépendantes. Nous verrons sur quelle architecture elles reposent.

1.2.1. Séparation effective des trois composants

S'il y a bien une séparation effective entre les trois composants, ils ne doivent toutefois pas tous les trois être gérés par l'application. En effet, le gestionnaire d'interface de l'environnement 4GL va prendre en charge la gestion de la présentation.

A. Présentation gérée par l'outil 4GL

L'outil 4GL est un environnement complet, offrant différents services aux développeurs. En l'occurrence, son gestionnaire d'interface libère l'application des manipulations physiques d'entrées et sorties. Il collecte entre autres, les touches frappées par l'utilisateur, il forme des unités abstraites de plus haut niveau (entier, chaîne de caractères) et les stocke dans un des attributs des OI (editbox.text par exemple) de telle manière que l'application, ou plus exactement le dialogue, puisse exploiter cette donnée sans avoir dû en gérer la saisie.

Le dialogue accède aux services qu'offre la présentation pour la manipulation physique des OI au travers d'un langage orienté objet. Il utilise les primitives et manipule les attributs des OI sans accéder à l'écran. Seul le gestionnaire d'interface peut accéder directement à l'écran et recevoir des données du clavier ou de la souris.

Outre les services qu'il offre pour la manipulation de l'interface, le gestionnaire previent également le composant dialogue des différentes actions de l'utilisateur. Il assure la gestion des événements utilisateur: les actions de celui-ci sur les OI sont converties en événements auxquels le dialogue peut répondre. Lorsqu'un événement pris en compte par le dialogue est généré par la présentation, le dialogue prend alors le contrôle et exécute la réponse qu'il avait prévue pour cet événement.

B. Dialogue et composant fonctionnel

La gestion de la présentation étant réalisée par l'outil 4GL, l'application à réaliser concrètement se compose désormais de la gestion du dialogue et du composant fonctionnel. Si dans l'identification logique des participants de l'interaction, ces deux composants étaient séparés, il devrait en être de même dans l'architecture physique de l'application. Le principe reconnu de séparation entre l'interface et le domaine fonctionnel pourrait ainsi être respecté.

B.1. Principe de séparation entre le composant fonctionnel et l'interface

Le composant fonctionnel de l'application devrait idéalement ignorer la manière dont ses données et ses fonctions sont présentées à l'utilisateur via l'interface. Il ne devrait pas savoir si les données qui lui sont transmises viennent d'un menu ou d'une ligne de commande. Cette indépendance entre l'interface et application facilite considérablement le développement et la maintenance des applications. Des personnes différentes peuvent développer séparément ces parties; des modifications de l'interface pourront se faire sans interférer avec le reste de l'application et inversément. Cette modularisation favorise également la portabilité du système.

Cette indépendance est rendue possible dans l'environnement 4GL par la possibilité de coder la partie fonctionnelle de l'application sous forme d'une librairie de fonctions totalement indépendante de la partie interface.

B.2. Dialogue

Le dialogue assure les traductions pour l'application des requêtes et données d'entrées en fonction de l'état courant du système, ainsi que les traductions pour l'utilisateur des retours de l'application [Mei,91]. Nous approfondirons et définirons précisément cette notion de dialogue dans le chapitre qui lui sera consacré. Néanmoins, nous pouvons déjà considérer que la gestion du dialogue aura essentiellement pour but d'assurer un enchaînement correct des fonctions de l'application, d'afficher les résultats de ces fonctions et d'apporter une guidance à l'utilisateur dans l'accomplissement de sa tâche.

B.3. Composant fonctionnel

Le composant fonctionnel, compte tenu du rôle du dialogue, est vu comme un presteur de services, qui attend que ses fonctions soient appelées. On appelera désormais ce composant la machine fonctionnelle.

Le dialogue se chargeant de vérifier que les conditions de déclenchement (les préconditions) sont vérifiées, les fonctions s'exécutent à sa demande en utilisant en entrée les paramètres fournis par le dialogue et en lui renvoyant les paramètres résultats.

B.4. Niveau d'abstraction des données échangées entre le composant fonctionnel et le composant dialogue

Si la communication entre ces deux composants se fera par un appel classique de fonctions, la machine fonctionnelle devra néanmoins recevoir du dialogue les données sur lesquelles les fonctions s'effectueront. Il faut donc définir le niveau d'abstraction des données échangées. Il peut aller du niveau lexical (donnée exprimée en pixels par exemple) au niveau sémantique (types de données classiques).

Si l'on veut maintenir l'indépendance du composant fonctionnelpar rapport à l'interface, les données échangées entre le module fonctionnel et le module dialogue devront être exprimées en terme de concepts sémantiques du domaine fonctionnel. Si une fonction utilise une donnée de type entier représentant un attribut, l'échange avec le module dialogue se fera en terme d'entier sans aucune considération sur la manière dont cette donnée est représentée par la présentation.

1.2.2. Architecture

Avant de voir comment les composants logiques s'organisent concrètement, Présentons deux approches extrêmes qui proposent de les implémenter de manières différentes.

A. Approche monolithique

Cette approche transforme les différents niveaux logiques en couches physiques dans l'architecture; une telle architecture fut identifiée par Seeheim. De cette perspective est née l'approche linguistique qui met en parallèle les notions de dimensions sémantique, syntaxique et lexicale avec les trois composants de l'application.

Ce découpage de l'application en gros modules semble peu approprié avec la multiplicité des actions possibles dans un dialogue conduit par l'utilisateur [Mei,91]. Cette faiblesse a provoqué l'émergence d'un autre modèle architectural plus adapté au dialogue événementiel: le modèle multi-agent.

B. Approche multi-agents

Une des difficultés majeures dans la construction des interfaces graphiques réside dans le contrôle du dialogue parce que l'interaction avec l'utilisateur s'effectue par l'intermédiaire d'une multitude de participants que sont les OI. Ce type d'interface événementielle offre des possibilités de choix d'action considérables, l'utilisateur peut intervenir n'importe où dans la fenêtre. En fait, le problème du dialogue événementiel est qu'il conduit rapidement à une combinatoire difficilement maîtrisable, sauf à le répartir sur des objets [Mei,91].

Par opposition à l'approche monolithique qui stratifiait l'application en un nombre restreint de blocs monolithiques, le paradigme de l'orienté objet permet de construire une architecture fondamentalement répartie. Les trois concepts de l'application sont séparés et pris en charge à chaque niveau d'abstraction. Le dialogue, notamment, peut reposer ainsi sur la multitude des participants. Ce modèle d'architecture répartie est appelé *modèle objet* ou *modèle multi-agents*.

Un agent réagit à des phénomènes externes (événements) et peut produire à son tour des événements. Il est considéré comme une unité de traitement complète regroupant une partie des trois composants de l'application. Les agents respectent le principe de responsabilité entre le composant fonctionnel et l'interface, et vont plus loin en appliquant cette séparation aux différents niveaux d'abstraction. Ils communiquent entre eux et avec l'utilisateur, ceux communiquant directement avec l'utilisateur sont appelés les OI. Outre l'avantage de mieux modéliser le dialogue, ce modèle peut facilement être implémenté sous la forme d'un langage orienté objet. Le parallélisme entre un agent et un objet est facile à réaliser.

Plusieurs modèles et outils trouvent leurs origines dans ce modèle multi-agent. Les modèles MVC et PAC en sont des exemples [Cou, 91].

C. Architecture résultant des deux approches

Compte tenu des possibilités des outils 4GL, leurs applications peuvent être construites suivant une architecture reposant à la fois sur les deux types de modèle. Une architecture répartie concernant le dialogue, et une architecture monolithique concernant la partie fonctionnelle de l'application.

Le dialogue sera réalisé par l'intermédiaire d'un ensemble de scripts indépendants les uns des autres. Le dialogue est en quelque sorte partitionné et réparti sur les différents OI réalisant l'interface. La partie fonctionnelle de l'application sera constituée d'un ensemble de fonctions stockées dans une librairie (Fig. 6.4). L'indépendance entre l'interface et la partie fonctionnelle est ainsi préservée.

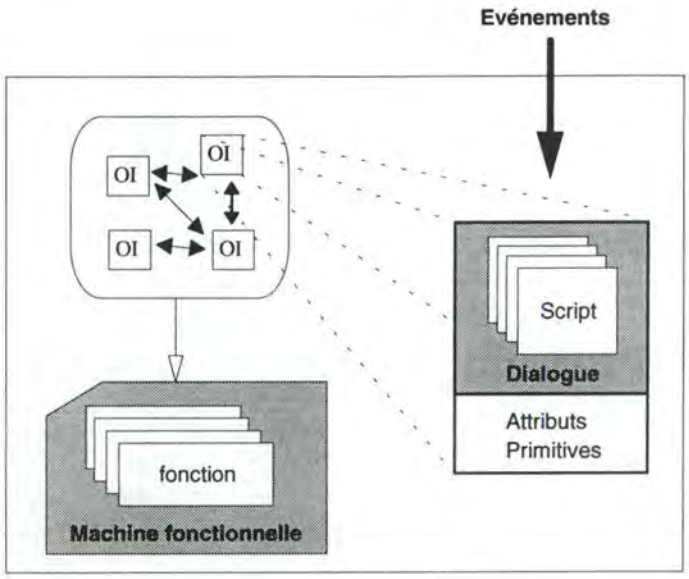

Figure 6.4 - Architecture de l'application

Le code de l'application à générer consistera donc en un ensemble de scripts et une série de fonctions dont nous envisagerons uniquement la signature. Il s'agira en outre de générer aussi la présentation, c'est à dire les différents OI réalisant l'interface. Chacun de ces composants va être détaillé dans ce qui suit.

2. Les fonctions de l'application

Pour pouvoir générer les fonctions, il faut tout d'abord décrire les traitements de l'application à l'aide de modèles de spécification. Il faudra structurer ces différents traitements et décrire ensuite les fonctions précisément.

2.1. Description des traitements de l'application - modèles de structuration et de la statique des traitements

Une application se compose d'un certain nombre de traitements qui ont pour objectif d'en manipuler les données. Un traitement[1] est en fait, une fonctionnalité à un certain niveau de détail de l'application; chaque niveau traduisant une certaine préoccupation de l'organisation. En l'occurrence, les traitements qui nous intéressent sont les traitements de type phase et de type fonction [Bod-Pig,89]; et plus particulièrement les fonctions puisqu'il s'agit dans cette partie de générer la signature des fonctions de l'application. Nous verrons que pour identifier les différentes fonctions de l'application, il faudra d'abord identifier les différentes phases de l'application.

Comme pour la structuration des données, des modèles de spécification des traitements ont été construits afin de pouvoir les définir convenablement et parmi ceux-ci, le modèle de structuration et de la statique des traitements de la méthode IDA[2]. Ils concourront à décrire les phases et les fonctions de l'application.

Nous allons tout d'abord définir comment se structurent les phases et fonctions de l'application. Nous verrons ensuite comment les fonctions seront spécifiées.

2.1.1. Structuration des différents types de traitement

Une application se compose tout d'abord d'un certain nombre de phases[3]. Celles-ci correspondent en fait à des fonctionnalités de l'application, à des services de gestion que l'application rend comme l'enregistrement d'une commande ou d'un vol de ligne par exemple.

Une phase, quant à elle, se compose d'un certain nombre de fonctions qui permettront de la réaliser. Une fonction est un traitement élémentaire indécomposable qui correspond à un service élémentaire de gestion que l'application rend et dont le but est de participer

---

[1] Seuls les traitements interactifs automatisés par l'application nous intéressent.

[2] Il faudra quelque peu aménager le modèle de la statique.

[3] On peut considérer l'application elle-même comme un traitement dont le but est de supporter un aspect de gestion d'une organisation. D'ailleurs, lorsque nous exposerons le sous-ensemble du langage DSL étendu, nous verrons que si le développeur veut définir son application, il pourra le faire en définissant un traitement de niveau application et pourra ainsi y rattacher les différentes phases qui la composent.

à l'objectif "plus global" d'une phase. L'ensemble des fonctions permettra en quelque sorte, de réaliser l'objectif d'une phase[4].

Il y a donc 2 niveaux de traitements: le niveau phase qui comprend les phases composant l'application et le niveau fonction qui comprend les fonctions composant une phase. On peut représenter ces 2 niveaux à l'aide d'une arborescence dont la racine est l'application, les noeuds les phases de l'application et les feuilles les fonctions de l'application (Fig. 6.5). Dans cette structuration arborescente de l'application, la relation de structuration est la relation "se compose de" (une application se compose d'un certain nombre de phases qui se composent elles-mêmes d'un certain nombre de fonctions).

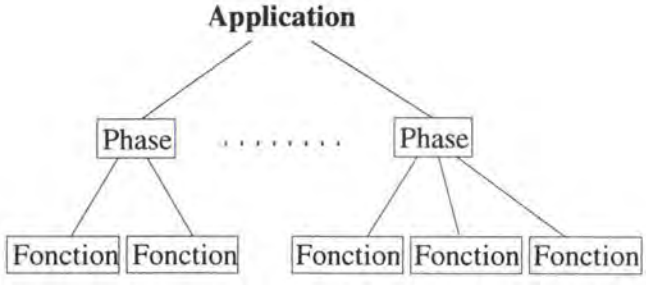

Figure 6.5 - Structuration arborescente d'une application

En réalité, la description des traitements va se faire en 2 étapes. Il faudra tout d'abord décomposer l'application afin d'identifier et de définir les différents traitements et ceci, au niveau des phases et des fonctions. On pourra ensuite décrire avec précision les différentes fonctions qui auront été identifiées.

La première étape consiste donc à décomposer l'application en ses différents traitements, ce qui permettra d'élaborer la structuration arborescente de l'application. Il faudra identifier et définir les différentes phases qui composent l'application, spécifier et définir ensuite les fonctions qui les composent[5]. Pour réaliser cette décomposition, il est intéressant d'utiliser les différents critères d'identification exposés dans le modèle de structuration des traitements [Bod-Pig,89].

La deuxième étape consiste à définir précisément et statiquement chacune des fonctions selon le modèle de la statique des traitements. La spécification de la statique des fonctions se compose de 2 parties: définition des entrées et sorties selon le modèle de la boîte noire et spécification des règles de transformation des informations en entrée en information de sortie.

La description d'une phase est assez simple dans la mesure où il faut juste en définir l'objectif et spécifier les fonctions qui la compose.

---

[4] L'exécution d'une fonction participe en fait à l'exécution d'une phase; autrement dit, la conjonction de l'exécution des fonctions d'une même phase permet d'exécuter celle-ci.

[5] Il est détaillé dans [Bod-Pig,89] une démarche constructive et déductive qui met en évidence et décrit progressivement les fonctions de la phase d'une application.

2.1.2. Description des fonctions de l'application

Une fois les fonctions composant une phase identifées, on pourra en faire la déscription détaillée. Celle-ci se composera de 2 parties : le modèle de la boîte noire et les règles de traitement.

A. Le modèle de la boîte noire

Cette première partie permet de spécifier les messages reçus et générés, c'est à dire les informations en entrée et en sortie, ainsi que la manipulation des données de l'application. Les fonctions ont autant besoin d'informations venant de l'extérieur de l'application, à savoir l'utilisateur, que venant de l'application elle-même. C'est pourquoi nous ferons tout d'abord la différence entre les messages externes et les messages internes d'une fonction.

A.1. Messages externes

Les messages externes sont en provenance ou à destination de l'interface, c'est à dire des informations qui proviennent de l'utilisateur ou qui lui sont destinées. Ce sont des messages élémentaires; autrement dit, chacun d'entre eux contient une information atomique. Ils correspondront donc à des attributs non-décomposables d'une entité ou d'une association (le numéro d'un client, la quantité commandée d'un produit, ...).

A.2. Messages internes

Les messages internes, quant à eux, sont en provenance ou à destination d'une autre fonction de la phase, autrement dit de l'application elle-même. Ces messages, au contraire des précédents, peuvent être décomposables: chacun d'entre eux peut contenir un ensemble d'informations, atomiques ou non. Cet ensemble d'informations peut contenir des entités, des associations ou des attributs (un message contenant un client valide et/ou plusieurs lignes de commande valides, ...).

A.3. Messages d'erreur

Les fonctions fournissent soit un résultat valide si le traitement s'est correctement déroulé, soit un message d'erreur indiquant que le traitement a échoué; il faut donc tenir compte des messages d'erreur. Ce dernier type de message est spécifique dans la mesure où, contrairement aux messages externes et internes, il n'est pas en rapport direct avec les données. En effet, ce n'est qu'un message contenant un texte expliquant succinctement l'erreur commise; c'est pourquoi nous le considérons comme un message à part entière[6].

---

[6] Etant donné que ce message est à destination de l'utilisateur, on peut le considérer comme un message externe particulier.

A.4. Manipulation des données

La manipulation des données de l'application par une fonction est de 2 types, soit c'est une consultation, soit c'est une mise à jour. Il y a, en tout, 4 façons de manipuler les données: consulter, ajouter, modifier ou supprimer des données. Il faudra spécifier pour chaque fonction toutes les manipulations réalisées sur les données.

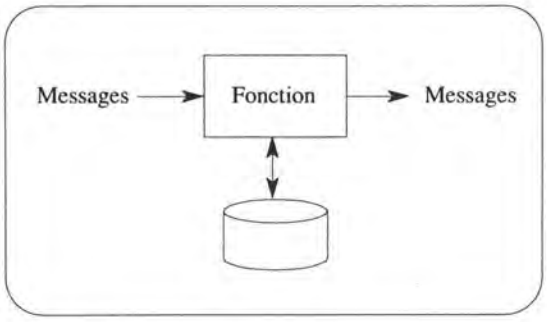

Figure 6.6 - Définition schématique d'une fonction

Pour illustrer ceci, on peut prendre l'exemple d'une fonction et définir en langage naturel les différents messages et les données qu'elle manipule :

Traitement Fonction Valider-Nom-client;

NIVEAU	*FONCTION*
DEFINITION	*a pour objectif d'identifier un Client, existant dans la mémoire (Type d'Entité Client), à partir de son Nom et Prénom;*
RECOIT	*les messages externes Nom-client, Prénom-client;*
GENERE	*les messages externes Num-client, Adresse-client,* *le message interne Client-valide* *SI le Client existe;*
GENERE	*message d'erreur* *SI le Client n'existe pas;*
CONSULTE	*l'entité Client;*

Maintenant que les messages en entrée et en sortie d'une fonction sont identifiés et que l'on connaît les concepts du schéma entité-association qu'elle manipule, on peut en définir les règles de traitement.

B. Les règles de traitement

La définition des règles de traitement permettra de compléter la définition des fonctions en décrivant comment on obtient les messages de sortie à partir des messages d'entrée.

Différentes manières existent de spécifier les règles de traitement. Nous ne proposerons pas une technique particulière dont le choix sera laissé au concepteur. La procédure d'une fonction pourra être définie par exemple de la manière suivante. poursuivons l'exemple précédent Validation-Nom-client.

Rechercher client (entité Client);
Si le client existe

```
Alors   Num-client=Client.Num-client;
        Adresse-client=Client.Adresse-client;
Sinon   erreur="client inexistant";
```

Maintenant que les fonctions peuvent être décrites de façon précise, nous pouvons passer à la génération de leur signature.

2.2. Génération de la signature des fonctions

Nous venons de voir que la définition des règles de traitements n'était pas spécifiée de façon formelle. Il ne sera donc pas possible d'en générer les instructions de traitement dans le corps des fonctions exécutables. De plus, ces règles tiennent compte du schéma entité-association alors que dans l'application, il ne sera plus question que de tables relationnelles. Donc, même si ces règles étaient en langage formel, il faudrait d'abord les transformer pour qu'elles tiennent compte des tables obtenues après transformation, ce qui n'est pas simple.

Nous définirons donc un générateur de la signature des fonctions, c'est à dire un générateur de fonctions exécutables mais dont le corps est vide; corps que le programmeur devra compléter dans l'outil 4GL. Afin de l'aider au mieux dans sa tâche, il sera tout à fait intéressant de fournir les règles de traitement de la fonction en commentaire ainsi que les transformations en tables relationnelles des entités et associations manipulées[7].

Le générateur aura pour résultat un certain nombre de fichiers texte qui comprendront la définition des fonctions exécutables (c'est à dire leur nom et leurs paramètres), ainsi que les règles de traitement et les transformations en tables relationnelles des concepts du schéma entité-association manipulés en commentaire dans le corps de ces fonctions.

La génération se déroulera en fait en 2 étapes principales: une première pour définir et stocker dans le repository de l'outil d'aide à la conception les paramètres correspondant aux messages d'une fonction et leur type, une deuxième pour générer le fichier texte de définition de la fonction exécutable vide qu'il faudra récupérer dans l'outil 4GL :

- Extraction de la définition des fonctions du repository de l'outil d'aide à la conception, identification des paramètres de la fonction exécutable et de leur type, stockage de ceux-ci et de leur correspondance avec le message auquel ils sont associés dans le repository;
- Génération du fichier texte de définition de chaque fonction exécutable vide et stockage du format d'appel de la fonction dans le repository.

Jusqu'à maintenant, nous avons parlé de fonctions alors qu'en réalité, il s'agira plutôt de procédures, c'est à dire que les informations traitées et celles du résultat seront en

[7] Pour ce faire, il faudra consulter les correspondances entite-association/relationnel stockées lors de la génération de la base de données.

paramètre[8]. Les paramètres seront donc tous les messages en entrée et en sortie de la fonction. Le point sensible de la génération de la signature des fonctions sera l'identification du type des différents paramètres.

2.2.1. Identification des paramètres

Cette première étape consiste à extraire la description des fonctions du repository de l'outil d'aide à la conception afin de pouvoir définir les paramètres de la fonction exécutable et leur type.

A. Extraction de la définition des fonctions

Il faudra extraire du repository le nom de la fonction et tous ses messages d'entrée et de sortie (c'est à dire leur nom et leur structure).

L'extraction sera réalisée grâce à une procédure d'extraction de spécifications réalisée par l'outil d'aide à la conception. Il faudra définir cette procédure qui aura pour résultat un fichier dont on aura spécifié le format et que l'on récupérera pour définir les paramètres.

B. Définition des paramètres et Stockage

Tous les messages qui auront été extraits deviendront des paramètres dont il faudra définir le type. En ce qui concerne les messages externes, cela ne pose pas de problème puisque ceux-ci sont élémentaires. Pour les paramètres correspondant aux messages internes, c'est moins évident puisqu'ils sont décomposables et qu'il faut donc en définir la structure. Enfin, pour les paramètres correspondant aux messages d'erreur, cela ne pose pas de problème puisqu'il sont de type texte.

Rappelons les messages que peut recevoir ou générer une fonction :

Une fonction peut recevoir : un ou plusieurs messages internes;
un ou plusieurs messages externes.
Une fonction peut générer : un message d'erreur;
un message interne;
un ou plusieurs messages externes.

Pour chaque fonction, il s'agit de passer en revue tous les messages et de définir le nom du paramètre qui lui sera associé ainsi que son type. Le nom du paramètre sera constuit à partir du nom du message; quant à la définition du type, elle sera propre à chaque type de message.

[8] On continura à parler de fonction par la suite pour rester cohérent avec tout le reste.

Pour définir le type des paramètres correspondant à un message externe, il suffit de reprendre le type de l'attribut du schéma entité-association auquel il correspond[9]. Ces paramètres seront donc de type simple (booléen, chaîne de caractères, date, entier, heure, réel).

Pour définir le type des paramètres correpondant à un message interne, il faut tout d'abord en déterminer la structure, c'est à dire les différents éléments qui le composent. Ces paramètres seront donc des variables de type structure de données[10] dont les champs correspondront aux différents éléments que le message contient. Pour éviter d'avoir un paramètre trop complexe, nous définirons un type structure à un seul niveau de décomposition. Il faudra donc "aplatir" la structure, autrement dit ramener tous les éléments atomiques qui composent le message sur un niveau. Afin de savoir si un message interne a été généré par une fonction, on ajoutera un champ spécial de type booléen indiquant si, oui ou non, ce message interne a été généré; cela indiquera si la fonction a réussi ou échoué.

Les paramètres correspondant à un message d'erreur seront de type chaîne de caractère puisqu'ils ne contiennent qu'un texte précisant l'erreur.

Une fois les paramètres et leur type définis, il faudra les stocker dans le repository de l'outil d'aide à la conception. Il faudra aussi stocker les correspondances entre ceux-ci et les messages auxquels ils correspondent[11]. Il est maintenant possible de générer une fonction exécutable vide.

2.2.2. Génération du fichier de définition d'une fonction exécutable vide

Dans le fichier de définition d'une fonction exécutable vide, se trouvent la définition de son en-tête (le nom de la fonction et ses paramètres), les informations sur les règles de traitement et les correspondances entre les concepts entité-association et relationnels en commentaire dans le corps de la fonction.

A. Définition de l'en-tête d'une fonction

La définition de l'en-tête des fonctions permet d'en spécifier le format d'appel en donnant un nom à la fonction et en déclarant ses paramètres. En ce qui concerne le passage des paramètres, ceux qui correspondent aux messages d'entrée de la fonction seront passés par valeur alors que les paramètres correspondant aux messages de sortie seront passés par adresse.

---

[9] Rappelons qu'un message externe est élémentaire, c'est à dire qu'il ne contient qu'un attribut non-décomposable.

[10] Par structure de données, nous entendons une collection d'informations de types divers. C'est l'équivalent des Record en Pascal.

[11] Nous verrons que le type d'un message nous permettra aussi de définir les variables correspondant aux messages d'une phase de l'application.

Maintenant que le passage des paramètres a été défini pour la génération, nous pouvons générer la définition de l'en-tête des fonctions. La définition se composera du nom de la fonction suivi de la définition de ses paramètres (leur nom et leur type). Etant donné que la structure des paramètres a déjà été définie, il ne reste plus qu'à assembler les différents éléments de l'en-tête pour en obtenir une définition identique à ceci :

nom-de-la-fonction (paramètre1 type-paramètre1, paramètre2 type-paramètre2, ...);

C'est évidemment un format qui ne tient compte d'aucune syntaxe. Nous verrons pour Powerbuilder ce que la génération devra réellement donner comme résultat.

B. Ajouter les informations utiles au développeur en commentaire

La définition de l'en-tête des fonctions étant définie, il ne reste plus qu'à ajouter au fichier, en commentaire dans le corps de la fonction, le texte définissant les règles de traitement et les correspondances entre les concepts entité-association manipulés et les concepts relationnels.

Pour ce faire, il faudra définir une procédure d'extraction de spécifications qui se chargera d'extraire le texte de définition des règles de traitement ainsi que la correspondance entre les concepts entité-association manipulés et les tables et colonnes de la base de données, puisque c'est sur celles-ci que les fonctions travaillent.

3. L'interface utilisateur de l'application

Avant d'aborder la génération proprement dite, nous allons tout d'abord préciser les choix que nous avons effectués en ce qui concerne le type d'interface qui sera générée. Ces choix sont évidemment nécessaires si l'on adopte l'optique d'une génération automatique et vont constituer en quelque sorte des restrictions par rapport aux différentes possibilités d'interface envisageables.

Ces restrictions vont permettre de cerner et de délimiter avec précision le contexte de la génération et le type d'application qui sera généré. Elles seront d'une part guidées par les caractéristiques des outils 4GLs envisagés et résulteront, d'autre part, d'hypothèses posées.

Une fois ces restrictions établies, nous définirons le modèle (GEF) sur lequel se basera la génération de la présentation et du dialogue de l'interface d'une phase. Nous expliquerons ensuite les principes et la démarche suivie pour effectuer ces générations.

Dans ce chapitre, nous expliquerons la génération d'une interface abstraite, c'est à dire d'une interface qui ne tient pas compte des particularités d'un quelconque outil 4GL. Cela pemettra d'expliquer les principes de la génération sans s'encombrer de la syntaxe et du fonctionnement d'un outil particulier.

3.1. Restrictions du contexte de génération

Avant d'envisager les restrictions délimitant concrètement le type d'interface générée, rappelons qu'une première restriction concernant le type des applications a déjà été faite au début du travail. En effet, l'environnement de génération automatique d'application que nous envisageons dans ce mémoire ne concerne que les applications de gestion hautement intéractives définies précédemment.

3.1.1. Styles d'interaction

Il y a de nombreuses manières de mettre en oeuvre le dialogue entre l'utilisateur et l'application. Le dialogue par question-réponse, les menus, les formulaires, les langages de commande, les touches de fonctions, le langage naturel, la manipulation directe et le dialogue objet-action sont des possibilités de dialogue [Mei,91].

Les caractéristiques des outils 4GL et des applications visées, nous permettent d'établir les restrictions suivantes.

Il est difficilement possible d'avoir un style d'interaction pour toute une application. En général, il est fait usage de plusieurs styles d'interaction. Et ce qui convient le mieux pour les applications de gestion et qui est permis par les outils 4GL, c'est la sélection de menu et le remplissage de formulaire.

Le remplissage de formulaire est tout à fait approprié lorsque l'application se compose d'un certain nombre de fenêtres permettant de saisir ou d'afficher des informations et

déclencher des fonctions, ce qui est le cas des applications de gestion[12]. La sélection de menu permettra d'accéder aux différentes fenêtres.

3.1.2. Contrôle de l'interaction

Le contrôle du dialogue entre l'utilisateur et l'application peut être de trois types. Il peut premièrement être géré par l'utilisateur et on parle dans ce cas de dialogue "user-driven". Il peut ensuite être géré par l'application et on qualifiera le dialogue de "system-driven". On peut finalement envisager un dialogue contrôlé alternativement par l'application et l'utilisateur que l'on appellera contôle mixte.

A. Dialogue "user-driven"

Avec ce type de contrôle, également appelé "event-driven", l'utilisateur est libre à tout moment de choisir parmis un nombre d'options valides [Cou,91]. Ses actions sur les OI de l'interface sont transformées en événements et en fonction de ceux-ci, les fonctions de l'applications sont invoqués.

Si l'on fait référence aux composants logiques d'une application, on dira que le composant présentation assure la traduction des actions de l'utilisateur en terme d'événements et que le composant fonctionnel agit comme un "serveur sémantique" dont les services (les fonctions) sont utilisés par le composant dialogue.

B. Dialogue "system-driven"

Lorsque le style d'interaction est "system-driven", les options d'interaction possibles sont réduites à une seule possibilité et l'utilisateur est guidé sans avoir le choix d'ordonner celles-ci de la manière qui lui convient le mieux. L'interaction est guidée par des considérations fonctionnelles et le contrôle est géré par le composant fonctionnel.

En général, l'utilisateur devrait avoir l'initiative du dialogue, mais dans certains cas, l'interaction doit être contrôlée par le système. Dans le cas d'un logiciel d'apprentissage par exemple, les connaissances de l'utilisateur du domaine d'application étant limitées, il se sentira plus à l'aise si l'ordonnancement des tâches à accomplir lui est soumis.

C. Contrôle mixte

On peut aussi considérer l'interaction de l'utilisateur avec l'ordinateur comme une collaboration dans laquelle les deux partenaires agissent selon leurs compétences. L'utilisateur agissant librement et le système prenant le contrôle arbitrairement à certains moments. La difficulté dans ce mode mixte d'interaction réside dans l'identification des points de transition où le contrôle passe de l'utilisateur à la machine et inversément.

---

[12] Nous appellerons ces fenêtres des fenêtres de type formulaire.

D. Restriction sur le contrôle de l'interaction

Il faudra laisser à l'utilisateur le plus possible de liberté dans la saisie des informations pour que son travail dans l'application soit le plus confortable possible. On lui donne la possibilité de rentrer ses informations là où il le désire et dans l'ordre qui lui convient le mieux. Autrement dit, toutes les informations qu'il peut saisir seront à tout moment accessible et modifiable.

Il n'y a pas que les informations à saisir, il y a aussi les fonctions à déclencher. Les différents types de déclenchement sont le déclenchement explicite, affiché ou non affiché[13], et le déclenchement implicite. Nous retiendrons le déclenchement implicite et explicite affiché. Un certain nombre de fonctions seront déclenchées implicitement (lors de la sortie d'un champ de saisie par exemple) et les autres fonctions seront déclenchées explicitement en manipulant les OI de déclenchement de l'interface associés à ces fonctions (un bouton de commande à cliquer par exemple)[14].

Le contrôle du dialogue sera mixte dans la mesure où d'une part, l'utilisateur sera entièrement libre dans l'ordonnancement de ses saisies et d'autre part, l'application effectuera certains déclenchements de fonctions implicitement une fois les conditions vérifiées.

3.1.3. Structure de l'interface de l'application

Le style d'interaction retenu étant composé de menus et de fenêtres de type formulaire, il reste encore à fixer la manière dont ils vont se structurer. Il est nécessaire de définir une structure type de l'interface si l'on veut pouvoir générer l'interface utilisateur de l'application sans devoir la spécifier.

Nous avons vu qu'une application se composait d'un certain nombre de phases qui correspondent chacune à une fonctionnalité de l'application. L'application propose donc, en quelque sorte, un certain nombre de fonctionnalités que l'utilisateur utilisera afin de réaliser sont travail. Pour permettre à l'utilisateur de manipuler les différentes fonctionnalités, celles-ci auront leur propre unité de présentation (UP)[15] qui sera accessible par un menu principal. La génération de l'interface utilisateur consistera donc à générer les UP de chaque phase et l'UP principale contenant le menu principal.

En ce qui concerne l'UP d'une phase, la solution la plus simple est d'avoir une UP composée d'une seule fenêtre, mais cela ne sera pas toujours possible. En effet, il se peut qu'il y ait trop d'informations à saisir et à afficher pour qu'on puisse toutes les mettre dans une fenêtre. Mais pour faciliter le travail, nous estimerons dans un premier temps, qu'une fenêtre est assez grande pour y mettre toutes les informations nécessaires à la

---

[13] Le déclenchement explicite affiché correspond à un OI de déclenchement de la fonction (il est donc affiché dans l'interface à travers l'OI). Le déclenchement explicite non affiché correspond par exemple à une touche de fonction à appuyer pour déclencher la fonction (il n'est donc pas affiché dans l'interface)[Bod,94].

[14] Les fonctions ne pourront être déclenchées que lorsque les conditions de déclenchement seront vérifiées.

[15] Par UP, on entend le monde de saisie et d'affichage propre à une tâche informatique [Bod94], en ce compris les fonctions de traitement de l'information utilisées. Une UP peut se composer d'une ou plusieurs fenêtres.

réalisation d'une phase. Car si nous raisonnons de façon très simpliste, utiliser plusieurs fenêtres revient à découper une fenêtre en plusieurs fenêtres de plus petite taille.

3.2. Génération de l'UP d'une phase

Avant d'aborder la génération de la présentation et du dialogue de l'UP d'une phase, nous expliquerons comment spécifier l'enchaînement des fonctions d'une phase de l'application.

3.2.1. Enchaînement des fonctions d'une phase - graphe d'enchaînement des fonctions (GEF)

Si l'on désire avoir une bonne compréhension du comportement précis d'une phase, il faut construire ce qu'on appelle la dynamique des fonctions d'une phase et ce, afin de spécifier précisément l'enchaînement des différentes fonctions qui composent une phase de l'application. Pour représenter ceci, nous utiliserons un graphe d'enchaînement des fonctions (GEF). C'est un graphe qui se construit naturellement à partir des résultats de l'étape de description des fonctions des différentes phases de l'application[16]. La différence entre ce graphe et le schéma classique de la dynamique, c'est que les enchaînements de fonctions se feront par génération de message ou par déclenchement de fonctions[17]. Le GEF permet ainsi de spécifier les conditions à respecter pour accomplir la phase en question; il illustre donc parfaitement les cheminements possibles pour réaliser la phase.

Les informations servant de base à la gestion du dialogue de l'interface associée à une phase y sont présentes. On connaît en effet les informations requises pour déclencher une fonction ainsi que les informations générées par une fonction. Et parmi ces informations, il y a celles venant et partant de l'exterieur qui correspondent aux messages externes et qui se retrouveront dans l'interface sous forme d'OI. Une fois que la présentation de l'interface associée à une phase sera définie, on pourra en spécifier la gestion du dialogue grâce au GEF, ce qui nous permettra de concevoir une interface opérationnelle pour l'UP d'une phase.

Par rapport à la statique, il faudra préciser quelques éléments notamment sur la réception des messages comme condition de déclenchement d'une fonction et la génération des messages comme terminaison d'une fonction (déterminer les connections entre les messages), sur le type de déclenchement d'une fonction (explicite ou implicite), sur son caractère simple ou répétitif et sur l'accumulation des messages.

---

16 En fait, cette étape donne implicitement un ordre partiel des fonctions au sein d'une phase.

17 Dans un schéma classique de la dynamique comme celui de la méthode IDA, l'enchaînement se fait par génération de message et par terminaison de processus.

Pour illustrer graphiquement les concepts du GEF nous utiliserons un format classique déjà utilisé dans plusieurs travaux antérieurs [Bod,94]. Les différents concepts utilisés dans le GEF sont les messages, les fonctions et les points d'accumulation[18].

Voyons tout d'abord les différents éléments qui se retrouveront dans le GEF :

Message externe ou d'erreur[19] : 

Fonctions : 

Message interne[20] : 

Point d'accumulation : 

Ces concepts de base permettent de construire le GEF en partie; il restera en effet à préciser les liens entre les messages en entrée et en sortie des fonctions. Le reste des spécifications du schéma ne se trouvera pas dans la représentation graphique, cela sera seulement écrit.

A. Messages en entrée d'une fonction

Chaque fonction reçoit en entrée des informations sous forme de messages, nécessaires au déclenchement de cette fonction. Ces messages en entrée constitueront la précondition de la fonction. Ils doivent, en effet, tous être disponibles pour que la fonction puisse s'exécuter.

Il y a plusieurs façons de combiner les informations en entrée. Dans la plupart des cas, la fonction a besoin de tous les messages en entrée et à ce moment là, les messages sont reliés par un connecteur ET. Il y a d'autres combinaisons de messages plus compliquées, mais c'est beaucoup plus rare. C'est pourquoi nous prenons le parti de ne considérer que les cas les plus fréquents afin de simplifier le graphe d'enchaînement des fonctions et donc la gestion du dialogue. Les messages en entrée seront donc tous reliés par un connecteur logique ET[21].

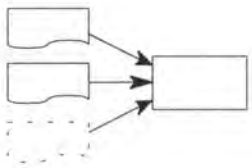

Figure 6.7 - Messages en entrée conditionnant le déclenchement d'une fonction

---

18 Un point d'accumulation est un endroit dans le GEF où l'on accumule un message résultat d'une fonction répétitive (V. plus loin).

19 Message externe en trait continu pour bien représenter le fait que c'est un message visible pour l'utilsateur [Bod,94].

20 Message interne en pointillé pour bien représenter le fait que c'est un message invisible pour l'utilsateur [Bod,94].

21 Il est de toute façon toujours possible d'étendre ultérieurement le GEF pour le rendre plus riche et pallier à ces restrictions.

Les messages en entrée se trouveront à gauche du rectangle représentant une fonction et on n'indique pas le connecteur (ils sont automatiquement reliés par un connecteur ET). La figure 6.7 représente une fonction utilisant 2 messages externes et un message interne.

B. Messages en sortie d'une fonction

Une fonction peut produire des informations en sortie sous forme de messages internes, externes ou d'erreur. Pour ces messages en sortie, il y a une combinaison de messages que l'on retrouvera tout le temps puisque qu'une fonction se termine la plupart du temps selon le même schéma. En effet, soit une fonction réussit et elle génère un message interne et d'éventuels messages externes, soit elle elle échoue et elle génère un message d'erreur. Une fonction génèrera donc un message d'erreur ou (Fig. 6.8/XOU) un message interne et (Fig. 6.8/ET) éventuellement des messages externes.

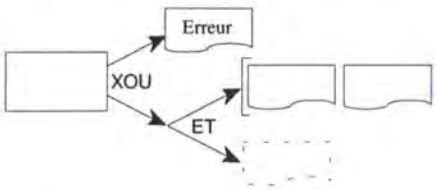

Figure 6.8 - Messages générés par une fonction

Les messages en sortie se trouveront à droite du rectangle représentant la fonction et on y indique les connecteurs. La figure 6.8, représente une fonction générant un message d'erreur en cas d'échec ou message interne et 2 messages externes en cas de succès.

C. Fonction simple et répétitive

Lors du déroulement d'une phase, une fonction peut être exécutée plusieurs fois et générer plusieurs messages de même nature qu'une autre fonction récupérera. Pour ce faire, il faudra accumuler les messages générés par ce genre de fonction (V. accumulation de messages). Cela sera spécifié dans le texte de définition du GEF.

D. Accumulation de messages

Pour accumuler les messages internes d'une fonction répétitive et ainsi fournir à la fonction qui suit l'ensemble des messages générés, il faudra spécifier un point d'accumulation. Dès qu'un message est généré, il est accumulé, c'est à dire ajouté aux messages précédents par le point d'accumulation et les messages générés sont alors disponibles pour la fonction qui les reçoit.

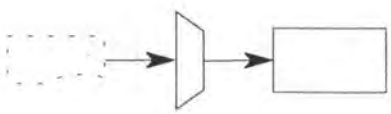

Figure 6.9 - Accumulation d'un message

Le point d'accumulation, représenté par un trapèze, se trouve entre le message généré à accumuler et la fonction qui en reçoit un ou plusieurs (Fig. 6.9).

E. Déclenchement d'une fonction

Une fonction peut être déclenchée soit explicitement, soit implicitement. Il faudra le spécifier dans le GEF car c'est une information nécessaire à la génération de l'interface utilisateur[22]. Une fonction est déclenchée explicitement lorsque l'utilisateur en demande expressément le déclenchement. Une fonction est déclenchée implicitement lorsque son déclenchement est réalisé dès que la condition de déclenchement le permet (autrement dit dès que tous les messages en entrée ont été générés). Ce mode de déclenchement ne sera pas représenté graphiquement mais sera défini dans le texte de spécification du GEF.

Nous mettons cependant 2 restrictions au déclenchement implicite. Il ne sera en effet, autorisé d'avoir un déclenchement implicite d'une fonction que si la fonction ne reçoit qu'un seul message externe et ne possède aucune accumulation de messages en entrée[23]. Ceci afin d'éviter une gestion trop complexe du déclenchement de la fonction.

F. GEF pour une phase d'enregistrement d'une commande

Maintenant que nous avons présenté tous les concepts nécessaires à la définition d'un GEF, nous allons en donner une illustration. Nous prendrons l'exemple de l'enregistrement d'une commande de produits d'un client. Lorsqu'un client commande des produits, on lui demande son numéro de client (qui l'identifie) ou ses nom et prénom. Quand le client est identifié, il faut enregistrer sa commande, c'est à dire le ou les produits commandés (numéro de produit, quantité) ainsi que le mode de paiement.

---

[22] Une fonction explicite sera en effet déclenchée par un OI de déclenchement, alors qu'une fonction implicite sera déclenchée après la saisie du message externe en entrée.

[23] Ceci afin de ne pas déclencher la fonction chaque fois qu'un message est accumulé.

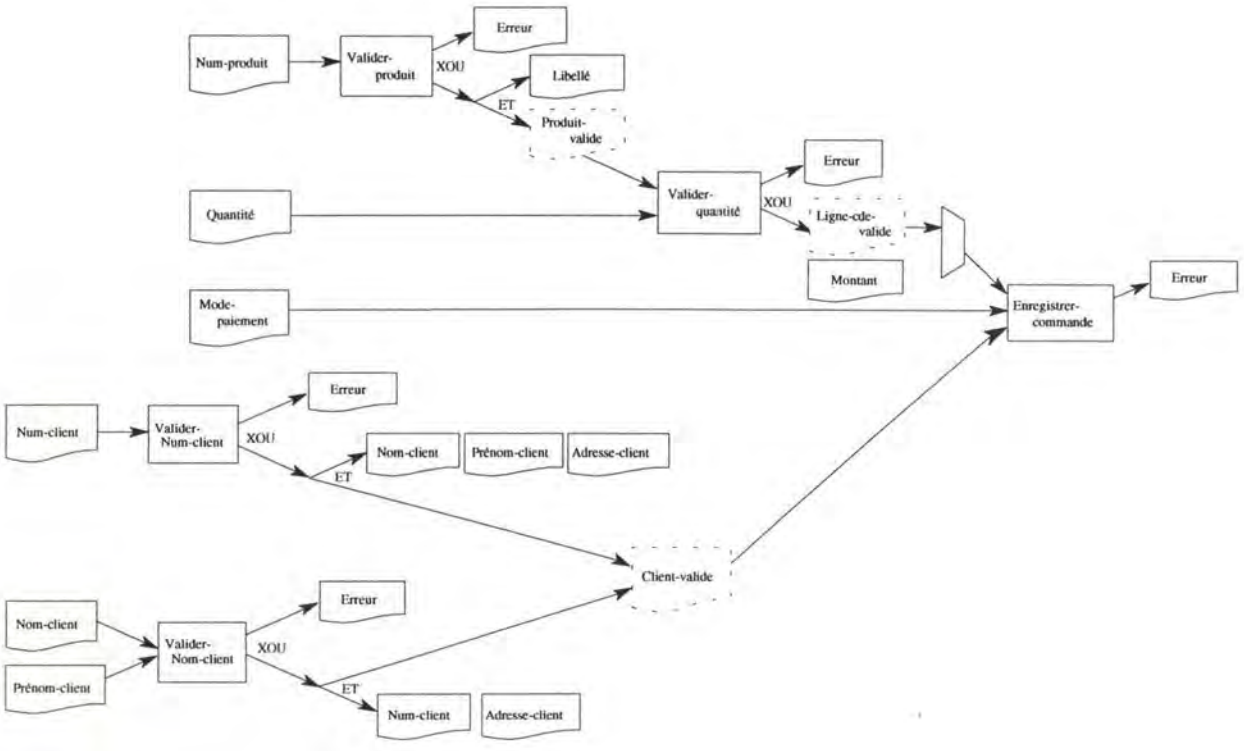

Figure 6.10 - GEF de l'enregistrement d'une commande de produits

Pour valider le numéro de client (Valider-Num-client), il faut que le message Num-client soit disponible dans l'interface (c'est à dire rentré par l'utilisateur dans l'interface). Si la fonction réussit, elle génère le message Nom-client, Prénom-client, Adresse-client et Client-valide, sinon elle génère un message d'erreur. Il y a un seul message en entrée, le déclenchement de la fonction peut donc être implicite.

Pour valider le nom et le prénom d'un client (Valider-Nom-client), il faut que les messages Nom-client et Prénom-client soient disponibles dans l'interface. Si la fonction réussit, elle génère le message Num-client, Adresse-client et Client-valide, sinon elle génère un message d'erreur. Il y a deux messages en entrée, le déclenchement de la fonction doit donc être explicite.

Pour enregistrer la commande (Enregistrer-commande), il faut que le message Client-valide ait été généré (le client est alors identifié), qu'un ou plusieurs messages Ligne-cde-valide aient été générés et accumulés et enfin que le message Mode-paiement soit disponible dans l'interface. Si la fonction échoue, elle génère un message d'erreur. Le déclenchement de la fonction doit être explicite puisqu'ayant une accumulation de messages en entrée.

3.2.2. Génération de la présentation de l'UP d'une phase

Nous avons posé l'hypothèse que le concept de phase constituait la base de la génération de l'interface. Autrement dit, nous avons considéré la génération d'une UP pour chaque phase; cette UP ne comprendra qu'une fenêtre pour plus de facilité. Les OI

de cette fenêtre correspondront aux messages externes et aux fonctions déclenchées explicitement du GEF.

La spécification conceptuelle des phases constitue une définition du contenu sémantique de l'interface de la même manière que le schéma entité-association définit conceptuellement la base de données. Les concepts du modèle entité-association sont différents et de plus haut niveau que les concepts manipulés par un SGBD relationnel. Il en est de même pour les concepts du GEF (messages externes, déclenchement explicite,...) par rapport à ceux utilisés par les outils 4GLs (champ d'édition, bouton de commande,...).

Il serait intéressant de transformer la définition conceptuelle de l'interface de chaque phase obtenue à partir du GEF en une définition abstraite composée d'objets interactifs abstraits (OIA) et conforme aux concepts manipulés par les outils 4GL. Ensuite sur base de la définition abstraite de l'interface et pour outil un 4GL particulier, on crérait le texte de définition physique de chaque fenêtre.

Si nous continuons ce parallélisme avec les données, il faut toutefois remarquer que le niveau de standardisation des SGBD est bien plus élevé que celui des 4GLs. L'étendue des concepts manipulés par les SGBD est évidemment plus restreinte que pour les 4GLs et de plus, la différence d'ancienneté de ces deux types d'outils justifie logiquement cet état de fait. Ainsi, si une standardisation au niveau de la définition des tables relationnelles est établie, il n'en est pas de même pour les OI proposés par les 4GLs. Le nombre et le type des OI sont différents d'un outil à un autre, de même pour leurs attributs et primitives. Ce manque de standardisation va impliquer deux choses. Tout d'abord, il sera moins évident que pour les données, de définir un modèle de définition abstraite commun aux différents outils. De plus la définition physique de l'interface différant d'un outil à l'autre, la génération du fichier correspondant, sur base de la définition abstraite de l'interface, sera différente pour chaque outil 4GL.

La génération de la présentation telle que nous l'envisageons va s'exécuter en deux étapes. Une première, commune à tous les outils, effectuera la construction de la définition abstraite en utilisant des règles de sélection faisant correspondre un OIA à un concept du GEF. Cette étape débutera par une extraction des informations sémantiques de l'interface du repository de l'outil d'aide à la conception. Il s'agira ensuite de sélectionner un OIA pour les concepts extraits afin d'obtenir la définition abstraite. Cette dernière sera finalement stockée dans le repository ainsi que ses correspondances avec la spécification conceptuelle.

La deuxième étape, propre à chaque outil, génère les fichiers de définition physique sur base de la définition abstraite en utilisant des règles de correspondance propres à chaque outil permettan de déterminer les OIC.

On a donc trois niveaux de définitions dont les correspondances sont illustrées par la Fig. 6.11.

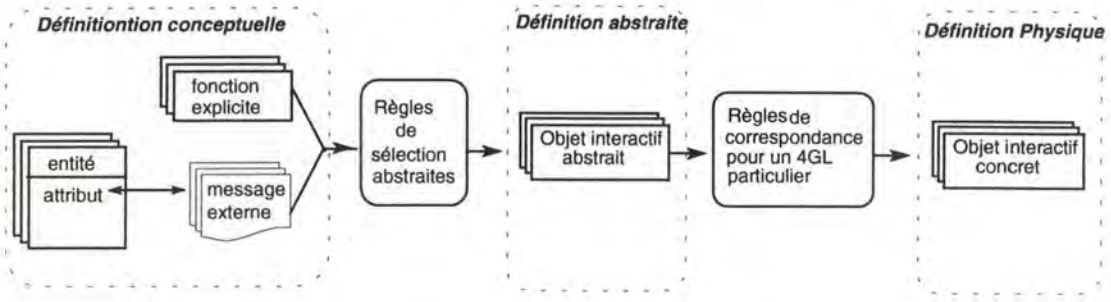

Figure 6.11 - Trois niveaux de définition de la présentation.

A. Extraction-Sélection-Stockage

Cette première étape consiste à extraire les informations sémantiques de l'interface stockées dans le repository de l'outil d'aide à la conception, afin de sélectionner les OIA et de les stocker, ainsi que les correspondances avec les informations sémantiques, dans le repository.

A.1. Extraction des informations sémantiques

Les informations nécessaires à la génération sont contenues dans le GEF et le schéma entité-association. Aux messages externes et aux fonctions déclenchées explicitement correspondra un OI.

Etant donné que nous ne considérons que les boutons de commande comme OI de déclenchement, l'information nécessaire à la génération de ces OI est suffisante avec l'identification des fonctions déclenchées explicitement.

Il en va autrement concernant les OI de saisie et d'affichage. Pour pouvoir sélectionner le type d'OIA associé à chaque message, il faudra extraire des propriétés de l'élément du schéma entité-association qui correspond à chaque message. De cette manière, un OIA adéquat pourra être défini en fonction des propriétés de cet élément.

Les propriétés des éléments qui nous intéressent sont les suivantes:

- le type de donnée parmi les types disponibles (booléen, chaîne de caractères, date, entier, heure, réel);
- le format quand il y en a un;
- le domaine, lorsqu'il est connu, et son caractère extensible ou non (par défaut, un domaine est non-extensible); on pourra en déduire le nombre de valeurs possibles; on peut préciser si ce domaine est extensible;
- le caractère obligatoire ou facultatif;
- le caractère simple ou répétitif.

Outre les messages externes et le déclenchement explicite des fonctions, une autre propriété d'un concept du GEF permettra d'identifier un OIA. Il s'agit du caractère

répétitif des fonctions. En effet, lorsqu'une fonction est répétitive, les OI de saisie et d'affichage correspondant aux messages d'entrée et de sortie sont modifiés à chaque itération de la fonction. Il serait donc intéressant d'y ajouter un tableau récapitulatif donnant une vue globale des données saisies et affichées dans lequel elles seraient accumulées.

Ces informations seront extraites par une procédure d'extraction des spécifications réalisée par l'outil d'aide à la conception. Elle aura pour résultat un fichier dont nous aurons spécifié le format de sorte que l'étape suivante de sélection puisse s'effectuer.

A.2. Identification et sélection des OIA

Pour chaque concept extrait précédemment (message externe, fonction explicite, fonction répétitive) un OIA va être créé. Un OIA va être caractérisé par un nom et deux propriétés: une première propriété que l'on qualifie d'usage de l'OIA distinguant les OIA de saisie, d'affichage et de déclenchement et une deuxième définissant le type de l'OIA.

- *Détermination de l'usage de l'OIA*

L'usage de l'OIA permettra, lors de la génération d'un OIC, de déterminer si ce dernier pourra être édité, modifié par l'utilisateur.

Information extraite	Usage
message externe en entrée d'une fonction	saisie
message externe en sortie d'une fonction répétitive	affichage
message externe en entrée et en sortie d'une fonction	saisie
fonction explicite	déclenchement

- *Détermination du type d'OIA*

En fonction de l'usage et des propriétés des éléments extraits, on va déterminer le type abstrait des OIA. Les règles que nous avons établies sont en partie basées sur [Van,93] et [Fol,91]. Classons les différents types d'OIA que nous considérons en fonction de leur usage.

Usage	Type
Saisie	champ d'édition simple et multi-lignes, interrupteur, CLOE, CLONE, CLFE, CLFNE, CEOE, CEONE, CEFE, CETNE,CEX
Affichage	champ d'édition simple et multi-lignes, Tableau
Déclenchement	Bouton

Le type d'un OIA permettra de choisir le type d'OIC de la palette du 4GL qui correspondra à l'OIA.

Légende:

Abréviation	Type complet
CLOE	Choix Limité, Obligatoire et Exclusif
CLONE	Choix Limité, Obligatoire et Non-Exclusif
CLFE	Choix Limité, Facultatif et Exclusif
CLFNE	Choix Limité, Facultatif et Non-Exclusif
CEOE	Choix Etendu, Obligatoire et Exclusif
CEONE	Choix Etendu, Obligatoire et Non-Exclusif
CEFE	Choix Etendu, Facultatif et Exclusif
CEFNE	Choix Etendu, Facultatif et Non-Exclusif
CEX	Choix EXtensible

Limité: lorsque le nombre d'options est est inférieur à 9.
Etendu : lorsque le nombre d'options est supérieur à 10.
Extensible : lorsque le choix n'est pas limité aux options présentées.

Les OIA de déclenchement sont automatiquement du type bouton de commande. On pourrait aussi imaginer d'autres types comme les items de menu. Mais ce choix des boutons de commande fait partie des restrictions que nous avons établies concernant l'interface. L'éventail des types d' OIA d'affichage et de saisie est par contre plus vaste. Nous allons exposer sous forme de tables, les règles qui permettront de sélectionner un OIA sur base de son usage et des propriétés de l'élément correspondant du modèle entité-association.

Nous présenterons tout d'abord les règles identifiant le type d'OIA d'affichage avant de voir celles concernant le type d'OIA de saisie.

OIA d'affichage

Pour définir le type de l'OIA, on utilisera le type de l'élément correspondant et son format:

Type de l'élément	Format	Type de l'OIA
chaîne de caractère	inférieur à 50	Champ d'édition simple
	supérieur à 50	Champ d'édition multi-lignes
Entier, réel, heure, date	-	Champ d'édition simple
Booléen	-	Interrupteur

OIA de saisie

Le choix du type d'un OIA de saisie dépend essentiellement du domaine de l'élément correspondant. Pour plus de clareté, nous présenterons les règles selon trois tables définies en fonction du domaine.

Domaine non-énuméré:

Lorsque le domaine de l'élément est inconnu, c'est-à-dire non-énuméré, les règles de sélection sont les même que pour les OIA d'affichage. Idem que pour un OIA de d'affichage.

Domaine énuméré et non-extensible:

La sélection est plus compliquée lorsque le domaine est énuméré. L'OIA représentera un choix. Ce choix se fera sur un nombre restreint ou étendu de valeurs possibles. Il sera obligatoire ou facultatif, et représentera une seule ou plusieurs valeurs. En fonction des propriétés des éléments on pourra déterminer le type du choix.

Type	Eventail du domaine	Obligatoire / facultatif	Simple / répétitif	Type d'OIA
Entier, réel, heure, date, chaîne de caractère	Restreint	obligatoire	simple	CLOE
			répétitf	CLONE
		facultatif	simple	CLFE
			répétitif	CLFNE
	Large	obligatoire	simple	CEOE
			répétitif	CEONE
		facultatif	simple	CEFE
			répétitif	CEFNE

Domaine énuméré, extensible:

Lorsque le domaine est extensible, on considera un seul type d'OIA.

Type de l'élément	Type de l'OIA
Entier, réel, heure, date, chaîne de caractère	CEX

Ces règles que nous avons établies peuvent évidemment être beaucoup plus complètes. Dans certains systèmes, tels que ceux sur lesquels nos règles sont inspirées [Fol,91], [trident], d'autres informations que celles que nous avons reprises sont utilisées pour déterminer le type de l'OIA. Ces informations peuvent avoir la forme de propriétés rajoutées aux éléments sémantiques ou encore d'indications sur le niveau d'expérience de l'utilisateur [Van,93].

Par exemple, on pourrait rajouter une propriété indiquant si un élément peut prendre la valeur faible ou forte de manière à donner une indication sur la précision qu'attend l'utilisateur dans la manipulation de cet élément à l'interface. Une telle information permettrait, par exemple, en cas de faible précision, de sélectionner un curseur de défilement plutôt qu'un champ d'édition pour représenter cet élément à l'écran.

Ce type d'information complémentaire permet évidemment d'être encore plus précis dans la sélection et contribue à obtenir une interface optimale. Néanmoins notre objectif,

rappelons-le, n'est pas là. En effet, nous cherchons à obtenir une interface satisfaisante sur base des informations conceptuelles. De plus l'étendue des OI proposés par les outils 4GL est limitée. Dès lors, l'ensemble de règles minimales présentées plus haut peut être considéré comme suffisant pour l'obtention d'un premier prototype d'interface.

A.3. Stockages de la définition des OIA

Chaque OIA identifié sera caractérisé par un nom et par ses deux propriétés: son usage et son type. Ils seront stockés dans le repository de l'outil d'aide à la conception, ainsi que les correspondances avec les informations sémantiques (messages, fonctions) auquels ils correspondent, de manière à fournir une définition abstraite de l'interface.

B. Génération de la présentation de la fenêtre

Il faudra transformer la définition abstraite de l'interface établie dans l'étape précédente en une définition concrète utilisable par un outil 4GL déterminé.

Il s'agira d'extraire du repository la définition abstraite de l'interface, d'établir ensuite les correspondances entre les OIA et les OI disponibles dans la palette de l'outil 4GL choisi et déterminer la valeur des attributs de ces OI.

B.1. Extraction de la définition abstraite

Une procédure d'extraction fournira la liste des OIA comprenant chacun leur nom, leur usage et leur type ainsi que le type de la valeur manipulée par l'OI. Cette procédure devra être définie et sera identique pour chaque module réalisant la génération dans un outil 4GL donné.

B.2. Correspondance entre les OIA et les OIC

Sur base du type de l'OIA, il s'agit de choisir l'OIC de la palette du 4GL qui y correspond. Si nous prenons l'exemple de Powerbuilder, on aurions la table de correspondance suivante:

Type d'OIA	Type d'OI
champ d'édition simple	Singlelinedit
champ d'édition multi-lignes	Multilinedit
interrupteur	Checkbox
CLOE	ens. de Radiobutton
CLONE	ens. de Checkbox
CLFE	ens. de Radiobutton
CLFNE	ens. de Checkbox
CEOE	Listbox
CEONE	Listbox
CEFE	Listbox
CEFNE	Listbox
CEX	Dropdownlistbox

Tableau	Listbox
Bouton	Commandbutton

B.3. Détermination des attributs

Il s'agira premièrement de déterminer la valeur des attributs rendant conforme l'OI avec sa spécification abstraite. Ainsi, l'attribut de l'objet Listbox stipulant la possibilité de sélectionner plusieurs options, devra être déterminé en fonction du type de l'OIA. Un OIA de type CEOE entraînera la détermination de l'attribut à la valeur précisant qu'une seule sélection est possible alors qu'un OIA de type CEONE entraînera l'assignation de la valeur contraire indiquant que le choix n'est pas exclusif et que plusieurs valeurs peuvent être sélectionnées. C'est donc le type de l'OIA qui intervient dans la détermination de ces attributs.

La propriété usage (saisie ou affichage) permettra quant à elle de déterminer la valeur de l'attribut précisant le caractère éditable ou non de l'OI.

La détermination des attributs de positionnement et de dimension sera effectuée à l'aide d'un algorithme de placement suivant des règles plus ou moins élaborées selon l'optique choisie. En l'occurrence, un algorithme s'occupera de déterminer la valeur des attributs concernés de manière à ce que les OI générés ne soient pas superposés les uns sur les autres. Cette gestion de la problématique du placement n'est pas envisagée ici, mais on pourra se référer à des travaux qui la traite [Kim,90].

La valeur des attributs nom et type sera donnée par les informations extraites.

Les autres attributs seront déterminés par défaut.

3.2.3. Génération du dialogue de l'UP d'une phase

Avant d'aborder la génération proprement dite, voyons d'abord en quoi consiste la gestion du dialogue.

A. Gestion du dialogue

Comme nous l'avons déjà énoncé, la gestion du dialogue assure les traductions des requêtes et données d'entrée de l'utilisateur pour l'application en fonction de l'état courant du système et les traductions des retours de l'application pour l'utilisateur [Mei, 91].

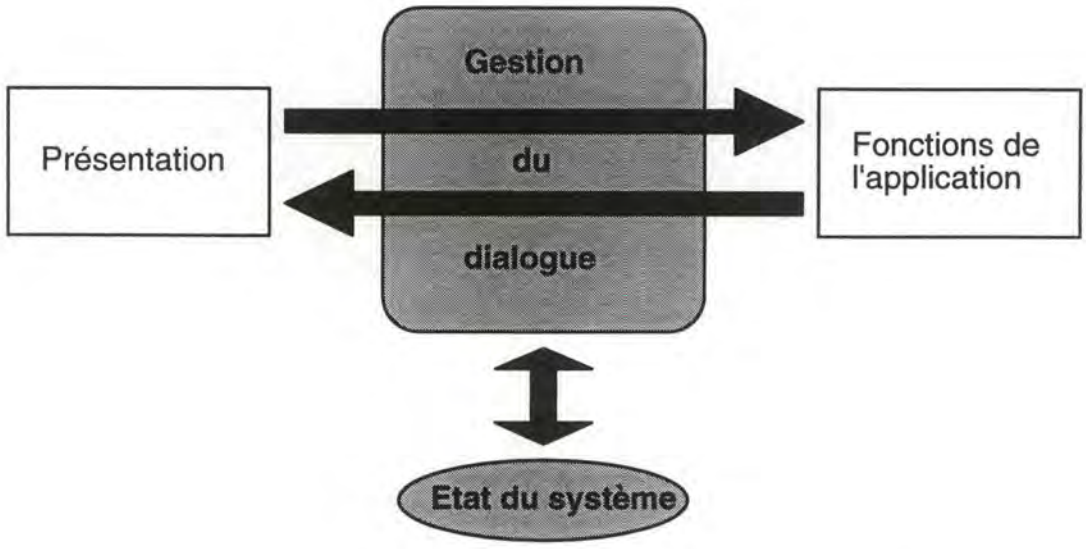

Figure 6.12- Gestion du dialogue et état du système

Le dialogue assure un rôle de liaison entre l'utilisateur d'une part, ou plus exactement la présentation qui traduit les actions de l'utilisateur, et les fonctions de l'application d'autre part, qui sont considérées comme des services appelés par le dialogue à la demande de l'utilisateur.

Pour exercer cette liaison, le dialogue utilise la notion d'état du système (Fig. 6.12). Nous allons définir ce que signifie pour nous cette nouvelle notion.

Nous exposerons ensuite les diverses missions que regroupera la gestion du dialogue qui devra être générée.

A.1. Etat du système

L'état du système pour l'UP d'un phase est constitué de l'ensemble des messages du GEF d'une phase de l'application qui participent à la précondition des fonctions. Traduire les requêtes de l'utilisateur en fonction de l'état du système revient donc à vérifier la précondition d'une fonction avant de l'appeler, c'est-à-dire consulter la partie de l'état du système constituée des messages en entrée de la fonction.

Il s'agira donc pour le dialogue de vérifier d'une part, que les messages internes en entrée ont bien été générés par les fonctions concernées et d'autre part, que les messages externes en entrée ont été correctement saisis. A tout moment, l'état courant du système est constitué du contenu de ces messages.

Cette nouvelle notion d'état du système étant éclaircie, précisons le rôle du dialogue proprement dit, tel que nous l'envisageons.

A.2. De l'utilisateur vers l'application.

- *Traduction des données*

Les données fournies par l'utilisateur au travers de l'interface vont devoir être transmises aux fonctions de l'application lorsque celles-ci seront appelées. Il s'agit donc concrètement, pour le dialogue, de traduire les données fournies via les OI de saisies et de les stocker pour pouvoir donner une valeur aux paramètres des fonctions.

Comme nous l'avons souligné précédemment, les données fournies aux fonctions devront avoir un niveau d'abstraction sémantique (et pas lexical) de manière à assurer l'indépendance interface-application. Les paramètres transmis aux fonctions correspondent aux messages d'entrée des fonctions dans le GEF. Le dialogue s'occupera donc de garnir ceux-ci avec les valeurs saisies dans l'interface.

Dans les applications de gestion de type formulaire telles que nous les envisageons[24], les données saisies par l'intermédiaire des OI ont un niveau d'abstraction sémantique. En effet, qu'il s'agisse de champs d'édition, de listes ou de boîtes à cocher, les données saisies au travers de ces OI sont de type standard (entier, chaîne de caractères,...) et donc du même niveau d'abstraction que les données traitées par les fonctions. Dès lors il n'y aura pas à proprement parler de traduction des données saisies. Le dialogue se chargera simplement de mémoriser celles-ci dans les messages externes avant de transmettre ceux-ci en paramètres aux fonctions de l'application lorsque celles-ci sont appelées.

En outre, comme nous venons de le voir plus haut, le dialogue utilise les valeurs des messages matérialisant l'état du système, pour la vérification des préconditions des fonctions , et pas les valeurs contenues dans les OI. Dès lors, il est très important de maintenir la correspondance entre les valeurs des messages externes et les valeurs affichées par les OI dans l'interface. En effet, il faut éviter un décalage entre d'une part, ce qui est affiché à l'utilisateur et d'autre part ce qui est réellement mémorisé, et qui servira notamment à la vérification des préconditions.

Pour maintenir cette cohérence, les mémorisations des valeurs d'OI dans les messages correspondant devront s'effectuer aussitôt la saisie effectuée.

- *Déclenchement des fonctions*

On vient de voir que le dialogue transmet les données en provenance de l'utilisateur vers l'application par l'intermédiaire des paramètres des fonctions. Ce transfert de données vers l'application ne se réalisera effectivement que lors de l'appel des fonctions. Le dialogue réalisera ces appels lorsque l'utilisateur en fera la demande mais également en fonction de l'état du système. Il s'assurera que le séquencement des fonctions respecte bien la spécification de l'application établie dans le GEF.

[24]Nous considérons les applications de gestion avec remplissage de formulaires.

Avant d'appeler une fonction, il s'agira pour le dialogue de s'assurer que la précondition de celle-ci est vérifiée, c'est-à-dire d'une part, que les messages internes en entrée ont bien été générés par les fonctions concernées et d'autre part, que les messages externes en entrée ont été correctement saisis.

Le but est de ne faire appel à la machine fonctionnelle que lorsque les conditions de déclenchement des fonctions sont remplies, et par le fait même, faire respecter le séquencement des fonctions établi par le GEF.

A.3. De l'application vers l'utilisateur

- *Affichage des résultats*

Après l'appel de fonction et l'exécution de celle-ci, le dialogue en assurera le retour vers l'utilisateur. En cas de succès de la fonction, il s'agira de garnir les OI d'affichage avec le contenu des messages externes de sortie de la fonction constituant les paramètres résultats de la fonction. Dans le cas contraire, une boîte de dialogue d'affichage apparaîtra et affichera le contenu du message d'erreur renvoyé également comme paramètre résultat de la fonction.

- *Expression de l'état du système*

La traduction des retours de l'application pour l'utilisateur ne devrait pas se limiter à afficher le résultat des fonctions. Une expression de l'état courant du système apportant une guidance à l'utilisateur lui serait également bénéfique. Il est d'ailleurs conseillé de rendre l'état du système explicite afin d'aider l'utilisateur notamment dans la planification de sa tâche et dans la détection des erreurs [Cou,91].

Cette expression explicite de l'état du système sera matérialisée par une distinction à l'écran des actions sémantiquement valides de celles qui ne le sont pas. Ainsi, lorsqu'une action sur un OI, ayant pour but de déclencher une fonction, est sémantiquement invalide parce que la précondition de la fonction n'est pas vérifiée, l'apparence de l'OI devra indiquer l'invalidité de l'action.

Ce type de feedback sera réalisé en ce qui concerne les boutons de commande déclenchant des fonctions. Lorsqu'une fonction ne pourra être sémantiquement déclenchée, le bouton de commande correspondant sera automatiquement "grisé". Dès que les conditions pour le déclenchement de la fonction seront remplies, le bouton sera aussitôt réactivé. Ce mécanisme ne sera pas adapté aux OI de saisie responsables du déclenchement de fonctions, et ce pour une bonne raison. En effet, les valeurs saisies par ces OI font partie de la précondition de la fonction; celle-ci n'a donc aucune chance d'être vérifiée si ces OI sont désactivés. Une telle pratique irait à l'encontre du mode de dialogue choisi, laissant l'utilisateur libre d'ordonnancer ces saisies de la manière qu'il désire. Le mécanisme d'activation-désactivation ne sera donc utilisé que pour les boutons de commande.

A.4. Gestion de la cohérence de l'état du système

Le mécanisme d'activation-désactivation des boutons de commande et la vérification systématique de la précondition des fonctions déclenchées implicitement se basent sur la notion d'état courant du système. C'est, en effet, en fonction de cet état que seront activés ou désactivés des boutons de commande et que les préconditions seront vérifiées ou non.

Il s'agit donc qu'à tout moment, les valeurs contenues dans les messages soient cohérentes. La valeur des messages externes doit être identique à la valeur affichée dans les OI correspondants; la valeur des messages internes indiquent l'état d'avancement dans le graphe d'enchaînement tout en étant cohérentes avec la valeur des messages externes.

La cohérence entre les valeurs affichées dans l'interface et les valeurs mémorisées dans les messages est maintenue aisément en garnissant les messages externes avec les valeurs des OI aussitôt les saisies effectuées. La flexibilité du dialogue offerte à l'utilisateur, lui permettant des retours en arrière, des modifications de données déjà entrées, va cependant rendre la gestion de la cohérence entre les messages nettement moins évidente à réaliser.

- *Dépendance des messages du GEF*

Si le GEF illustre l'enchaînement des fonctions ainsi que leur condition de déclenchement, il peut être vu également comme un graphe de dépendance des messages qui sont reliés entre eux par des fonctions.

Les messages de sortie d'une fonction sont considérés comme dépendants des messages d'entrée de cette fonction, dans la mesure où la valeur qui sera attribuée à ces messages dépend entièrement de la valeur des messages d'entrée de la fonction. A tout moment, la valeur des messages de sortie doit être cohérente avec les valeurs prises par les messages d'entrée.

Les fonctions s'enchaînant en utilisant des messages fournis par d'autres fonctions, nous pouvons voir le GEF comme un graphe de dépendance de messages dans lequel les sommets sont les messages et les arcs sont représentés par les fonctions. Les messages externes atomiques constituent des sommets sans précédents et les messages internes situés le plus à droite sur le GEF constituent les sommets sans suivants (voir fig. 6.13.).

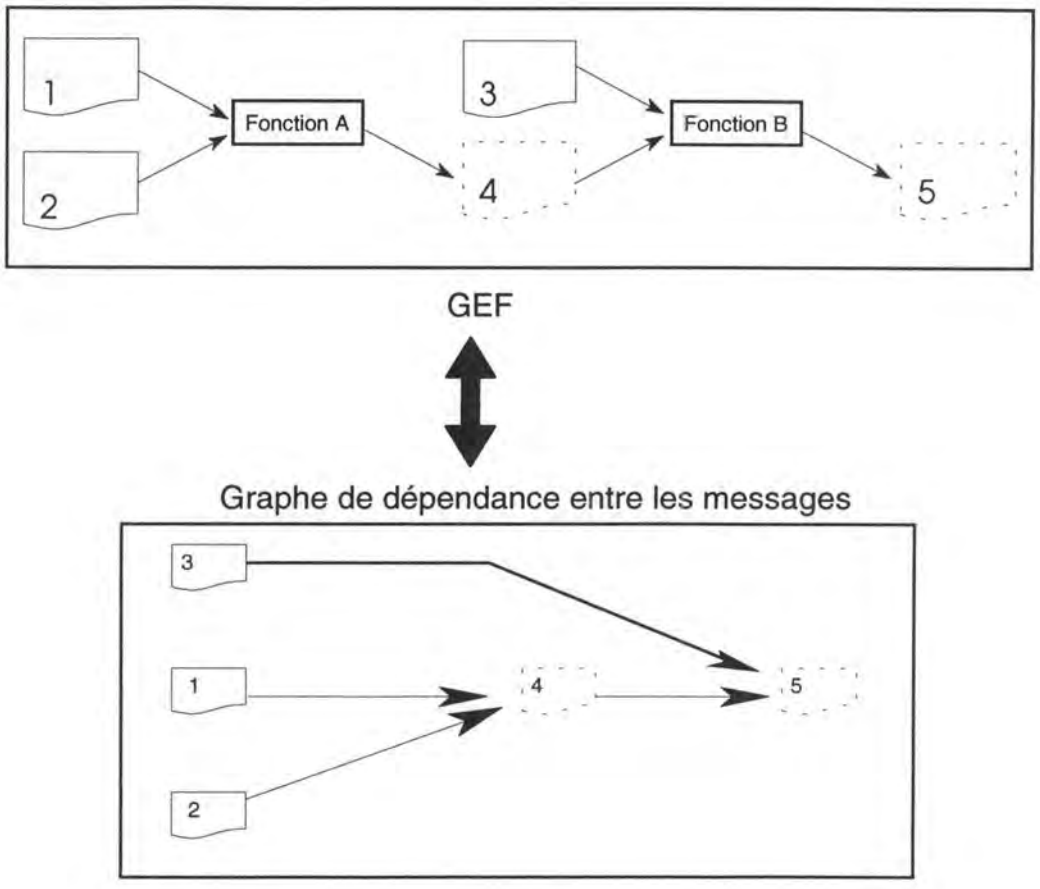

Figure 6.13- Du GEF au graphe de dépendance des mesages

Ce graphe de dépendance nous permettra de maintenir la cohérence de l'état du système grâce à l'invalidation en cascade des messages.

- *Invalidation en cascade des messages*

Lorsque le contenu d'un des messages externes du graphe de dépendance est modifié, tous les messages qui en dépendent devront être invalidés; c'est ce que nous appelons l'invalidation en cascade des messages. En l'occurence, lorsqu'une saisie est effectuée via un OI, il faudra garnir le message externe correspondant à cet OI avec la valeur saisie et invalider tous les messages de sortie dépendant directement et indirectement de ce message externe. En effet, il faut invalider les messages de sortie directement dépendant puisqu'ils correspondent à la valeur du message externe précédemment validé. De plus, compte tenu de la transitivité de la dépendance, il faut également répercuter cette invalidation sur tous les messages internes dépendant indirectement de ce message externe d'entrée, c'est-à-dire tous les messages consécutifs aux messages directement dépendants de ce message dans le graphe.

Afin d'illustrer cette invalidation en cascade, considérons l'introduction et la validation d'une ligne de commande de l'exemple de l'enregistrement d'une commande (fig. 6.14). Lorsqu'un numéro de produit est introduit, il faudra immédiatement le mémoriser dans le message Num-produit correspondant, mais également invalider le

message interne Produit_valide. En effet, la valeur de ce message correspond à un produit précédemment validé.

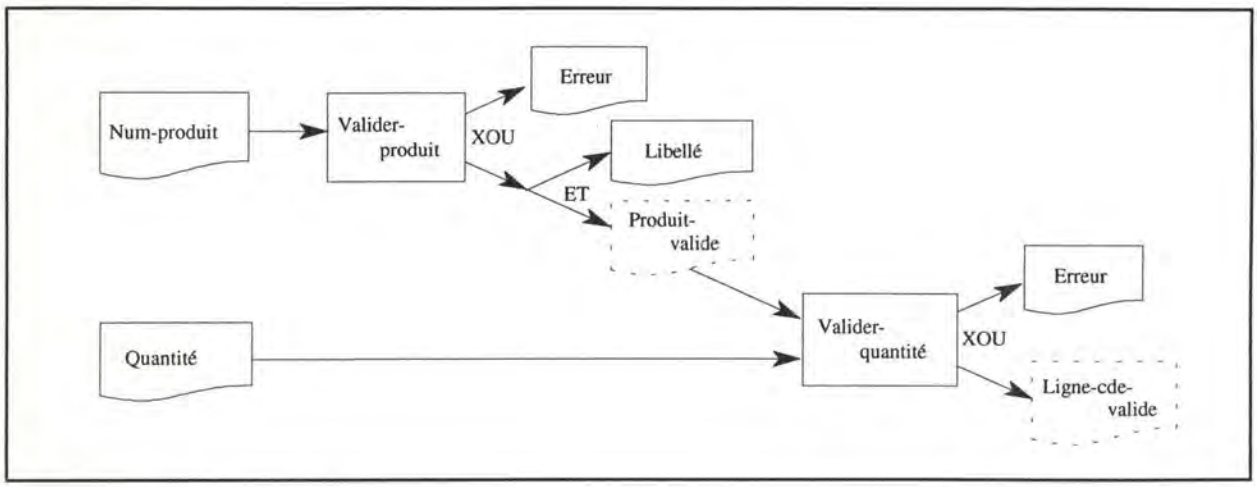

Figure 6.14- Introduction et validation d'une ligne de commande

Si cette invalidation n'est pas réalisée, la précondition de la fonction Valider_quantité peut être vérifiée si le message Quantité a déjà été saisi et la fonction peut alors être déclenchée en utilisant en entrée une ancienne valeur du message Produit-valide qui n'est plus cohérente avec le numéro de produit dernièrement saisi.

Il faudra également répercuter cette invalidation sur tous les messages dépendant indirectement de ce message d'entrée, c'est-à-dire tous les mesages consécutifs au message Produit_valide dans le graphe. Cela signifie que la saisie d'un nouveau numéro de produit impliquera, dès que la mémorisation de celui-ci dans le message Num-produit a été réalisée, l'invalidation des messages Produit-valide et Ligne-cde-valide.

La maintenance de la cohérence de l'état du système, va donc consister à répercuter automatiquement les modifications du contenu des messages externes sur tous les messages qui en dépendent.

B. Génération de la gestion du dialogue

Nous allons voir tout d'abord comment matérialiser l'état du système et la gestion de sa cohérence. Nous verrons ensuite comment les scripts associés aux OI de l'interface seront générés.

B.1. Génération de l'état du système

Les messages représensant l'état du système vont être représentés par des variables globales de sorte qu'ils soient manipulables à partir des différents scripts. La détermination du type de ces variables est identique à celle utilisée pour le type des paramètres de fonctions correspondant à ces mêmes messages.

Une variable de type "tableau" sera également générée en cas d'accumulation de messages. En effet, si le message de sortie d'une fonction répétitive sera représenté par

une variable simple, le message d'entrée de la fonction suivante, constituant une accumulation de messages de sortie de la fonction précédente, devra être représenté par un tableau contenant les valeurs des messages accumulés.

En plus de cette génération d'une variable "tableau", l'accumulation de messages entraînera également la génération d'une variable "compteur" indiquant le nombre de messages valides accumulés. Cette variable compteur sera utilisée pour accumuler le contenu des variables simples à la fin du tableau, mais également pour la vérification de la précondition de la fonction recevant cette accumulation de messages. Il s'agira de tester si le tableau contient des messages valides (compteur<>0) avant de déclencher la fonction.

B.2. Génération de la maintenance de la cohérence de l'état du système

La maintenance de la cohérence pourra s'effectuer par l'intermédiaire d'un ensemble de fonctions réalisant la mémorisation des valeurs dans les variables et l'invalidation en cascade. On qualifiera ces fonctions, de fonctions de cohérence afin d'éviter toute confusion avec les fonctions proprement dites de l'application.

Chacune des fonctions encapsulera, de manière logique, un message. Elles auront pour but d'attribuer une valeur au message encapsulé et d'appeler la ou les fonctions encapsulant les messages directement dépendants qui se chargeront, quant à elle d'appeler les fonctions qui leur sont dépendantes et ainsi de suite. L'invalidation en cascade sera réalisée de cette façon.

Les fonctions sont abstraitement organisées suivant le graphe de dépendance des messages qu'elles encapsulent. Les sommets représentent cette fois les fonctions encapsulant un message et les arcs représentent ici les appels de fonctions (Fig. 6.15).

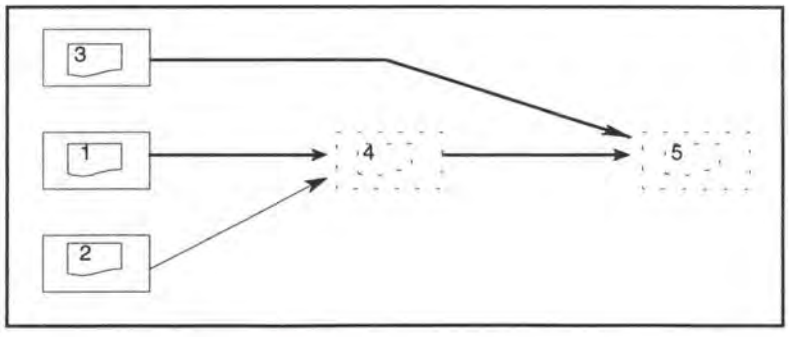

Figure 6.15 - Organisation des fonctions associées aux messages

Les fonctions associées aux messages externes mémoriseront, dans le message encapsulé, la valeur saisie via l'OI correspondant. Ces fonctions seront automatiquement appelées par les scripts, gérant le dialogue, exécutés en réponse aux événements signalant la saisie d'une information (la sortie d'un champ de saisie) (Fig. 16). La correspondance entre l'état du système et l'interface est ainsi assurée. Une fois la valeur mémorisée dans la variable, ces fonctions feront appels à ou aux fonctions directement dépendantes qui encapsulent un message interne.

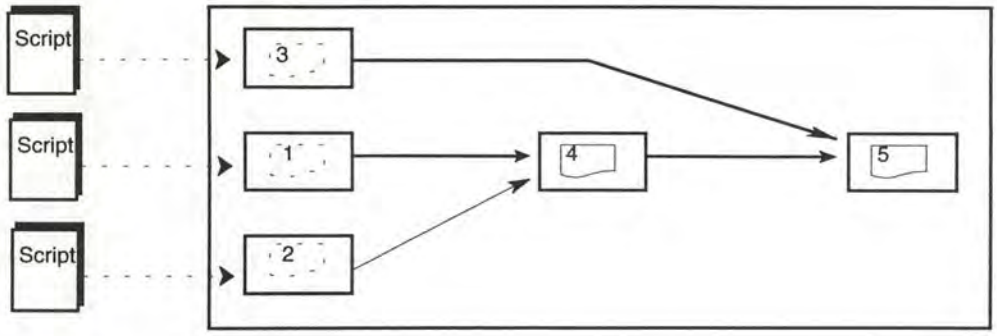

Figure 6.16-- Appel des fonctions associées aux messages par les scripts

Les fonctions associées aux messages internes auront pour but d'invalider ces messages, c'est-à-dire d'assigner la valeur "faux" au champ booléen du message indiquant sa validité. Ces fonctions seront appelées par les fonctions qui les précèdent directement dans l'arbre et ferons appel à leur tour aux fonctions directement dépendantes et ainsi de suite; ceci afin d'assurer l'invalidation en cascade des messages.

Les fonctions associées aux messages externes auront la forme suivante :

Etablir_valeur (valeur)

Message_externe := valeur

Faire appel aux fonctions directement dépendantes

Les fonction associées aux messages internes auront quant à elles la forme suivante :

Invalider ()

Message_interne.valide := "faux"

Faire appel aux fonctions directement dépendantes

Illustrons ces fonctions avec l'exemple de l'enregistrement d'une commande. Considérons les messages dépendants: Num_produit, Produit_valide et Ligne_cde_valide. Ces trois messages entraînent la génération des trois fonctions suivantes.

Etablir_valeur_num_produit (valeur)

Num_produit=valeur

Invalider_produit_valide ()

Invalider_Produit_valide ()

Produit_valide.généré="faux"

Invalider_Ligne_cde_valide()

Invalider_Ligne_cde_valide()

Ligne_cde_valide.généré="faux"

Cette couche de fonctions, directement déduite de l'analyse fonctionnelle, constitue en quelque sorte un invariant qui doit être respecté quelque soit l'interface choisie. Indépendante de tout choix d'interface, elle sera conservée en cas de modification de celle-ci. Cette couche assurant la cohérence de l'état du système étant identifiée, voyons maintenant comment la gestion du dialogue proprement dite, effectuée via les scripts, va être générée sur base du GEF.

B.3. Génération des scripts

- *Dialogue : ensemble de "micro-dialogues"*

Comme nous l'avons répété de nombreuses fois, nous sommes en présence d'un style d'interaction événementiel qui répartit la gestion du dialogue sur les différents OI de l'interface et qui sera implémentée sous forme de scripts.

En effet, entre l'application et l'utilisateur, il n'y a pas un seul dialogue contrôlé par l'application mais plutôt un ensemble de "micro-dialogues" entre l'utilisateur et les différents OI de l'interface. Ces micro-dialogues sont initiés par l'utilisateur et gérés par l'application au travers des scripts associés aux OI.

Toutefois, l'utilisateur pouvant à tout moment exécuter une multitude d'actions sur les objets de l'interface, seules les actions jugées significatives seront prises en considération. Le fait de "double-clicker" avec la souris, sur un champ d'édition par exemple, peut être considéré dans une application comme un événement à prendre en considération et déclenchera par exemple l'apparition d'une liste alors que dans une autre application, ce même événement n'entrainera pas de réaction.

Il faudra donc générer des scripts répondant aux événements jugés significatifs et qui assurent la gestion du dialogue exposée précédemment.

Cette découpe du dialogue en "micro-dialogue" n'apparaît pas dans le GEF. C'est pourquoi nous allons introduire un nouveau concept sémantique permettant de représenter sur le GEF cette découpe.

Nous appelerons ce nouveau concept : Micro-dialogue[25].

- *Le concept de Micro-dialogue*

Ce concept sémantique est destiné à modéliser la découpe du dialogue introduite par la programmation événementielle.

---

[25] Afin d'éviter une confusion, nous utiliserons un 'm' minuscule pour le terme micro-dialogue lorsque celui-ci désignera le concept réel d'intéraction élémentaire entre l'utilisateur et un O.I alors qu'un 'M' majuscule sera utilisé pour désigner le concept sémantique modélisant dans le GEF les micro-dialogues réels.

Il ne sera pas utilisé par le concepteur dans son travail de modélisation, il sera uniquement utilisé et stocké par le générateur du dialogue dans une étape intermédiaire menant à la génération des scripts. Les instances de ce concept seront superposées aux éléments (messages et fonctions) du GEF.

La génération des scripts, sur base du GEF utilisant ce concept de Micro-dialogue, respectera la démarche suivie lors de la programmation.

Avec les outils 4GL, avant d'écrire les scripts, il s'agit premièrement de construire l'interface statique (la présentation) composée des différents OI. Ensuite, il faut identifier les "micro-dialogues" significatifs, c'est-à-dire les OI et les événements de ces OI pour lesquels un script devra être écrit, et définir l'étendue de la réponse de l'application à ces événements. La sortie d'un champ de saisie doit-elle seulement entraîner une mémorisation de la donnée saisie ou bien doit-elle en plus déclencher implicitement la fonction utilisant cette donnée en entrée? Un bouton de commande doit-il déclencher une ou plusieurs fonctions? Une fois le contenu des scripts délimité, il reste à les programmer.

Ainsi, la génération des scripts suivra cette démarche. Nous avons vu dans la section traitant de la génération de la présentation que les messages externes et les fonctions déclenchées explicitement du GEF seraient représentés chacun par un OI dans l'interface. Considérant cela, il s'agira premièrement d'identifier les Micro-dialogues sur le GEF, et deuxièmement définir leur portée, c'est-à-dire les fonctions du GEF qu'ils encapsulent, déterminant ainsi l'étendue de la réaction de l'application. Finalement, en fonction du type des Micro-dialogues identifiés et de leur portée, il faudra générer les scripts correspondants.

- *Identification des Micro-dialogues*

Nous avons considéré dans la réalité deux types de "micro-dialogues" significatifs. Il y a d'une part les "micro-dialogues" initiés par la saisie d'une information au travers d'un OI de saisie (sortie d'un champ d'édition par exemple), et d'autre part les "micro-dialogues" initiés par l'utilisation d'un OI d'action (pression d'un bouton de commande en l'occurence) pour déclencher de manière explicite un traitement.

Le premier type de "micro-dialogues" pourra être modélisé et identifié sur le GEF via les messages externes d'entrée. Nous avons vu dans la génération de la présentation que ces messages correspondaient à un OI de saisie et qu'ils vont donc forcément donner lieu à un "micro-dialogue" avec l'utilisateur. La gestion du "micro-dialogue" se fera via le script associé à l'événement signifiant par exemple que le contenu de cet OI a été modifié. Chaque message externe entraîne donc la création d'un objet Micro-dialogue, qui encapsulera au moins ce message qui a permis de l'identifier. On appellera ces Micro-dialogues : Micro-dialogue (MD) de saisie.

Le deuxième type de "micro-dialogue" sera identifié via le mode de déclenchement des fonctions. En effet, les fonctions dont le mode de déclenchement est explicite, provoqueront, comme on vu précédemment, la génération d'un bouton de commande qui donnera lieu également à un "micro-dialogue". La gestion du "micro-dialogue" se fera

via le script associé à l'événement généré lors de la pression de ce bouton. Chaque fonction déclenchée explicitement entraîne également la création d'un objet Micro-dialogue qui encapsulera au moins cette fonction. On appellera ces Micro-dialogues : Micro-dialogue (MD) d'action.

Les messages d'entrée et les fonctions déclenchées explicitement permettent donc d'identifier les Micro-dialogues, c'est-à-dire concrètement, les OI et les événements de ces OI pour lesquels un script devra être généré.

Pour générer le contenu des scripts correspondants à ces Micro-dialogues, il faudra déterminer l'étendue de la réponse de l'application à cet événement déclenchant. On déterminera cette réponse en définissant la portée du Micro-dialogue sur le GEF.

- *Portée des Micro-dialogues*

Le but est ici d'établir l'étendue de la réaction de l'application à l'action de l'utilisateur qui initiera le micro-dialogue. Il s'agit de délimiter le contenu des scripts à générer. La sortie d'un champ de saisie entraînera-t-elle seulement une mémorisation de la donnée saisie dans le message correspondant, ou bien déclenchera-t-elle également une fonction utilisant cette donnée en entrée?

Cette délimitation de la réaction de l'application va s'effectuer au travers d'une détermination automatique de la portée des Micro-dialogues sur le GEF. Cette dernière utilisera la métaphore du lasso [Bod 94] et pourra être automatisée en utilisant essentiellement la notion du mode de déclenchement des fonctions. Il s'agira de déterminer les fonctions qui sont encapsulées dans les Micro-dialogues.

Détermination automatique de la portée

Dans le cas d'un Micro-dialogue de saisie, il s'agit premièrement de voir si ce Micro-dialogue encapsule la fonction dépendante du message externe d'entrée correspondant. Soit le mode de déclenchement de la fonction est explicite, et dans ce cas le Micro-dialogue n'encapsule pas la fonction. Soit le mode de déclenchement est implicite, et le Micro-dialogue encapsule alors cette fonction.

Il s'agit ensuite de se poser la même question concernant la fonction dépendante du message interne résultat de la fonction précédemment encapsulée. Si son mode de déclenchement est implicite, elle sera, elle aussi, encapsulée dans le Micro-dialogue.

Il faut continuer cette analyse en parcourant les fonctions situées sur la même "branche" du graphe, tant que le mode de déclenchement des fonctions testées est implicite et qu'une fonction terminale du GEF n'a pas été atteinte.

Il arrivera des situations dans lesquelles des Micro-dialogues encapsulent des fonctions communes. Ce type de situation se retrouvera lorsque lorsqu'une fonction a un mode de déclenchement implicite et qu'elle possède plusieurs messages en entrée, un externe et un ou plusieurs internes. Cette situation sera illustrée dans l'exemple de la ligne commande (Fig. 6.16).

Dans le cas d'un Micro-dialogue d'action, le raisonnement est identique. Il s'agira également d'encapsuler les fonctions situées sur la même "branche" de l'arbre dont le mode de déclenchement est implicite. On analysera donc les fonctions dépendantes tant que leur mode de déclenchement est implicite et que l'on a pas atteint une fonction terminale du graphe.

Le résultat de la détermination automatique de la portée des Micro-dialogues sur le GEF de l'enregistrement d'une commande, est illustré par la fig. 6.17. Pour distinguer les fonctions dont le mode de déclenchement déclaré est explicite, de celles dont le mode est implicite, on a marqué d'un point noir les fonctions déclenchées explicitement.

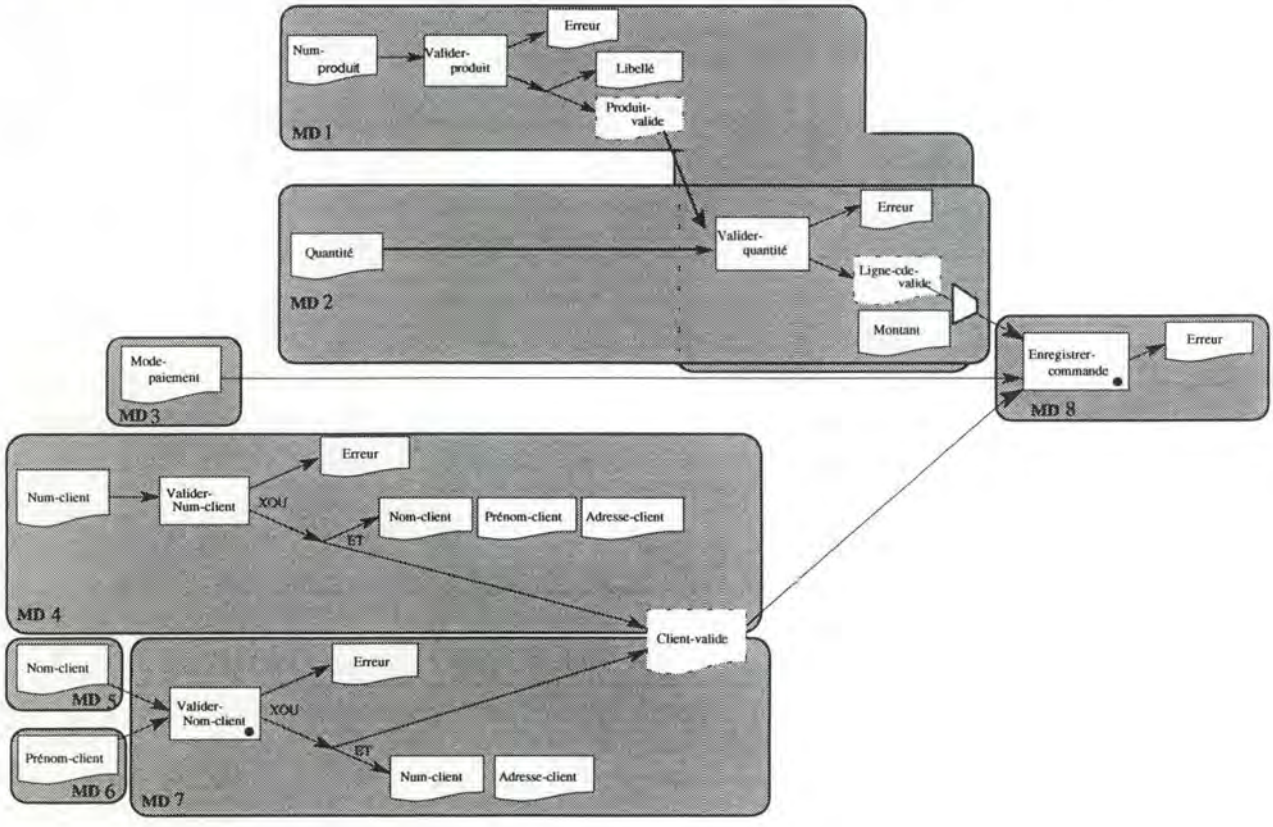

Figure 6.17 - Identification et portée des Micro-dialogues.

On remarque que la fonction Valider-quantité, déclenchée implicitement, se trouve encapsulée à la fois dans le Micro-dialogue MD1 et le Micro-dialogue MD2.

Cette double encapsulation est nécessaire si l'on veut que le déclenchement implicite de cette fonction soit correctement exécuté, c'est-à-dire que la fonction soit automatiquement exécutée une fois les deux messages correctement saisis. L'ordre des saisies n'étant pas déterminé, l'utilisateur peut très bien remplir la quantité avant le numéro du produit, et inversément. Or la fonction doit être déclenchée dès que les deux messages ont été saisis et qu'ils sont corrects. Dès lors, la vérification de la précondition et l'appel à la fonction devront se trouver dans les deux scripts associés respectivement à la saisie du numéro et à la saisie de la quantité.

La définition automatique de la portée a permis de délimiter, pour chaque Micro-dialogue identifié, les éventuelles fonctions déclenchées implicitement qu'ils encapsulent. Outre cette détermination automatique de la portée classe les Micro-dialogues en trois types et met en évidence une relation de dépendance entre les Micro-dialogues.

Trois types de Micro-dialogues.

Dans la section concernant l'identification des Micro-dialogues, nous avions distingué deux catégories de Micro-dialogues. D'une part, les Micro-dialogues de saisie, et d'autre part, les Micro-dialogues d'action.

Après la définition de la portée, on remarque que les Micro-dialogues de saisie se distinguent en fait en deux types. Premièrement, les Micro-dialogues n'encapsulant aucune fonction et deuxièmement, les Micro-dialogues encapsulant une, voire plusieurs fonctions déclenchées implicitement. A ces deux types de Micro-dialogues de saisie s'ajoute le type des Micro-dialogues d'action.

Nous verrons ultérieurement, dans la section consacrée à la définition des scripts, que c'est le type du Micro-dialogue qui déterminera la structure du script qui sera généré.

Dépendance des Micro-dialogues

Le GEF introduit des relations de dépendances entre ses différents concepts, selon l'interprétation que l'on en fait. Nous avons identifié précédemment la dépendance entre les messages du GEF qui nécessitait une maintenance de la cohérence entre ces différents messages. L'introduction du concept de Micro-dialogue, se superposant en quelque sorte aux concepts du GEF, introduit une nouvelle relation de dépendance.

Cette dépendance entre les Micro-dialogues est représentée par un graphe calqué à nouveau sur la structure du GEF. Les Micro-dialogues vont constituer les sommets du graphe et les flèches allant d'un message encapsulé dans un Micro-dialogue vers une fonction encapsulée dans un autre Micro-dialogue constitueront les arcs du graphe.

Les Micro-dialogues associés aux messages externes constituent des sommets sans précédents. Les Micro-dialogues encapsulant les fonctions les plus à droite dans le GEF, représentent quant à eux les sommets sans suivants.

Illustrons cette dépendance des Micro-dialogues en reprenant la Fig.6.17. et en faisant apparaître clairement ce graphe de dépendance sur la Fig. 6.18.

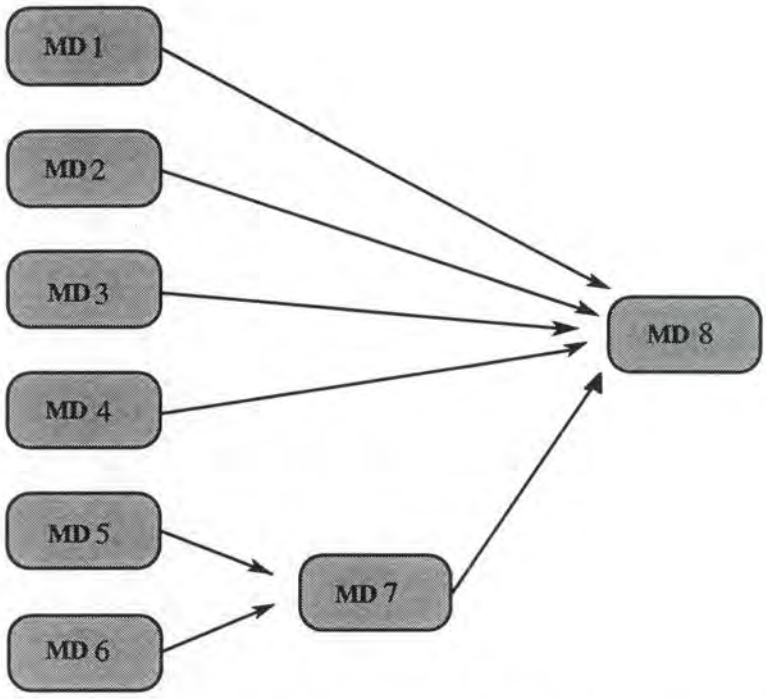

Figure 6.18- Graphe de dépendance des Micro-dialogues.

Cette relation de dépendance entre les Micro-dialogues du GEF modélise en fait cette même relation existant entre les "micro-dialogues" de la réalité. En effet, la pression d'un bouton de commande, déclenchant une fonction, n'est une action valide que lorsque les données nécessaires à l'exécution de la fonction sont disponibles. Le "micro-dialogue" d'action, initié par la pression du bouton de commande et déclenchant la fonction, est donc en quelque sorte dépendant, d'une part, des "micro-dialogues" de saisies réalisant la récolte des informations nécessaires à l'exécution fonction, et d'autre part, des éventuels autres "micro-dialogues" d'action déclenchant des fonctions devant être exécutées antérieurement de manière à respecter les contraintes établies par le GEF.

Cette relation de dépendance entre les "micro-dialogues" réels sera matérialisée, comme nous l'avons déjà énoncé, par le mécanisme d'activation-désactivation des boutons de commande. Rappelons qu'il s'agit de veiller, à tout moment, à activer les boutons de commande déclenchant des fonctions dont les messages en entrée sont correctement garnis (c'est-à-dire non-vides pour les messages externes, et valides pour les internes), et à désactiver les autres.

La relation de dépendance entre les Micro-dialogues du GEF, formalisant en quelque sorte la dépendance entre les "micro-dialogues" réels, va donc servir de base pour générer ce mécanisme d'activation-désactivation.

Après avoir vu premièrement comment identifier les Micro-dialogues, nous avons montré ensuite comment la détermination automatique de la portée des Micro-dialogues s'effectuait. Nous avons également souligné la classificarion en trois types des Micro-dialogues ainsi que la relation de dépendance existant entre eux. Voyons maintenant comment les scripts vont être générés sur base de ces Micro-dialogues.

Avant cela, précisons le choix effectué concernant la mise en oeuvre du mécanisme d'activation-désactivation des boutons.

- *Mécanisme d'activation-désactivation*

Le mécanisme sera mis en oeuvre par l'intermédiaire d'événements que les OI s'enverront entre eux.

On a vu que le graphe de dépendance des Micro-dialogues modélisait en quelque sorte la dépendance réelle existant entre les OI de saisies et les boutons de commande en vue du respect de l'invariant fonctionnel établi par le GEF.

Un bouton ne devra être activé que lorsque les informations en entrée de sa ou ses fonctions sont disponibles. Ainsi, lorsqu'une donnée est saisie au travers d'un OI, ce dernier envoie dans un premier temps un événement de désactivation à tous les boutons de commande dépendants. Ces derniers, recevant cet événement, se désactivent et transmettent à leur tour cet événement de désactivation aux boutons de commande qui leur sont dépendants de manière à désactivater tous les boutons directement ou indirectement dépendant de l'OI de saisie.

Dans un deuxième temps, l'OI de saisie mémorise la donnée saisie dans le message correspondant, et envoie un événement d'activation aux boutons dépendants. Recevant cet événement, les boutons vont tester la validité des messages en entrée des fonctions qu'ils encapsulent. Si ces messages sont valides, ils se réactiveront. Un événement d'activation ne sera envoyé aux boutons dépendants de ces derniers, que lorsque la fonction encapsulée aura été exécutée avec succès.

Dans l'exemple de la commande, dès que le nom du client est saisi, un événement de désactivation est envoyé au bouton valider_client. Ce dernier se désactive et transmet l'événement au bouton enregistrer_commande qui se désactive également. Le nom du client est ensuite mémorisé dans le message correspondant et un événement d'activation est envoyé au bouton valider_client. Celui teste la validité du contenu du message nom et du message prénom. Si ces deux messages sont non-vides, le bouton s'active. A ce stade il est inutile d'envoyer un événement d'activation au bouton mémoriser_commande alors que la fonction de validation associée au bouton valider_client n'a pas encore été exécutée. En effet, le message de sortie (Client_valide) de cette fonction, nécessaire à l'activation du bouton enregistrer_commande, n'a pu être généré. Cet événement d'activation ne sera envoyé qu'en cas de pression du bouton valider_client et de succès de la fonction déclenchée.

La mise en oeuvre de ce mécanisme étant exposée, définissons maintenant la structure des scripts qui seront générés sur base des Micro-dialogues.

- *Les scripts*

Nous avons vu comment, sur base du GEF, générer une déclaration de fonction, constituée du nom de la fonction et des paramètres. Nous avons également expliqué qu'à un message externe et à une fonction déclenchée explicitement était associé un OI, et qu'à un message était associé une variable globale. Nous avons également mis en évidence la génération d'un ensemble de fonctions de cohérence, associées aux messages, assurant la maintenance de la cohérence entre ces messages. Ces différents concepts, dont on a expliqué la génération sur base des éléments du GEF, vont être manipulés par des scripts.

C'est le concept de Micro-dialogue, partitionnant le GEF, qui permet d'établir la liaison avec le concept de script.

Premièrement, l'élément du GEF qui a permis d'identifier un Micro-dialogue déterminera l'OI et l'événement de cet OI pour lequel un script sera généré. Si l'élément du GEF est une fonction déclenchée explicitement, l'OI sera le bouton correspondant et l'événement sera celui signalant la pression du bouton. Si l'élément du GEF est un message externe d'entrée, le type de l'OI et l'événement associé dépendront du résultat de la génération de la présentation. Pour un champ de saisie, l'événement déclenchant sera celui signalant la modification du contenu de l'OI. Pour une liste de sélection, ce sera l'événement signalant la sélection d'un élément de la liste.

Deuxièmement, la portée des Micro-dialogues, exprimée en terme de fonctions encapsulées, va permettre de délimiter le contenu des scripts, c'est à dire les concepts qu'ils vont manipuler. En effet, pour un Micro-dialogue, le fait d'encapsuler une fonction n'établit pas seulement un lien entre lui-même et cette dernière. Indirectement le Micro-dialogue est également en relation avec les messages d'entrée et sortie de la fonction. A partir d'un Micro-dialogue, on peut donc accéder aux différents OI, variables, et fonctions générés sur base des messages et fonctions du GEF.

Troisièmement, la dépendance entre les Micro-dialogues va permettre de mettre en oeuvre le mécanisme d'activation-désactivation des boutons.

Enfin, le type des Micro-dialogues, va définir la structure suivant laquelle les scripts vont être construits.

Dans ce qui suit nous allons exposer la structure des trois types de scripts correspondant aux trois types de Micro-dialogues. Les deux scripts répondant aux événements d'activation et désactivation seront également présentés.

Pour chaque type de script, nous en ferons tout d'abord une première description. Nous ferons également un rappel, sous forme graphique, des liens existant entre les concepts du GEF, encapsulés dans le Micro-dialogue correspondant, et les concepts manipulés dans le script. Ensuite, tout en faisant abstraction d'un langage 4GL en

particulier, nous définirons la structure du script un peu plus formellement dans une sorte de pseudo-langage. Les parties variables du script, générées à partir des informations du GEF, encapsulées dans les Micro-dialogues, seront affichées en gras de manière à faire la liaison avec le rappel graphique précédent. Un exemple de script, basé sur l'exemple de l'enregistrement d'une commande, illustrera finalement chaque type de script. Cet exemple de script sera réalisé à la fois en pseudo-code, et en Powerbuilder.

Scripts associés aux Micro-dialogues de type 1

Rappelons que ces Micro-dialogues sont identifiés par un message externe d'entrée et qu'ils n'encapsulent aucune fonction.

Description du script

Le script à générer sera associé à l'OI correspondant au message externe et, plus concrètement, à l'événement signalant la modification de celui-ci. Il s'agira tout d'abord de désactiver tous les boutons de commande dépendants en leur envoyant l'événement de désactivation. Il faudra ensuite mémoriser la valeur saisie par l'OI dans la variable correspondante et entraîner l'invalidation en cascade des variables dépendantes. La mémorisation et l'invalidation en cascade se feront via l'appel à la fonction de cohérence correspondant à cette variable. Il s'agira finalement d'envoyer l'événement activation à tous les boutons dépendants.

Liens entre les concepts du Micro-dialogue et ceux du script

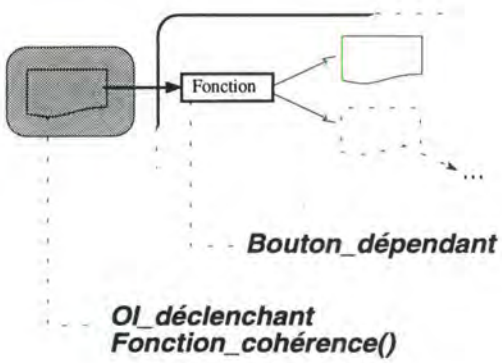

Figure 6.19 - Micro-dialogues et scripts de type 1

OI_déclenchant : OI de saisie associé au message.

Evénement déclenchant : événement signifiant la modification de l'OI déclenchant.

Fonction_cohérence() : fonction associée au message.

Bouton_dépendant : bouton de commande associé aux Micro-dialogues dépendants.

Script

{Pour tous ***boutons_dépendant*** Faire}

{Générer événement (Désactivation) pour ***Bouton_dépendant***}

Appeler fonction (***fonction_cohérence (OI_déclenchant)***)

{Pour tous ***bouton_dépendant*** Faire}

{Générer événement (Activation) pour ***Bouton_dépendant***}

Exemple

Prenons l'exemple du script associé au Micro-dialogue (MD5) identifié par le message Nom_client (fig. 6.17).

En pseudo-code:

Générer événement (désactivation) pour ***Bt_Valider_Nom_client***

Appeler fonction (***Etablir_valeur_nom_client (OI_Nom_client)***)

Générer événement (activation) pour ***Bt_Valider_Nom_client***

En Powerbuilder:

```
triggerevent(cb_valider_nom_client,"desactivation")
etab_nom (sle_nom_cli.text)
triggerevent(cb_valider_nom_client,"activation")
```

Scripts associés aux Micro-dialogues de type 2

Rappelons que ces Micro-dialogues sont identifiés par un message externe et qu'ils encapsulent une, voire plusieurs fonctions déclenchées implicitement.

Description du script.

Le script à générer sera associé à l'OI correspondant au message externe et, plus concrètement, à l'événement signalant la modification de celui-ci. Il s'agira tout d'abord de désactiver tous les boutons de commande dépendants en leur envoyant l'événement de désactivation. Il faudra ensuite mémoriser la valeur saisie par l'OI dans la variable correspondante et entraîner l'invalidation en cascade des variables dépendantes. La mémorisation et l'invalidation en cascade se feront via l'appel à la fonction de cohérence correspondant au message.

Ensuite, il faudra appeler la fonction déclenchée implicitement si sa précondition est vérifiée. Si la fonction s'est bien terminée, on affichera les résultats s'il y en a, et s'il y a lieu on accumulera le contenu de la variable dans une variable tableau utilisée par la fonction suivante. On vérifiera ensuite la précondition de la fonction suivante et, en cas de succès, on répètera les mêmes opérations. Dans le cas contraire, une boîte de dialogue affichera le message d'erreur.

Il s'agira finalement d'envoyer l'événement Activation à tous les boutons dépendants. Cet événement n'est envoyé qu'en cas de succès de la ou des fonctions.

Liens entre les concepts du Micro-dialogue et ceux du script.

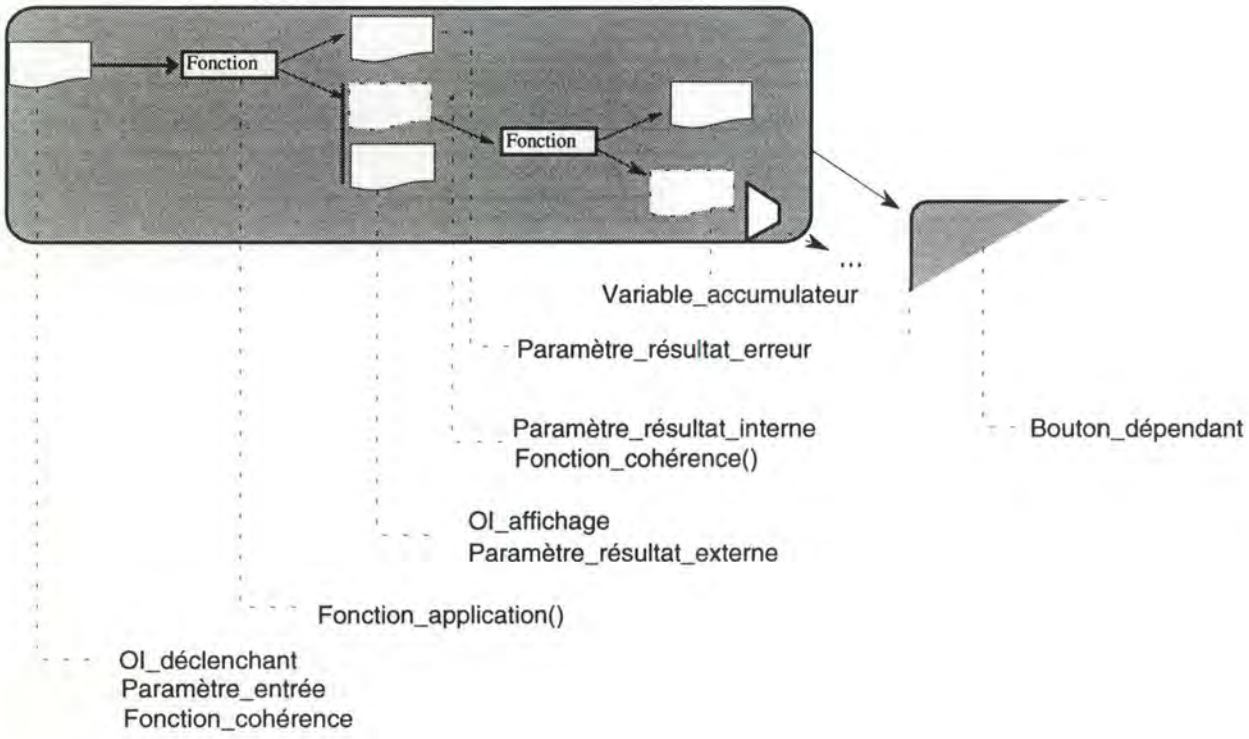

Figure 6.20- Micro-dialogues et scripts de types 2.

OI_déclenchant: OI de saisie associé au message.

Evénement déclenchant: événement signifiant la modification de l'OI déclenchant.

Fonction_cohérence(): fonction associée au message.

Bouton_dépendant: boutons de commande associés aux Micro-dialogues dépendants.

Fonction_encapsulée(): fonction de l'application.

Paramètre_entrée: variable globale associée au message externe d'entrée de la fonction.

Paramètre_résultat_interne:.variable globale associée au message interne de sortie de la fonction.

Paramètre_résultat_externe: variable globale associée au message externe de sortie de la fonction.

Paramètres_résultat_erreur : variable globale associée au message d'erreur de la fonction.

Paramètres : ensemble des paramètres.

Variable_accumulateur : variable globale associée au message interne de sortie de la fonction répétitive.

OI_affichage : OI d'affichage associé au message externe de sortie de la fonction.

Script.

{Pour chaque ***Bouton_dépendant*** Faire}
{Générer événement (Désactivation) pour ***Bouton_dépendant***}
Appeler fonction ***Fonction_cohérence (OI_déclenchant)***
Si ***Paramètres_entrée*** OK Alors
Appeler fonction ***Fonction_application (paramètres)***
Si ***paramètre_résultat_interne.généré="*** OK" Alors
{Afficher ***paramètre_résultat-externe*** dans ***OI_affichage***}
{Accumuler ***paramètre_résultat_interne*** dans ***Accumulateur***}
{*TRAITER FONCTION SUIVANTE*}
{Pour chaque ***Bouton_dépendant*** Faire}
{Générer événement (activation) pour ***Bouton_dépendant***}
Sinon
Afficher ***paramètre_résultat_erreur*** dans Boîte_d'avertissement
Fsi
Fsi

Exemple.

Illustrons ce type de script par deux exemples. Le premier, le plus simple correspond au script associé au Micro-dialogue MD4, identifié par le message Num_client et n'encapsulant que la fonction Valider_Num_client (Voir Fig. 6.17). Le second script, plus complexe, est celui associé au Micro-dialogue MD1 identifié par le message Num_produit et encapsulant les fonctions répétitives Valider_produit et Valider_quantité.

1. MD4.

En pseudo-code:

Générer événement (Désactivation) pour ***Bt_Enregistrer_commande***
Appeler fonction (***Etablir_valeur_num_client (OI_num_produit)***)
SI ***Num_client***<>""ALORS
Appeler fonction (***Valider_Num_client (Num_client, Nom_client, Prénom_client, Adresse_client, Client_valide, Erreur_client)***)

SI ***Client_valide.généré***="OK" ALORs
Afficher ***Nom_client*** dans ***OI_Nom_client***
Afficher ***Prénom_client*** dans ***OI_Prénom_client***
Afficher ***Adresse_client*** dans ***OI_Adresse_client***
Générer événement (activation) pour ***Bt_Enregistrer_commande***
SINON
Afficher ***Erreur_client*** dans boîte d'avertissement
FSI
FSI

En Powerbuilder:

```
triggerevent(cb_enregistrement,"desactiver")
etab_num_cli ( sle_num_cli.text )
IF num_cli<>"" THEN
    valider_num_client(num_cli,client_valide,nom_cli,prenom_cli,adresse_cli,
                                    erreur_client)
    IF client_valide.genere="OK" THEN
            sle_nom_cli.text=nom_cli
            sle_prenom_cli.text=prenom_cli
            mle_adresse_cli.text=adresse_cli
            triggerevent(cb_enregistrer,"activation")
    ELSE
            messagebox("Message d'erreur",erreur_client,Exclamation!)
    END IF
END IF
triggerevent(cb_enregistrement,"activation")
```

2.MD1.

En pseudo-code.

Générer événement (Désactivation) pour ***Bt_Enregistrer_commande***
Appeler fonction (***Etablir_valeur_num_produit (OI_num_produit)***)
SI ***Num_produit***<>""ALORS
Appeler fonction (***Valider_produit (Num_produit, Libellé, Produit_valide, Erreur_produit)***)
SI ***Produit_valide.généré***="OK" ALORs
Afficher ***Libellé*** dans ***OI_libellé***
SI Quantité<>"" ALORS

```
            Appeler fonction (Valider_quantité (Produit_valide, Quantité,Montant,
                                Ligne_cde_valide, Erreur_quantité))
            SI Ligne_cde_valide.généré="OK" ALORS
              Afficher Montant_ligne dans OI_montant
                    Nbre_ligne_cde_valide=Nbre_ligne_cde_valide + 1
              Lignes_cde_valides(Nbre_ligne_cde_valide)=Ligne_cde_valide
              Afficher Num_produit+ Libellé + Quantité + Montant     dans
                    OI_tableau_lignes_cde
              Générer événement (activation) pour Bt_Enregistrer_commande
            SINON
              Afficher Erreur_quantité dans boîte d'avertissement
            FSI
          FSI
        SINON
          Afficher Erreur_produit dans boîte d'avertissement
        FSI
      FSI
```

En Powerbuilder.

```
triggerevent(cb_enregistrer_commande,"desactivation")
etab_num_prod ( sle_num_prod.text )
IF num_prod<>"" THEN
    Valider_produit(num_prod,lib_prod,produit_valide,erreur_produit)
    IF produit_valide.genere="OK" THEN
      sle_lib_prod.text=lib_prod
      IF qte_prod<>"" THEN
        Valider_quantite(produit_valide,qte_prod,montant_ligne,ligne_cde_valide,
              erreur_quantité)
        IF ligne_cde_valide.genere="ok" THEN
          sle_mt_ligne.text=montant_ligne
          Nbre_ligne_cde_valide=Nbre_ligne_cde_valide + 1
          Tab_ligne_cde_valide(Nbre_ligne_cde_valide)=ligne_cde_valide
          ligne=num_prod + lib_prod + qte_prod + montant_ligne
          lb_ligne_commande.additem(ligne)
          triggerevent(cb_enregistrement,"activation")
        ELSE
          messagebox("Message d'erreur",erreur_quantité,Exclamation!)
        END IF
```

```
            END IF
        ELSE
            messagebox("Message d'erreur",erreur_produit,Exclamation!)
        END IF
    END IF
```

Scripts associés aux Micro-dialogues de type 3

Rappelons que ces Micro-dialogues sont associés à une fonction explicite et qu'ils encapsulent une, voire plusieurs autres fonctions déclenchées implicitement.

Description des scripts

Un script "principal" et deux autres scripts répondant aux événements activation et désactivation seront générés.

Le script "principal" à générer sera associé au bouton de commande correspondant à la fonction déclenchée explicitement et, plus concrètement, à l'événement signalant la pression de celui-ci. Il s'agira tout d'abord de désactiver le bouton de commande. Il faudra ensuite appeler la fonction déclenchée explicitement. Si la fonction s'est bien terminée, on affichera les résultats s'il y en a, on réalisera l'accumulation s'il y a lieu et on appelera l'éventuelle fonction suivante. Dans le cas contraire, on affichera une boîte de dialogue affichant le message d'erreur. Il s'agira finalement d'envoyer l'événement activation à tous les boutons dépendants. Cet événement n'est envoyé qu'en cas de succès de la ou des fonctions.

En ce qui concerne le script correspondant à l'événement d'activation, il s'agira d'activer le bouton seulement si les variables nécessaires à l'activation sont correctement remplies. Il s'agit des variables correspondant aux messages reçus par les fonctions du Micro-dialogue. et qui sont externes ou proviennent de fonctions non-encapsulées dans le Micro-dialogue. Pour le script désactivation, il s'agira de désactiver le bouton et d'envoyer un événement désactivation aux boutons dépendants.

Liens entre les concepts du Micro-dialogue et ceux du script

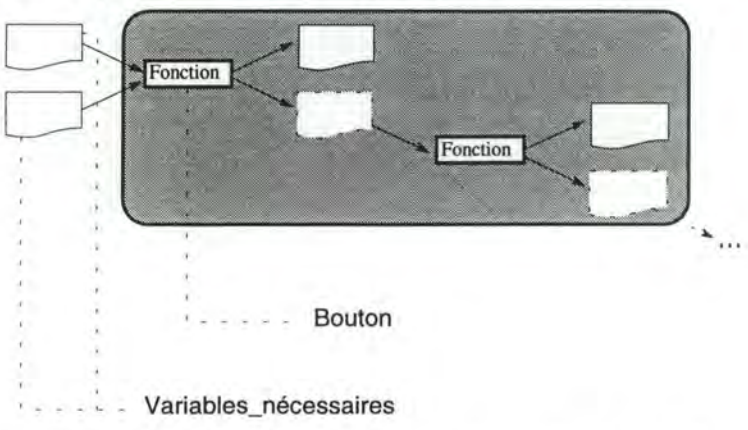

Figure 6.21- Micro-dialogues et scripts de type 3.

On n'a fait apparaître dans la Fig. 6.21 que les concepts nouveaux par rapport à ceux déjà identifiés dans les scripts du type précédent.

Bouton : bouton de commande associé à la fonction déclenchée explicitement.

Variables_nécessaires : variables globales associées aux messages d'entrée des fonctions encapsulées.

Scripts.

• Script associé à la pression du bouton de commande:

Désactiver ***bouton***
Appeler fonction ***Fonction_application (paramètres)***
Si ***paramètre_résultat_interne*** OK Alors
{Afficher ***paramètre_résultat-externe*** dans ***OI_affichage***}
{Accumuler ***paramètre_résultat_interne*** dans ***Accumulateur***}
{*TRAITER FONCTION SUIVANTE*}
{Pour chaque ***Bouton_dépendant*** Faire}
{Générer événement (activation) pour ***Bouton_dépendant***}
Sinon
Afficher ***paramètre_résultat_erreur*** dans Boîte_d'avertissement
Fsi

•. Script (activation):

Si ***Variables_nécessaires*** OK Alors
Activer ***Bouton***

• Script (désactivation):

Désactiver ***Bouton***
Pour tous ***Bouton_dépendan*** Faire
Générer événement (Désactivation) pour ***Bouton_dépendant***

Exemple.

Illustrons ces types de script par l'exemple des scripts associés au Micro-dialogue MD7 identifié par la fonction explicite Valider_Nom_client (Voir Fig. 6.17).

En pseudo-code.

•Script associé à la pression du bouton Bt_valider_nom_client:

Désactiver ***Bt_valider_Nom_client***

Appeler fonction (***Valider_Nom_client (Nom_client, Prénom_client, Num_client, Adresse_client, Erreur_client)***)

SI ***Client_valide.généré***="OK" ALORS

Afficher ***Num_client*** dans ***OI_Num_client***

Afficher ***Adresse_client*** dans ***OI_Adresse_client***

Générer événement (activation) pour ***Bt_Enregistrement_commande***

SINON

Afficher ***Erreur_client*** dans boîte d'avertissement

FSI

•Script associé à l'événement activation:

Si Nom_client <>"" ET Prénom_client<>"" ALORS

Activer ***Bt_valider_Nom_client***

•Script associé à l'événement désactivation:

Désactiver ***Bt_valider_client***

Générer événement (désactivation) pour ***Bt_Enregistrement_commande***

En Powerbuilder.

•Script associé à la pression du bouton Bt_valider_nom_client:

```
cb_valider_nom_client.enabled=false
Valider_nom_client(nom_cli,prenom_cli,num_cli,adresse_cli,erreur)
IF client_valide.genere="true" THEN
    sle_num_cli.text=num_cli
    mle_adresse_cli.text=adresse_cli
    triggerevent(cb_enregistrement_commande,"activation")
ELSE
    messagebox("Message d'erreur",erreur_client,Exclamation!)
END IF
```

•Script associé à l'événement activation:

```
IF nom_cli<>"" and prenom_cli<>"" THEN
cb_valider_nom_client.enabled=true
END IF
```

•Script associé à l'événement désactivation:

```
cb_valider_nom_client.enabled=false
triggerevent(cb_enregistrement_commande,"desactivation")
```

3.3. Génération de l'UP principale de l'application

Nous avions posé l'hypothèse d'avoir un certain nombre d'UP pour les différentes phases de l'application accessibles via un menu principal. L'application générée comportera donc une UP principale qui sera constituée d'une seule fenêtre munie d'un certain nombre de menus déroulants permettant d'accéder, via leurs items, aux différentes fonctionnalités de l'application[26]. Afin de pouvoir structurer ces menus nous introduirons le concept de groupement de phases.

Avant de détailler la génération de la fenêtre principale, faisons quelques restrictions et proposons une fenêtre principale type. Concernant l'organisation des menus dans la fenêtre, nous nous baserons sur des règles ergonomiques d'interfaçage des applictions de gestion tant que ce sera possible [Van,92].

3.3.1. Restrictions

Etant donné que nous sommes dans un approche de prototypage, il ne sera pas ici, question de générer une fenêtre principale définitive, mais bien une fenêtre prototype permettant d'accéder à toutes les fonctionnalités de l'application. C'est pourquoi nous faisons dès maintenant quelques restrictions quant à l'organisation des menus dans la fenêtre.

Nous adopterons une structure de menus assez simple : une barre de menu constituant un premier niveau de découpe et des menus déroulants accessibles via cette barre de menu comme deuxième niveau. La barre de menu comprendra des items correspondant aux menus déroulants regroupant des fonctionnalités de l'application et un item "Quitter" qui permettra de sortir de l'application. Les premiers items permettront de dérouler les menus de la fenêtre principale et l'item "Quitter" ouvrira une boîte de dialogue demandant confirmation pour quitter l'application.

Le concept de groupement de phase nous permettra de structurer et construire les différents menus déroulants.

---

[26] C'est d'ailleurs une structure d'application très utilisée dans l'environnement Windows.

3.3.2. Groupement de phases

Un groupement de phases se définira comme un ensemble de phases de l'application qui ont des similitudes, des objectifs similaires. Une fois toutes les phases de l'application identifiées, le concepteur devra donc en spécifier des groupements. Ceux-ci correspondront à des menus déroulants dans la fenêtre principale dont les items ne son autres que les phases du groupement (on les appellera item-phase). Cela pemettrait ainsi d'avoir un certain nombre de menus déroulants plus ou moins cohérents.

Outre l'ensemble des phases du groupement, il faut y définir un ordre de priorité afin de déterminer l'ordonnacement des items-phase dans les menus. La phase ayant la plus grande priorité se retrouvera en haut du menu déroulant, celle ayant la deuxième plus grande priorité se trouvera juste en-dessous et ainsi de suite (de haut en bas pour un ordre de priorité décroissant).

Evidemment pour que les menus déroulants soient simples et compréhensibles, il faudra donner un intitulé à chaque groupement et à chaque phase afin de donner un nom approprié et clair à chaque menu déroulant et à chaque item de menu.

Si le concepteur constate par exemple, des similitudes entre les phases A, B et C, il les groupera dans les spécifications sous l'intitulé "Groupement A"; celui-ci deviendra le nom du menu déroulant. Ensuite, il donne un ordre de priorité pour ces 3 phases au sein du groupement (1°-PhaseA / 2°-PhaseB / 3°-PhaseC par exemple) pour pouvoir placer les items-phase dans le menu. Il donne enfin un intitulé à chacune des 3 phases ("PhaseA", "PhaseB" et "PhaseC" par exemple), chacun des intitulés deviendra le nom des items du menu déroulant "Groupement A"; ceux-ci seront placés selon l'ordre de priorité spécifié précédemment (Fig. 6.).

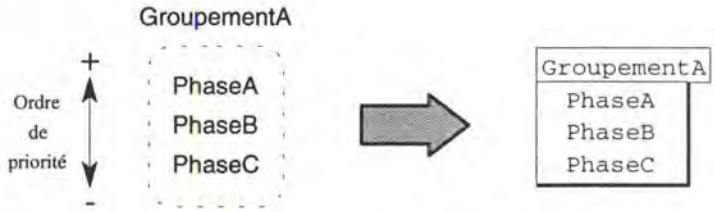

Figure 6.22- Un groupement de phases permet de définit un menu déroulant

Il faudra donc définir des groupements afin de couvrir toutes les phases de l'application, ainsi toutes les phases de l'application seront accessibles via le menu principal. Chaque menu déroulant se retrouvera dans la barre de menu sous forme d'un item, il faut donc encore définir l'ordre dans lequel ceux-ci seront placés. Ainsi, une fois que toutes les phases sont couvertes par un groupement, il faudra donner un ordre de priorité à chaque groupement afin de pouvoir les placer dans la barre de menu. Le placement commencera en fait après l'item "Quitter" qui se trouvera le plus à gauche dans la barre de menu. Le groupement avec la plus grande priorité se trouvera juste à droite de l'item "Quitter", le deuxième groupement se trouvera à droite de ce dernier et ainis de suite (de gauche à droite pour un ordre de priorité décroissant). De cette manière, on aura un barre de menu se composant de l'item "Quitter" et des différents

menus déroulants identifiés par les groupements (on les appellera les menus-groupement) selon l'ordre de priorité.

Si le concepteur a couvert toutes les phases de l'application en les groupant (Groupement A et B), il pourra définir un ordre de priorité des groupements qui permettra de définir un l'ordre de placement des items de la barre de menu (des 2 groupements, le groupement le plus prioritaire est le groupement A par exemple). Les items de la barre de menu seront donc, de gauche à droite : "Quitter", "GroupementA" et "GroupementB".

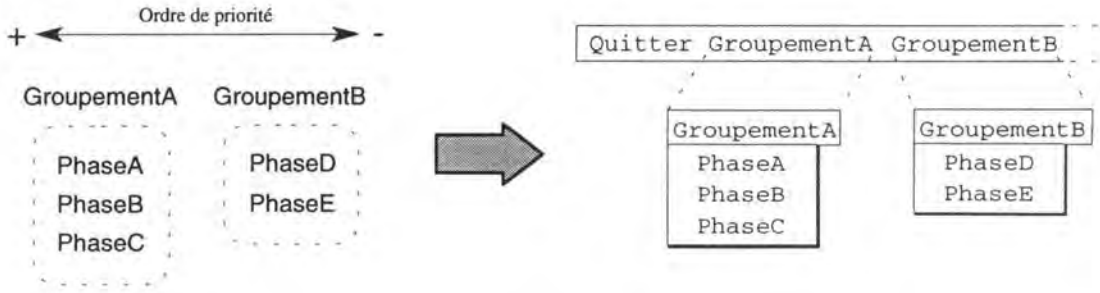

Figure 6.23 - L'ordre de priorité des groupements pemet de définir l'ordre des items de la barre de menu

Les spécifications concernant les groupements permettront donc de générer les menus de la fenêtre principale de l'application. L'idéal serait qu'à chaque item de la barre de menu et à chaque item-phase soit associé un texte d'aide d'une ligne qui s'afficherait dans le bas de la fenêtre lorsque l'item en question est en surbrillance et donnerait une indication sur ce que permet la sélection de cet item. Pour cela, il faudra encore compléter les spécifications en associant ce petit texte à chaque groupement et à chaque phase.

Ainsi, toutes ces informations qui stockées dans l'outil d'aide à la conception vont permettre de générer une fenêtre principale minimale opérationnelle, c'est ce que nous allons voir maintenant.

3.3.3. Génération de la fenêtre principale

La présentation de la fenêtre principale se composera de la fenêtre elle-même, de ses menus déroulants et de son espace d'aide en bas de celle-ci.

Le dialogue de cette fenêtre se composera de la navigation dans les menus, du déroulement des menus, de l'affichage de l'aide quand un item sera en surbrillance et de l'ouverture de la fenêtre correspondant à l'item de menu lorsque celui-ci sera sélectionné[27]. Il faudra de plus, que le nom de la fenêtre d'entrée d'une phase soit connu,

[27] Il faudra que le nom de la fenêtre d'entrée correspondant à une phase soit connu pour toutes les phases afin de pouvoir l'ouvrir lorsque l'item est sélectionné; c'est le cas lorsque la génération la présentation de l'UP de chaque phase est réalisée. Il faudra donc que la génération

et ceci pour toutes les phases. C'est seulement à ce moment-là que la génération de la fenêtre principale sera possible.

Nous verrons tout d'abord le type de fenêtre que nous générerons, ensuite nous analyserons la génération de la fenêtre principale selon ces 2 composants (présentation et dialogue)pour rester cohérent avec l'architecture de l'application cliente. En réalité, la génération de la fenêtre donnera un seul fichier de description de la fenêtre et mêlera la présentation avec le dialogue. Nous expliquerons donc ce qui doit être généré pour la présentation et ce qui doit être généré pour le dialogue, ensuite nous donnerons un algorithme abstrait de génération assez simpliste qui ne tient compte d'aucun outil en particulier, ni d'un quelconque fichier résultat[28].

A. Fenêtre principale de type MDI

Il est possible de générer plusieurs types de fenêtre, parmi lesquels la fenêtre MDI. Une fenêtre MDI (Multiple-Document Interface) est une fenêtre qui agit en tant que fond d'une application, c'est-à-dire qu'aucune fenête fille ne peut en dépasser les limites. La fenêtre principale que nous générerons sera de type MDI, ainsi toutes ses fenêtres filles seront toujours à l'intérieur de son cadre. Cela permet de ne pas mélanger les fenêtres de l'application avec les applications éventuellement ouvertes dans l'environnement fenêtré[29]. Ce type de fenêtre permet de plus l'afichage d'une 'micro-aide' en réservant un champ spécial d'aide dans le bas de la fenêtre sut toute le largeur de celle-ci.

B. Génération de la présentation de la fenêtre principale

B.1. Barre de menu et menus déroulants de la fenêtre principale

Les outils 4GL sont conçus pour faciliter la tâche du développeur et en ce qui concerne la barre de menu et les menus déroulants, L'outil 4GL se charge du placement exact des items de menu, et dimensionne automatiquement les menus déroulants.

En ce qui concerne la barre de menu, il suffira donc de déclarer dans leur ordre les items de la barre de menu. L'outil 4GL se chargera de placer les items l'un à côté de l'autre (de gauche à droite) de telle manière qu'ils soient à égale distance.

En ce qui concerne les menus déroulants, il suffira de déclarer dans leur ordre les items de menu. L'outil 4GL se chargera de les placer l'un en-dessous de telle manière à ce qu'ils soient à égale distance et de telle manière que la largeur du menu déroulant soit un peu plus large que l'item de menu le plus large et la longueur permette de voir tous les items de menu.

---

[28] Nous donnerons un algorithme abstrait plus ou moins détaillé de génération de la fenêtre principale dans le cadre de Powerbuilder puisque pour générer un texte de définition d'une fenêtre, il faudra tenir compte d'un format précis de définition.

[29] Un environnement de type Windows est, rappelons-le, un environnement multi-fenêtré; de ce fait, quand plusieurs applications sont ouvertes, il n'est pas toujours facile de deviner quelle fenêtre appartient à quelle application.

B.2. Déclaration des menus

Il suffira donc de déclarer les items de la barre de menu ainsi que les items des menus déroulants dans l'ordre spécifié (l'ordre de priorité). La déclaration des différents éléments d'un menu se font de la manière suivante. On déclare le premier item de la barre de menu (le plus à gauche), ensuite le menus déroulable à partir de cet item, ensuite les menus déroulables à partir des items de ce dernier menu et ainsi de suite, on fait de même pour le deuxième item de la barre de menu, pour le troisième et ainis de suite. L'ordre de déclaration correspond à l'ordre de priorité : on déclare d'abord l'item le plus prioritaire (celui à gauche dans la barre de menu ou celui en haut dans un menu déroulant), ensuite le second (celui juste à droite dans la barre de menu ou celui juste en-dessous dans le menu déroulant) et ainsi de suite.

En fait, la structure des menus de la fenêtre principale de l'application peut être vue comme un arbre à 2 niveaux. ayant pour racine la fenêtre elle-même, pour noeud de premier niveau les items de la barre de menu et pour noeud de second niveau les items des menus déroulants (nous n'allons pas plus loin qu'un niveau de menus déroulants - V. restrictions). La déclaration reviendrait donc à une "déclaration du menu en profondeur d'abord" commencant au premier niveau de noeud, c'est-à-dire aux items de la barre de menu.

B.3. Génération de la présentation

La génération du menu de la fenêtre principale consistera donc à déclarer et à décrire le type de la fenêtre (MDI), de même pour la barre de menu et les menus déroulants de cette fenêtre en suivant le schéma de déclaration que nous venons de voir. L'objet fenêtre et les différents objets qui la composent sont dans un premier temps tous déclarés et ensuite, chacun d'entre eux est décrit de façon précise (initialisation des paramètres, c'est-à-dire descritpion des propriétés de ces objets quand ils sont créés à l'ouverture de la fenêtre).

On déclare tout d'abord l'objet fenêtre et on déclare les objets qu'elle contient comme suit : la déclaration de la barre de menu commence avec la déclaration de l'item "Quitter" tout à gauche de la barre de menu; elle continue avec la déclaration des autres items qui correspondent aux menus déroulants et pour chacun de ces items, il faudra déclarer les items de menus présents dans le menu déroulant correspondant.

Quand la déclaration des objets de la fenêtre est faite, il faut en décrire précisément les propriétés. En ce qui concerne la fenêtre, sa taille sera celle de l'écran, c'est-à-dire que la fenêtre sera dans l'état "maximisé" (Maximized). Elle comprendra de plus, un champ spécial d'aide dans le bas de la fenêtre. Quant aux menus, les propriétés seront celles oar défaut dans l'outil.

Voici maintenant la démarche à suivre pour définir la présentation de la fenêtre principale :

```
// Déclaration de l'objet fenêtre et des objets qu'elle contient:
Déclaration de la fenêtre;
Déclaration de l'item de barre menu "Quitter";
```

Pour tous les groupements de phases dans l'ordre décroissant de priorité :

Déclaration de l'item de barre de menu correspondant au groupement (nom de l'item = intitulé du groupement);

Pour toutes les phases du groupement dans l'ordre décroissant de priorité :

Déclaration de l'item de menu correspondant à la phase (nom de l'item = intitulé de la phase);

// **Description** de l'objet fenêtre et des objets qu'elle contient:

Description de la fenêtre;

Description de l'item de barre menu "Quitter";

Pour tous les groupements de phases dans l'ordre décroissant de priorité :

Description de l'item de barre de menu correspondant au groupement (nom de l'item = intitulé du groupement);

Pour toutes les phases du groupement dans l'ordre décroissant de priorité :

Description de l'item de menu correspondant à la phase (nom de l'item = intitulé de la phase);

Maintenant que les différents objets sont déclarés et décrits, on peut aborder le dialogue, c'est-à-dire tous les appels de fenêtre ou de boîte de diaogue (pour l'item "Quitter").

C. Génération du dialogue de la fenêtre principale

Le dialogue de la fenêtre principale consistera en la navigation dans les menus (c'est-à-dire le déroulement des menus lorsqu'ils sont sélectionnés, mettre l'item de menu en surbrillance et afficher l'aide correspondante lorsque celui-ci a le 'focus') et à ouvrir les fenêtres correspondant aux items de menu lorsque ceux-ci sont sélectionnés.

Le gestionnaire d'interface des outils 4GL s'occupe de toute la navigation dans les menus, ce qui soulage grandement la gestion du dialogue de la fenêtre principale. En fait, une fois les menus les menus déclarés et décrits, il ne reste plus qu'à rattacher la 'micro-aide' à tous les items (ceux de la barre de menu et des menus déroulants) et à rattacher un script aux différents items de menu afin de pouvoir ouvrir les fenêtres correspondant à la phase sélectionnée.

Donc, par rapport à la présentation, il suffit d'associer les scripts et la 'micro-aide' aux différents items. Concernant la 'micro-aide', on attachera un petit texte à chaque item et le gestionnaire d'interface se chargera de l'afficher dans le champ d'affichage prévu à cet effet (dans le bas de la fenêtre), dès que cet item a le 'focus'. Concernant les scripts, il suffit de mettre simplement une instruction d'ouverture de la fenêtre correspondant à la phase sélectionnée.

On verra que dans le cas de Powerbuilder, la génération de la présentation et du dialogue sont beaucoup plus liées. En réalité, le rattachement de l'aide et du script se fera lors de la descritpion de chaque item (V. chapitre 9).

D. Génération de la fenêtre principale

Maintenant, nous pouvons donner un algorithme abstrait de génération de la fenêtre principale.

// Déclaration de l'objet fenêtre et des objets qu'elle contient:

Déclaration de la fenêtre;

Déclaration de l'item de barre menu "Quitter";

Pour tous les groupements de phases dans l'ordre décroissant de priorité :

Déclaration de l'item de barre de menu correspondant au groupement (nom de l'item = intitulé du groupement);

Pour toutes les phases du groupement dans l'ordre décroissant de priorité :

Déclaration de l'item de menu correspondant à la phase (nom de l'item = intitulé de la phase);

// Description de l'objet fenêtre et des objets qu'elle contient:

Description de la fenêtre;

Description de l'item de barre menu "Quitter" (+ associer le texte d'aide);

Associer le script d'appel à la boîte de dialogue de confirmation;

Pour tous les groupements de phases dans l'ordre décroissant de priorité :

Description de l'item de barre de menu correspondant au groupement (nom de l'item = intitulé du groupement) (+ associer le texte d'aide);

Pour toutes les phases du groupement dans l'ordre décroissant de priorité :

Description de l'item de menu correspondant à la phase (nom de l'item = intitulé de la phase) (+ associer le texte d'aide);

Associer le script d'ouverture de la fenêtre correspondante;

Il faudra en plus, déclarer cette fenêtre comme étant la fenêtre d'ouverture de l'application.. Nous verrons dans le chapitre 9, qu'il faudra d'abord déclarer et donner les caractéristiques de l'application, et mettre une instruction d'ouverture de la fenêtre principale.

7

Définition du langage DSL-II

Jusqu'à présent nous avons présenté les modèles utilisés pour la génération des différents composants de l'application et expliqué comment ceux-ci étaient exploités.

Dans cette partie, nous allons définir formellement ces modèles en introduisant un langage de spécification. Ces modèles et le langage associé devront être instancié dans l'outil générique Paquita de manière à réaliser l'outil d'aide à la conception.

Nous n'allons pas créer un nouveau langage de spécification, mais définir un sous-ensemble d'un langage existant, le langage DSL (Dynamic Specification Language), que nous modifierons et étendrons selon nos besoins. Le langage résultant de ces modifications et extensions sera appeller le langage DSL-II.

1. Le langage DSL

Le langage DSL (Dynamic Specification Language) est un langage de spécification non procédural basé sur le modèle entité-association et "partitionable" [DSL,89]. C'est le langage utilisé dans l'atelier logiciel IDA[1] [Bod-Pig,89].

1.1. Langage de spécification

C'est un langage de description des spécifications d'un sytème d'information. C'est un langage qui a été principalement conçu comme un outil de représentation des spécifications fonctionnelles, autrement dit comme un outil d'expression des spécifications basées sur les modèles de l'analyse fonctionnelle de la méthode IDA.

[1] L'atelier logiciel IDA est un outil d'aide à la conception permettant de décrire ses spécifications sur base des modèles de la méthode IDA à l'aide du langage DSL et de les y stocker [Bod-Pig,89].

1.2. Langage non-procédural basé sur le modèle entité-association

Le langage DSL est basé sur le modèle de structuration des données entité-association, il peut donc être exprimé grâce à un schéma entité-association. Il est en fait construit à partir d'objets (ou entités) possédant un type, de types de relation (ou associations), et de propriétés (ou attributs) associés à ces objets ou relations.

C'est un langage non-procédural dans la mesure où il pemet la description dans un ordre quelconque et d'une manière progressive du système cible. Ceci vient du fait que le langage DSL est construit à partir d'une expression entité-association. Le langage DSL peut être considéré comme une formulation syntaxique appropriée pour décrire des entités et leurs associations correspondant aux concepts des modèles de spécification [Bod-Pig,89].

1.3. Langage "partitionable"

Le langage est "partitionable" dans la mesure où l'ensemble des objets, des types de relation et des propriétés peut être structuré en sous-ensembles relativement autonomes permettant de décrire les différents aspects d'un système d'information. Il est ansi possible de définir un sous-ensemble pour la structuration des données, un autre pour la statique des traitements, ...

1.4. Langage textuel et graphique

Le langage DSL s'exprime de 2 manières différentes: textuellement et graphiquement. La forme d'expression textuelle permet de d'exprimer les spécifications sous forme de phrases structurées grâce à des mots-clés et rédigées dans une langue naturelle. La forme d'expression graphique permet elle, d'exprimer les mêmes spécifications à l'aide de la représentation graphique[2] des différents concepts des modèles. Ces 2 formes sont utilisables conjointement dans l'atelier logiciel IDA d'aide à la conception.

Ces 2 formes d'expression sont toutes les deux importantes. Le forme graphique présente l'avantage d'être très visuelle, et donc facile à utiliser. La forme textuelle quant à elle, autorise plus de souplesse dans la rédaction. Tous les aspects de spécification qui ne peuvent être représentés graphiquement pourront être complétés grâce au texte.

A noter que pour chaque modèle de spécification, il y a un certain nombre de règles à respecter pour que les spécifications soient complètes et cohérentes : les règles de

[2] La représentation graphique du modèle entité-assocation que nous avions présentée dans le chapitre consacré aux données de l'application, et celle du GEF dans le chapitre consacré à l'application cliente.

complétude et de cohérence [Bod-Pig,89]. Sans nous étendre la-dessus, nous pouvons dire que ces règles peuvent être réalisées à l'aide de contraintes d'intégrité associées au schéma entité-association qui représente le langage DSL. Si l'on formalise ces contraintes d'intégrité, il est alors possible de faire des vérifications automatiques dans l'atelier logiciel IDA, ce qui n'est pas négligeable.

2. DSL-II : sous-ensemble du langage DSL étendu

Dans notre environnement, nous ne prenons que quelques modèles de la méthode IDA. Autement dit, nous ne partirons que d'un sous-ensemble du langage DSL. De plus, ces modèles ont été quelque peu aménagés, c'est pourquoi le sous-ensemble de départ devra être étendu. Nous apporterons quelques modifications à DSL pour pouvoir décrire les spécifications telles que nous le voulons. Pour la structuration des données (entité-association), il y a peu de changement. Pour la description des traitements, il y a quelques changements notables tels que la différenciation entre les messages externes, internes et d'erreur. Pour le graphe d'enchaînement de fonctions d'une phase, il n'existe pas dans le langage DSL et il nous faudra donc définir une syntaxe pour pouvoir le décrire. Quant au reste des spécifications (schéma relationnel et ses correspondances avec le schéma entité-association, l'interface utilisateur), elles sont essentiellement définies par les générateurs; il faudra aussi définir de nouveaux objets et une syntaxe.

Nous définirons donc un sous-ensemble du langage DSL étendu : le langage DSL-II. La suite immédiate consistera à présenter la syntaxe utilisée, les différents objets, leurs relations et leurs propriétés pour les aspects qui nous intéressent. Nous n'indiquerons pas les modifications par rapport au langage DSL[3], nous présenterons simplement les objets du langage DSL-II.

2.1. Syntaxe du langage

La syntaxe du langage DSL-II sera la même que celle de DSL[4]. Nous avions vu que son expression textuelle permettait d'exprimer des phrases structurées à l'aide de mots-clés; Examinons maintenant les 5 types de phrases du langage:

- Une phrase d'en-tête de section permet de spécifier le type et le nom d'un objet du langage (une section consiste en une ou plusieurs phrases du langage);
- Une phrase de relation permet de spécifier des relations entre plusieurs objets du langage;

[3] Le langage DSL est complètement défini dans le manuel de référence [DSL,89].

[4] Pour plus de détails, consulter [DSL,89].

- Une phrase de propriété permet de spécifier une relation entre un objet et un ensemble de caractères;
- Une phrase d'entrée de texte libre permet d'associer un texte descriptif à un objet, il est ainsi possible de représenter une information en langage naturel qui ne pouvait être exprimée par un autre type de phrase;
- Une phrase de commentaire permet de commenter une suite de phrases.

Une phrase du langage est constituée de mots, de noms, de nombres et de signes de ponctuation. Le langage est un langage de format libre, c'est à dire que les phrases d'une section peuvent apparaître dans un ordre quelconque. La syntaxe de la description du langage est la suivante:

- Les mots en majuscule sont des mots réservés;
- Le mots en minuscule sont des noms dont les valeurs sont choisies par l'utilisateur;
- les mots entourés de crochets [] sont des clauses optionnelles à l'intérieur d'une phrase;
- Les mots entourés de parenthèses () sont des clauses répétitives;
- les mots entourés d'accolades { } et disposés en colonne ne peuvent être utilisés simultanément dans la phrase, il faut utiliser un des mots entourés.

Lors de la spécification en langage DSL-II, il faudra veiller à ce que toute phrase commence par un mot réservé et se termine par un point-virgule, à ce qu'il y ait au moins un blanc pour séparer les mots et les noms. Une phrase de commentaire doit commencer avec comme premier caractère un #. Lorsque l'on donne une liste de noms dans une phrase, ceux-ci doivent être séparé par une virgule.

2.2. Propriétés et relations des différents objets du langage

Les différents aspects que pourront couvrir les objets du langage sont les aspects "généralités", "structuration des données", "description des traitements", "graphe d'enchaînement des fonctions", "relationnel", "fonctions de l'application" et "interface utilisateur". Certains aspects seront définis manuellement, c'est à dire par le concepteur lui-même, alors que d'autres seront définis automatiquement par les générateurs. Nous considérons chaque aspect, excepté l'aspect "généralités", comme un modèle de spécification, certains correspondant à une spécification et les autres à une génération.

L'aspect "généralités" recouvre les propriétés communes à tous les objets, en l'occurrence des propriétés de généralité.

L'aspect "structuration des données" recouvre la spécifica-tion des données sous forme d'un schéma entité-association; il correspond au modèle entité-association. L'aspect "description des traitements" recouvre la description de l'application[5], de ses

[5] Lorsque nous avons abordé la description des fonctions de l'application, nous avions considérer un traitement comme étant une phase ou une fonction. Nous considérerons aussi un traitement de "niveau application" afin de pouvoir définir l'application et de regrouper tous ses traitements sous son nom.

phases et de ses fonctions; il correspond aux modèles de structuration et de la statique des traitements. L'aspect "graphe d'enchaînement des fonctions" recouvre la définition des conditions d'enchaînement et déclenchement des fonctions au sein d'une phase. Ces 3 aspects sont des aspects en rapport avec une spécification manuelle.

L'aspect "fonctions de l'application" recouvre la définition des paramètres des fonctions de l'application et des correspondances avec les messages. L'aspect "relationnel" recouvre la définition de la base de données relationnelle et des correspondances entre ses différents éléments et les concepts du schéma entité-association; il correspond au schéma relationnel standard généré sur base du schéma entité-association et à ses correpondances avec celui-ci. L'aspect "interface utilisateur" recouvre la définition des fenêtres de l'application (fenêtre principale et UP d'une phase) ainsi que celle des micro-dialogues identifiés pour chaque phase[6]; il correspond au travail de génération de l'IU. Seule la définition des groupements de phases[7] est réalisée manuellement, tout le reste est défini automatiquement. Ces 2 aspects sont en rapport avec le travail de génération.

2.2.1. Les objets du langage et les concepts qu'ils représentent

Avant de passer en revue tous les objets du langage DSL-II, leurs propriétés et leurs relations, nous allons présenter succinctement les objets propres à chaque aspect (nous établirons des correspondances entre certains objets du langage et les concepts qu'ils sont censés représenter quand cela s'avèrera possible).

A. Aspect "structuration des données"

Les objets utilisés pour la définition des spécifications entité-association sont les suivants : Agrégat, Association, Contrainte d'intégrité, Elément, Entité, Groupe et Rôle.

Nous allons présenter les correspondances entre les concepts du modèle entité-association de structuration des données et les objets du langage (Table 7.1). Pour définir un des concepts du modèle entité-association, il faudra utiliser l'objet du langage correspondant et en spécifier les différentes propriétés et relations.

Concepts de la structuration des données	Objets du langage
type d'entité	objet ENTITE
type d'association	objet ASSOCIATION
rôle joué par un type d'entité dans un type d'association	objet ROLE
attribut élémentaire	objet ELEMENT

[6] Le générateur de l'interface générera les spécifications de cet aspect et les utilisera pour générer l'IU.

[7] Groupement de phases qui donneront lieu aux menus déroulants.

attribut décomposable (groupe d'attirbuts)	objet GROUPE
groupe d'attributs élémentaires et/ou d'attributs décomposables et/ou de rôles	objet AGREGAT
contrainte d'intégrité informelle[8]	objet CONTRAINTE-D'INTEGRITE

Table 7.1 - Correspondances entre les concepts du modèle entité-association et les objets du langage

Seuls les concepts de base du modèle entité-association sont présentés sous forme d'objet, les contraintes d'intégrité telles que la connectivité, l'identification, les dépendances fonctionnelles, l'inclusion, l'exclusion et l'égalité se retrouveront sous forme de relations entre les objets sur lesquels elles sont définies.

B. Aspect "description des traitements"

Les objets utilisés pour la description des traitements sont les suivants : Traitement, Message-externe, Message-interne et Message-erreur.

Nous allons présenter les correpondances entre les concepts nécessaires à la description des traitements et les objets du langage (Table 7.2). Comme précédemment, pour définir un des concepts de ce modèle, il faudra utiliser l'objet du langage correspondant et en spécifier les différentes propriétés et relations.

Concepts de la descritpion des traitements	Objets du langage
traitement	objet TRAITEMENT
message externe en entrée ou en sortie d'un traitement	objet MESSAGE-EXTERNE
message interne en entrée ou en sortie d'un traitement	objet MESSAGE-INTERNE
message d'erreur en sortie d'un traitement	objet MESSAGE-ERREUR

Table 7.2 - Correspondances entre les concepts de la description des traite-ments et les objets du langage

C. Aspect "graphe d'enchaînement des fonctions"

Les objets concernés dans cet aspect seront le mêmes que ceux de la Table 7.2, on y ajoutera cependant de nouvel objet: nous aurons besoin d'un objet POINT-

[8] Les contraintes d'intégrité informelles sont des contraintes qui ne seront pas reprises structurellement dans l'expression du schéma entité-association. Rappelons que ces contraintes ne sont pas prises en compte lors de la génération de la base de données.

D'ACCUMULATION afin de pouvoir accumuler des messages pour un traitement. Il y aura quelques relations à rajouter pour les objets TRAITEMENT pour définir les connections de messages en entrée et en sortie de chaque traitement[9].

D. Aspect "relationnel"

Les différents objets utilisés pour stocker la définition du schéma relationnel sont les objets TABLE, COLONNE et INDEX. Les correspondances avec le schéma entité-association seront stockées sous forme de relation entre ces objets et ceux de l'aspect "structuration des données"[10].

E. Aspect "fonctions de l'application"

Les objets utilisés pour stocker la définition des paramètres des fonctions de l'application sont les objets PARAMETRE et CHAMP-PARAMETRE. Les relations avec les messages d'une fonction seront stockées sous forme de relation.

F. Aspect "interface utilisateur"

Les objets utilisés pour stocker la définition de l'interface utilisateur recouvre la définition des OI qui composent l'UP d'une phase, celle des micro-dialogues identifiés dans le dialogue correspondant à l'UP d'une phase et celle des groupements correspondant aux différents menus déroulants de la fenêtre principale. Les objets seront donc UNITE-DE-PRESENTATION, OBJET-INTERACTIF-ABSTRAIT, MICRO-DIALOGUE et GROUPEMENT-DE-PHASES. Ces objets auront des propriétés propres à l'interfaçage et des relations avec les différents objets des aspects auxquels ils sont rattachés.

G. Remarque

Nous avons présenté les objets propres à chaque aspect comme si chaque aspect était indépendant des autres aspects. En réalité, ces objets sont beaucoup plus reliés qu'il n'y paraît; ils peuvent intervenir dans plusieurs aspects comme nous le verrons[11].

2.2.2. Propriétés communes à tous les objets

Les propriétés communes aux différents objets permettent de définir des généralités à propos de chaque objet. Nous prenons simplement un sous-ensemble de ces propriétés du langage DSL. Parmi ces généralités, une description, des synonymes du nom d'un objet et un mémo.

---

[9] Uniquement pour les traitement de type fonction évidemment, puisqu'il s'agit de définir l'enchaînement et les conditions de déclenchement des fonctions d'une phase interactive de l'application.

[10] Les correspondances entre concept relationnel et entité-association sont présentées Table 6.1.

[11] Un message (interne ou externe) est un concept de la desritpion des traitements mais il est lié à la structuration des données par le fait que ce message contient des informations en rapport avec les données de l'application; les objets de la descritpion des traitements interviendront dans la définition du graphe d'enchaînement des fonctions; ... .

Voici la syntaxe de ces propriétés de généralité :

```
GENERALITES
SYNONYME (synonyme : , ) ;
MEMO (nom-de-memo : , ) ;
DESCRIPTION ;
    <texte> ;
```

La propriété DESCRIPTION permettra de rentrer un texte de description du concept que représente l'objet. Il est ainsi possible de décrire clairement l'objet en langage naturel en complément de ses relations et autres propriétés.

La propriété MEMO permettra de mettre des commentaires à propos d'un objet; cela peut être intéressant dans la mesure où plus il y a de commentaires, plus la documentation des spécifications sera complète[12]. Tout objet sera défini par un nom qui l'identifiera et pour en faciliter la recherche dans toutes les spécifications, il peut être utiles d'en définir des synonymes, c'est ce que permet la propriété SYNONYME. Ces deux propriétés ne sont là que pour aider le concepteur dans sa tâche de spécification[13].

2.2.3. Propriétés et relations propres à chaque objet

Nous présentons dans ce qui suit, la syntaxe des propriétés et relations des objets pour les aspects propres à la modélisation; ils sont présentés par ordre alphabétique.

L'objet **AGREGAT** permettra de définir des regroupements d'éléments et/ou de groupes et/ou de rôles (aspect structuration des données).

```
DEFINIR AGREGAT ( NOM : , ) ;
STRUCTURATION DES DONNEES
INCLUT {nom-élément I nom-groupe I nom-rôle} [ DE {nom-association I nom-entité} ] ;
IDENTIFIE {nom-association I nom-entité} ;
DETERMINE FONCTIONNELLEMENT {nom-agrégat I nom-élément I nom-groupe I nom-rôle} POUR {nom-association I
    nom-entité} ;
EST FONCTIONNELLEMENT DEPENDANT DE {nom-agrégat I nom-élément I nom-groupe I nom-rôle} POUR {nom-
    association I nom-entité} ;
```

L'objet **ASSOCIATION** permettra de définir un lien sémantique entre les entités, autrement dit un type d'association (aspect structuration des données). Il sera aussi manipulé par un ou plusieurs processus, il faut donc le mettre en rapport avec ce ou ces processus (aspect description des traitements).

```
DEFINIR ASSOCIATION ( NOM : , ) ;
```

---

[12] Cela permet par exemple de se justifier dans ses spécifications.

[13] Cela devient intéressant lorsque les spécifications deviennent très volumineuses.

```
STRUCTURATION DES DONNEES
RELIE nom-entité [ EN TANT QUE nom-rôle ] [ AVEC CONNECTIVITE <chaîne-de-caractère> ] ;
COMPREND ( [ <entier> FOIS ] {nom-élément | nom-groupe} : , ) ;
EST IDENTIFIE PAR {nom-agrégat | nom-élément | nom-groupe | nom-rôle} ;
A SON CONTENU COMPRIS DANS ( nom-message-interne : , ) ;
POUR AJOUT CONTRAINT PAR nom-contrainte-d-intégrité [ VERIFIE PAR nom-traitement ] ;
POUR MODIFICATION [ DE {nom-élément | nom-groupe} ] CONTRAINT PAR nom-contrainte-d-intégrité [ VERIFIE PAR
    nom-traitement ] ;
POUR SUPPRESSION CONTRAINT PAR nom-contrainte-d-intégrité [ VERIFIE PAR nom-traitement ] ;
DESCRIPTION DES TRAITEMENTS
EST REFERENCE [ <entier> FOIS ] PAR nom-traitement [ POUR {nom-élément | nom-groupe} ] ;
EST AJOUTE [ <entier> FOIS ] PAR nom-traitement ;
EST MODIFIE [ <entier> FOIS ] PAR nom-traitement [ POUR {nom-élément | nom-groupe} ] ;
EST SUPPRIME [ <entier> FOIS ] PAR nom-traitement ;
RELATIONNEL
CORRESPOND A LA FOREIGN KEY ( nom-colonne : , ) DANS nom-table ;
CORRESPOND A LA TABLE nom-table ;
    // ces 2 relations sont exclusives
```

L'objet **COLONNE** permettra de définir une colonne d'une table relationnelle de la base de données de l'application (aspect relationnel).

```
DEFINIR COLONNE ( NOM : , ) ;
RELATIONNEL
FAIT PARTIE DE LA TABLE nom-table ;
EST INCLUS DANS LA PRIMARY KEY DE nom-table ;
EST INCLUS DANS LA FOREIGN KEY DE nom-table REFERENCANT nom-table ;
EST INCLUS DANS L'INDEX nom-index ;
CORRESPOND A {nom-table | nom-élément} ;
```

L'objet **CONDITION** permettra de définir un état en fonction duquel on fait un choix quant aux message à générer (aspect description des traitements).

```
DEFINIR CONDITION ( NOM : , ) ;
DESCRIPTION DES TRAITEMENTS
ETAT <texte> ;
```

L'objet **CONTRAINTE-D'INTEGRITE** permettra de définir des contraintes informelles, non reprises structurellement dans l'expression DSL du schéma E/A (aspect structuration des données).

```
DEFINIR CONTRAINTE-D'INTEGRITE ( NOM : , ) ;
STRUCTURATION DES DONNEES
PREDICAT <texte> ;
VERIFIE ( [ PAR nom-traitement ] POUR AJOUTER {nom-association | nom-entité}: , );
```

VERIFIE ([PAR nom-traitement] POUR MODIFIER [{nom-élément | nom-groupe} DANS] {nom-association | nom-entité} : ,) ;

VERIFIE ([PAR nom-traitement] POUR SUPPRIMER {nom-association | nom-entité} : ,) ;

COUVRE (nom-rôle : ,) ;

L'objet **ELEMENT** permettra de définir des unités d'information "non-subdivisibles", c'est à dire des attributs non-décomposables (aspect structuration des données). Il sera aussi manipulé par un ou plusieurs processus, il faut donc le mettre en rapport avec ce ou ces processus (aspect description des traitements).

DEFINIR ELEMENT (NOM : ,) ;

STRUCTURATION DES DONNEES

TYPE {booléen | chaîne-de-caractère | date | entier | heure | réel} ;

FORMAT <entier> ;

STATUT {OBLIGATOIRE | FACULTATIF} ;

EST {SIMPLE | REPETITIF} ;

DOMAINE DE VALEUR (<toute valeur> [JUSQUE <toute valeur>] : ,) [EXTENSIBLE] ;

EST COMPRIS ([<entier> FOIS] DANS {nom-association | nom-entité | nom-groupe | nom-message-interne | nom-message-externe} : ,) ;

IDENTIFIE {nom-association | nom-entité} ;

DETERMINE FONCTIONNELLEMENT {nom-agrégat | nom-élément | nom-groupe | nom-rôle} POUR {nom-association | nom-entité} ;

EST FONCTIONNELLEMENT DEPENDANT DE {nom-agrégat | nom-élément | nom-groupe | nom-rôle} POUR {nom-association | nom-entité} ;

EST INCLUS [DEPUIS {nom-association | nom-entité}] DANS nom-agrégat ;

POUR MODIFICATION DANS {nom-association | nom-entité} CONTRAINT PAR nom-contrainte-d-intégrité [VERIFIE PAR nom-traitement] ;

DESCRIPTION DES TRAITEMENTS

EST REFERENCE DANS [<entier>] {nom-association | nom-entité} PAR nom-traitement ;

EST MODIFIE DANS [<entier>] {nom-association | nom-entité} PAR nom-traitement ;

RELATIONNEL

CORRESPOND A LA COLONNE nom-colonne DANS nom-table ;

CORRESPOND A LA TABLE nom-table ;

// ces 2 relations sont exclusives

INTERFACE UTILISATEUR

LIBELLE <chaîne-de-caractère> ;

L'objet **ENTITE** permettra de définir un type d'objet appartenant au réel perçu, autrement dit un type d'entité (aspect structuration des données). Il sera aussi manipulé par un ou plusieurs processus, il faut donc le mettre en rapport avec ce ou ces processus (aspect description des données).

DEFINIR ENTITE (NOM : ,) ;

STRUCTURATION DES DONNEES

EST RELIE PAR nom-association [EN TANT QUE nom-rôle] [AVEC CONNECTIVITE <chaîne-de-caractère>] ;

COMPREND ([<entier> FOIS] {nom-élément | nom-groupe} : ,) ;

EST IDENTIFIE PAR {nom-agrégat | nom-élément | nom-groupe | nom-rôle} ;

A SON CONTENU COMPRIS DANS (nom-message-interne : ,) ;

POUR AJOUT CONTRAINT PAR nom-contrainte-d-intégrité [VERIFIE PAR nom-traitement] ;

POUR MODIFICATION [DE {nom-élément | nom-groupe}] CONTRAINT PAR nom-contrainte-d-intégrité [VERIFIE PAR nom-traitement] ;

POUR SUPPRESSION CONTRAINT PAR nom-contrainte-d-intégrité [VERIFIE PAR nom-traitement] ;

DESCRIPTION DES TRAITEMENTS

EST REFERENCE [<entier> FOIS] PAR nom-traitement [POUR {nom-élément | nom-groupe}] ;

EST AJOUTE PAR [<entier> FOIS] PAR nom-traitement ;

EST MODIFIE PAR [<entier> FOIS] PAR nom-traitement [POUR {nom-élément | nom-groupe}] ;

EST SUPPRIME PAR [<entier> FOIS] PAR nom-traitement ;

RELATIONNEL

CORRESPOND A LA TABLE nom-table ;

L'objet **GROUPE** permettra de définir une structure de données formée d'éléments et de groupes (aspect structuration des données). Il sera aussi manipulé par un ou plusieurs processus, il faut donc le mettre en rapport avec ce ou ces processus (aspect description des traitements).

DEFINIR GROUPE (NOM : ,) ;

STRUCTURATION DES DONNEES

COMPREND ([<entier> FOIS] {nom-élément | nom-groupe} : ,) ;

EST COMPRIS DANS ([<entier> FOIS] DANS {nom-association | nom-entité | nom-groupe | nom-message-interne} : ,) ;

IDENTIFIE {nom-association | nom-entité} ;

DETERMINE FONCTIONNELLEMENT {nom-agrégat | nom-élément | nom-groupe | nom-rôle} POUR {nom-association | nom-entité} ;

EST FONCTIONNELLEMENT DEPENDANT DE {nom-agrégat | nom-élément | nom-groupe | nom-rôle} POUR {nom-association | nom-entité} ;

EST INCLUS [DEPUIS {nom-association | nom-entité}] DANS nom-agrégat ;

POUR MODIFICATION DANS {nom-association | nom-entité} CONTRAINT PAR nom-contrainte-d-intégrité [VERIFIE PAR nom-traitement] ;

DESCRIPTION DES TRAITEMENTS

EST REFERENCE DANS [<entier>] {nom-association | nom-entité} PAR nom-traitement ;

EST MODIFIE DANS [<entier>] {nom-association | nom-entité} PAR nom-traitement ;

L'objet **GROUPEMENT-DE-PHASES** permettra de regrouper des phase qui ont des similitudes (aspect interface utilisateur).

DEFINIR GROUPEMENT-DE-PHASES (NOM : ,) ;

INTERFACE UTILISATEUR

PRIORITE <entier> ;

REGROUPE LES PHASES (nom-traitement AVEC PRIORITE D'ITEM <entier> : ,) ;

L'objet **INDEX** permettra de définir un index sur une table relationnelle de la base de données de l'application (aspect relationnel).

```
DEFINIR INDEX ( NOM : , ) ;
------------------------------ RELATIONNEL ------------------------------
IINDEXE LA TABLE  nom-table ;
NCLUT  ( nom-colonne : , ) ;
```

L'objet **MEMO** permet de mettre des commentaires à propos d'objets. Il fait uniquement partie de l'aspect généralités.

```
DEFINIR MEMO ( NOM : , ) ;
```

L'objet **MESSAGE-ERREUR** permettra de représenter la communication d'un message d'erreur vers l'extérieur, autrement via l'interface (aspect description des traitements).

```
DEFINIR MESSAGE-ERREUR ( NOM : , ) ;
---------------------- DESCRIPTION DES TRAITEMENTS ----------------------
EST GENERE  PAR  nom-traitement  VERS INTERFACE  [ SI PAS nom-condition ] ;
------------------------ FONCTIONS DE L'APPLICATION ----------------------
CORRESPOND AU PARAMETRE  nom-paramètre ;
```

L'objet **MESSAGE-EXTERNE** permettra de représenter les communications externes d'information, c'est à dire via l'interface (aspect description des traitements). Ce sont des messages élémentaires, et ils comprennent des informations en relation avec le schéma entité-association, il faut donc les "relier" aux éléments de ce schéma (aspect structuration des données).

```
DEFINIR MESSAGE-EXTERNE ( NOM : , ) ;
---------------------- STRUCTURATION DES DONNEES ------------------------
COMPREND  (  nom-élément : , ) ;
---------------------- DESCRIPTION DES TRAITEMENTS ----------------------
EST GENERE  PAR  nom-traitement  VERS INTERFACE  [ SI nom-condition ] ;
EST RECU  PAR  nom-traitement  DEPUIS L'INTERFACE ;
------------------------ FONCTIONS DE L'APPLICATION ----------------------
CORRESPOND AU PARAMETRE  nom-paramètre ;
------------------------ INTERFACE UTILISATEUR ---------------------------
EST REPRESENTE PAR  nom-objet-interactif  DANS  nom-unité-de-présentation ;
```

L'objet **MESSAGE-INTERNE** permettra de représenter les communications internes d'information (aspect description des traitements). Ces messages, quant ils sont simples, comprennent des informations du schéma entité-association, il faut donc les "relier" à ce schéma (aspect structuration des données).

```
DEFINIR MESSAGE-INTERNE ( NOM : , ) ;
---------------------- STRUCTURATION DES DONNEES ------------------------
COMPREND  (nom-élément I nom-groupe) ;
```

COMPREND LE CONTENU DE ({nom-association I nom-entité} : ,) ;

DESCRIPTION DES TRAITEMENTS

SOUS-ENSEMBLE EST (nom-message-interne : ,) ;

EST SOUS-ENSEMBLE DE (nom-message-interne : ,) ;

EST GENERE PAR nom-traitement [SI nom-condition] ;

EST RECU PAR nom-traitement [<entier> FOIS [AU PLUS]] ;

FONCTIONS DE L'APPLICATION

CORRESPOND AU PARAMETRE nom-paramètre ;

L'objet **MICRO-DIALOGUE** permettra de définir un micro-dialogue du dialogue de l'UP d'une phase de l'application (aspect interface utilisateur), ou plus précisément le contenu sémantique d'un script.

DEFINIR MICRO-DIALOGUE (NOM : ,) ;

INTERFACE UTILISATEUR

TYPE {SAISIEIDECLENCHEMENTIIMPLICITE} ;

EST DECLENCHE PAR {nom-message-externelnom-fonction} ;

ENCAPSULE (nom-fonction : ,) ;

L'objet **OBJET-INTERACTIF-ABSTRAIT** permettra de définir un OI se trouvant dans l'UP d'une phase de l'application (aspect interface utilisateur).

DEFINIR OBJET-INTERACTIF (NOM : ,) ;

INTERFACE UTILISATEUR

USAGE {AFFICHAGE I DECLENCHEMENT I MIXTE I SAISIE } ;

TYPE {BOUTON I CHAMP-EDITION-SIMPLE I CLOE I CLONE I ... I CEX} ;

REPRESENTE {nom-message-externe I nom-traitement} DANS nom-unité-de-présentation ;

L'objet **PARAMETRE** permettra de définir le format du paramètre d'une fonction de l'application correspondant à un des messages de cette fonction (aspect fonctions de l'application).

DEFINIR PARAMETRE (NOM : ,) ;

FONCTIONS DE L'APPLICATION

FORMAT {SIMPLE I STRUCTURE} ;

DE TYPE {booléen I chaîne-de-caractère I date I entier I heure I réel} ;

COMPREND CHAMP (nom-champ-paramètre : ,) ;

// ces 2 relations sont exclusives

CORRESPOND AU MESSAGE {nom-message-erreur I nom-message-externe I nom-message-interne} ;

L'objet **CHAMP-PARAMETRE** permettra de définir les champs d'un paramètre de type structure (aspect fonctions de l'application).

DEFINIR CHAMP-PARAMETRE (NOM : ,) ;

FONCTIONS DE L'APPLICATION

DE TYPE {booléen | chaîne-de-caractère | date | entier | heure | réel} ;

EST UN CHAMP DU PARAMETRE nom-paramètre ;

L'objet **POINT-D'ACCUMULATION** permettra d'accumuler un certain nombre de messages internes et les fournir au traitement qui en a besoin sous forme d'un message interne agrégé (aspect graphe d'enchaînement des fonctions).

DEFINIR POINT-D'ACCUMULATION (NOM : ,) ;

GRAPHE D'ENCHAINEMENT DES FONCTIONS

ACCUMULE [[AU PLUS] <entier>] nom-message-interne ;

L'objet **ROLE** permettra de définir le rôle que joue une entité dans une association (aspect structuration des données).

DEFINIR ROLE (NOM : ,) ;

STRUCTURATION DES DONNEES

EST ASSUME PAR nom-entité DANS nom-association [AVEC CONNECTIVITE <chaîne-de-caractère>] ;

IDENTIFIE {nom-association | nom-entité} ;

DETERMINE FONCTIONNELLEMENT {nom-agrégat | nom-élément | nom-groupe | nom-rôle} POUR {nom-association | nom-entité} ;

EST FONCTIONNELLEMENT DEPENDANT DE {nom-agrégat | nom-élément | nom-groupe | nom-rôle} POUR {nom-entité | nom-association} ;

EST INCLUS [DEPUIS {nom-association | nom-entité}] DANS nom-agrégat ;

EST COUVERT PAR nom-contrainte-d-intégrité ;

L'objet **TABLE** permettra de définir une table relationnelle de la base de données de l'application (aspect relationnel).

DEFINIR TABLE (NOM : ,) ;

RELATIONNEL

POSSEDE COMME COLONNE (nom-colonne : ,) ;

A POUR PRIMARY KEY (nom-colonne : ,) ;

A POUR FOREIGN KEY (nom-colonne : ,) REFERENCANT nom-table ;

EST REFERENCEE PAR LA FOREIGN KEY (nom-colonne : ,) DANS nom-table ;

EST INDEXEE PAR nom-index ;

CORRESPOND A {nom-entité | nom-élément | nom-association} ;

FOREIGN KEY (nom-colonne : ,) CORRESPOND A nom-association ;

L'objet **TRAITEMENT** permettra de définir un ensemble d'opérations de traitement de l'information obéissant à un certain nombre de règles, autrement dit un traitement (aspect description des traitements). Il peut aussi vérifier un certain nombre de contraintes d'intégrité sur les données (aspect structuration des données). Et dans le graphe d'enchaînement des fonctions, il faut établir son occurrence, son type de déclenchement et les liens qui lient les messages en entrée et en sortie.

DEFINIR TRAITEMENT (NOM : ,) ;

STRUCTURATION DES DONNEES

VERIFIE nom-contrainte-d-intégrité POUR AJOUTER {nom-association | nom-entité} ;

VERIFIE nom-contrainte-d-intégrité POUR MODIFIER [{nom-élément | nom-groupe} DANS] {nom-association | nom-entité} ;

VERIFIE nom-contrainte-d-intégrité POUR SUPPRIMER {nom-association | nom-entité} ;

DESCRIPTION DES TRAITEMENTS

NIVEAU {APPLICATION | PHASE | FONCTION} ;

COMPOSÉ DE (nom-traitement : ,) ;

EST COMPOSANT DE nom-traitement ;

RECOIT (nom-message-interne [[AU PLUS] <entier> FOIS] : ,) ;

RECOIT (nom-message-externe : ,) DEPUIS L'INTERFACE ;

GENERE nom-message-interne [SI nom-condition] ;

GENERE (nom-message-externe : ,) VERS INTERFACE [SI nom-condition] ;

GENERE nom-message-erreur VERS INTERFACE [SI PAS nom-condition] ;

REFERENCE [{nom-élément | nom-groupe} DANS] [<entier>] {nom-association | nom-entité} ;

AJOUTE [<entier>] {nom-association | nom-entité} ;

MODIFIE [{nom-élément | nom-groupe} DANS] [<entier>] {nom-association | nom-entité} ;

SUPPRIME [<entier>] {nom-association | nom-entité} ;

PROCEDURE ;
<texte> ;

GRAPHE D'ENCHAINEMENT DES FONCTIONS

RECOIT {nom-message-interne | nom-message-externe} [(ET {nom-message-interne | nom-message-externe} : ,)] ;

GENERE nom-message-erreur XOU nom-message-interne [ET (nom-message-externe : ,)] ;

OCCURENCE {SIMPLE | MULTIPLE} ;

DECLENCHEMENT {EXPLICITE | IMPLICITE} ;

FONCTIONS DE L'APPLICATION

FORMAT D'APPEL (nom-paramètre : ,) ;

INTERFACE UTILISATEUR

EST INCLUS DANS GROUPEMENT DE PHASES nom-groupement-de-phases AVEC PRIORITE D'ITEM <entier> ;

EST REPRESENTE PAR nom-unité-de-présentation ;

EST REPRESENTE PAR nom-objet-interactif DANS nom-unité-de-présentation ;

// ces 2 relations sont exclusives

L'objet **UNITE-DE-PRESENTATION** permettra de définir une unité de présentation correspondant à une phase de l'application (aspect interface utilisateur).

DEFINIR UNITE-DE-PRESENTATION (NOM : ,) ;

INTERFACE UTILISATEUR

REPRESENTE nom-traitement ;

CONTIENT nom-objet-interactif REPRESENTANT {nom-traitement | nom-message-externe} ;

3. Instanciation de Paquita pour réaliser l'outil d'aide à la conception

Maintenant que le langage DSL-II a été défini, il faudrait un outil d'aide à la conception permettant de rentrer ses spécifications et de les y stocker. Il sera possible de réaliser cet outil grâce à l'outil Paquita de la société Metsi.

Paquita est un outil générique, c'est à dire un outil multi-méthodes. Il peut aussi bien supporter des méthodes classiques telles que IDA, que des méhodes spécifiques. Pour que l'outil supporte la méthode que l'on désire utiliser, et plus exactement les modèles de la méthode que l'on désire utiliser, il faut en faire une instanciation. Elle se fait après qu'une méta-modélisation dans l'instanciateur de Paquita ait été réalisée, c'est à dire une définition dans l'outil des modèles utilisés, plus précisément le langage de spécification des modèles utilisés (en l'occurrence DSL-II). L'instanciation nous permet ainsi de définir notre propre outil de modélisation[14].

3.1. Méta-modélisation

Le travail de modélisation recouvre 4 aspects : la définition de la sémantique des modèles de la méthode à instancier, l'expression textuelle et graphique de cette sémantique et la définition des profils utilisateurs [Paq,93]. Seuls les 3 premiers aspects nous intéressent particulièrement.

3.2. L'instanciateur de Paquita

L'instanciateur se compose d'un certain nombre d'éditeurs textuels et d'un éditeur graphique orienté objet (OOG). Ces éditeurs nous permettront de définir les concepts des différents modèles. L'éditeur graphique permettra de définir le méta-schéma[15] entité-association directement, graphiquement; le éditeurs textuels permettront de compléter la définition (notamment de définir les mots-clés du langage).

Une fois la méta-modélisation réalisée, il faudra faire l'instanciation de Paquita et ainsi avoir à sa disposition un outil d'aide à la conception (Fig. 7.1).

---

[14] Tous les concepts définis dans cet outil seront stockés dans son repository.

[15] Le méta-schéma ou schéma de définition d'un modèle de spécification, il modélise les différents objets du langage et leur interrelations.

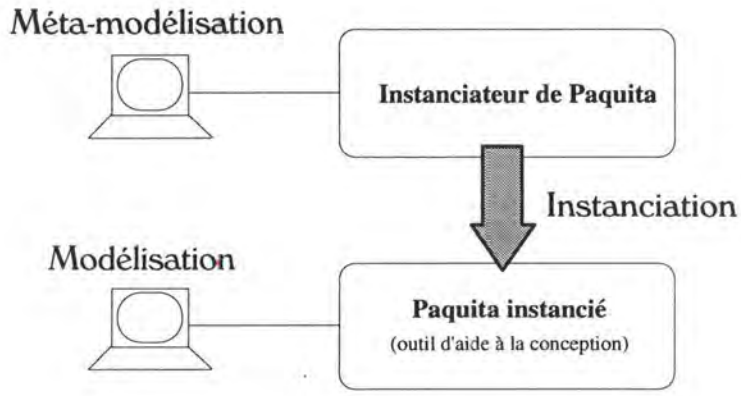

Figure 7.1 - Méta-modélisation et instanciation

L'instanciateur nous pemettra donc de réaliser le travail de méta-modélisation. Il faudra dans un premier temps spécifier le méta-schéma entité-association de définition de DSL-II et associer l'expression textuelle du langage aux différents concepts[16]. Ceci permet d'instancier l'outil Paquita pour l'expression textuelle de DSL-II. Il faudra ensuite associer la représentation graphique des différents concepts aux objets qui viennent d'être défini afin de pouvoir instancier l'outil Paquita de telle manière qu'il permette de rentrer les spécifications soit graphiquement à l'aide de l'éditeur graphique, soit textuellement à l'aide du langage DSL-II lui-même. Ainsi, le concepteur pourra utiliser conjointement le langage graphique et textuel pour rentrer ses spécifications[17]. Une fois la méta-modélisation terminée, on peut faire l'instanciation et réaliser ainsi un outil instancié supportant la spécification des modèles définis.

L'outil instancié sera notre outil d'aide à la conception tel que nous le voulons, c'est à dire avec les modèles dont nous aurons besoin pour faire une génération. Il sera ainsi possible au concepteur de spécifier et de stocker son schéma entité-association, la description des traitements et les graphes d'enchaînement de fonctions d'une phase, il permettra aussi de stocker le schéma relationnel, les paramètres des fonctions et la spécification de l'interface utilisateur qui auront été générés. Toutes les informations nécessaires à la génération des différentes parties de l'application seront ainsi disponibles. L'extraction des spécifications pourra se faire à l'aide de procédures d'extraction qui seront utilisées par les générateurs.

---

[16] L'expression textuelle consiste en des mots-clés permettant d'exprimer des phrases structurées de spécification telles que celles définies ci-dessus.

[17] Un des avantages du repository central d'un outil tel que Paquita est de faciliter l'utilisation conjointe des 2 formes d'expression. En effet, que les spécifications soient rentrées graphiquement ou textuellement, elles sont stockées dans le repository de la même façon. En fait, on représente les mêmes choses mais de manière différente. Si une modification est effectuée graphiquement, elle se répercute forcément dans la forme textuelle et vice-versa.

8

Format des fichiers Powerbuilder à générer

Jusqu'ici, nous avons présenté les différentes générations à réaliser afin d'obtenir un prototype à compléter dans l'outil 4GL[1] en faisant abstraction d'un quelconque outil 4GL. Envisageons maintenant le cas particulier de l'outil Powerbuilder de Powersoft et présentons le format précis des fichiers à générer.

En plus des différents éléments de l'applications (OI, scripts, fonctions, variables, ...) précédemment évoqués, il s'agira dans Powerbuilder de définir un certain nombre de paramètres et un objet application pour lequel notamment un script d'entrée dans l'application sera défini. Ce script comprendra entre autre l'instruction d'ouverture de la fenêtre principale de l'application.

Tous les composants de l'application sont stockés dans une librairie qui sera accessible par un module assurant la gestion du répertoire contenant l'ensemble des librairies. Ce module possède un certain nombre de fonctions de gestion du répertoire dont des fonctions d'exportation et d'importation de fichiers. Ces fonctions de manipulation du répertoire pourront également être exploitée en dehors de l'environnement Powerbuilder grâce à une API[2] créée afin de pouvoir faire coexister Powerbuilder avec le monde des outils CASE. Parmi cette API, on trouve une fonction de création de librairie, une fonction d'importation et de compilation d'un élément de l'application. La génération dans le cadre de Powerbuilder utilisera les fonctions de cette API. pour compiler et stocker les fichiers de définition dans le répertoire de l'application générée.

Définissons maintenant le format Powerbuilder de ces fichiers, définissant les différents composants de l'application, que les générateurs devront respecter. Dans la définition de la syntaxe que nous utilisons, les termes en gras correspondent à des mots réservés ou des mots qui ne changeront pas d'une application à l'autre. Les autres mots

---

[1] Rappelons qu'il faut programmer le corps des fonctions de l'application et le prototype devrait fonctionner.

[2] Application Program Interface.

(caractères standards) sont les parties de fichiers qu'il faudra remplir à partir des informations extraites des modèles conceptuels.

1. Génération du fichier application

Le fichier application est le celui qui définit l'objet représentant l'application à développer. C'est le point de départ de la construction de l'application, on pourra ensuite définir tous les autres éléments de l'application. Voici le format de ce fichier:

En-tête du fichier avec le nom que portera l'application; Déclaration d'objets globaux nécessaires au fonctionnement de l'application (sqlca, sqlda, sqlsa, error et message);	**$PBExportHeader$**<nom-application>**.sra** **forward** **global transaction sqlca** **global dynamicdescriptionarea sqlda** **global dynamicstagingarea sqlsa** **global error error** **global message message** **end forward**
Déclaration de l'objet application;	**global type** <nom-application> **from application** **end type** **global** <nom-application> <nom-application>

Instructions exécutées lors de la création de l'objet application (dès que l'on lance l'application, on créer l'objet et par voie de conséquence, on exécute ces instructions);	**on** <nom-application>**.create**
	  appname = "<nom-application>**"**
	  sqlca = create transaction
	  sqlda = create dynamicdescriptionarea
	  sqlsa = create dynamicstagingarea
	  error = create error
	  message = create message
	end on

Instructions exécutées lors de la destruction de l'objet application (destruction des objets globaux sqlca, sqlda, sqlsa, error et message);	**on** <nom-application>**.destroy**
	  destroy(sqlca)
	  destroy(sqlda)
	  destroy(sqlsa)
	  destroy(error)
	  destroy(message)
	end on

Script d'entrée dans l'application, une fois l'application lancée, ces instructions seront exécutées. Dans notre cas, il n'y aura qu'une instruction d'ouverture de la fenêtre principale (la définition de cette fenêtre se trouve plus loin dans le chapitre).	**on open;** **Open(**<nom-fenetre-principale>**)** **end on**

Lors de la génération du prototype de l'application, il faudra définir le nom de l'application qui remplacera <nom-application> dans le fichier. Quand au script d'ouverture de la fenêtre principale, il faut attendre que celle-ci soit créée pour y mettre l'instruction d'ouverture de la fenêtre.

2. Génération de la signature des fonctions de l'application

La génération des fonctions consistera tout d'abord à générer le fichier de définition des différents types de données structurées et à générer ensuite le fichier de définition des différentes fonctions.

4.1. Génération des types de données structurées

Dans les fonctions, les messages internes correspondront à des paramètres de type 'données structurées'. Ces types seront déclarés dans Powerbuilder grâce à une Structure Powerbuilder. Voici le format d'un fichier de définition d'une structure powerbuilder:

En-tête du fichier avec le nom que portera la structure;	**$PBExportHeader$**<nom-structure>**.srs**
Déclaration de l'objet structure et de ses champs et leur type (les champs seront déclarés l'un en-dessous de l'autre).	**global type** <nom-structure> **from structure** type-de-variable <nom-champ> **end type**

type-de-variable = {boolean | date | integer | real |
string | structure | time }

Lors de la génération du fichier de définition de la structure, <nom-structure>, <type-de-variable> et <nom-champ> seront définis à partie des informations extraites sur le type des paramètres.

4.2. Génération des fonctions de l'application

Pour chaque fonction, il faudra générer un fichier de définition d'une fonction globale Voici le format d'un fichier de définition d'une fonction Powerbuilder:

En-tête du fichier avec le nom que portera la fonction;	**$PBExportHeader$**<nom-fonction>**.srf**
Déclaration de l'objet fonction;	**global type** <nom-fonction> **from function_object** **end type**
Déclaration de la fonction et de ses paramètres et leur type;	**forward prototypes** **global subroutine** <nom-fonction> (type-de-paramètre <nom-paramètre>,) **end prototypes**
Description du corps de la fonction.	**global subroutine** <nom-fonction> (type-de-paramètre <nom-paramètre>,); <corps-fonction / texte> **end subroutine**

type-de-paramètre = [ref] {boolean | date | integer | real |
string | structure | time }

3. Génération de l'interface utilisateur de l'application

La génération de l'interface de l'application consiste à générer les fenêtres correspondant aux différentes phases de l'application et la fenêtre principale de l'application.

3.1. Génération de la fenêtre d'une phase

Lorsque nous avions présenter la génération de l'UP d'une phase, nous avions traité séparément la génération de l'état du système, de sa gestion, de la présentation et du dialogue. Or ici, le résultat de la génération ne sera qu'un seul fichier dans lequel seront intégrés ces différents aspects. Voici le format d'un fichier de définition d'une fenêtre Powerbuilder que le générateur dvra respecter:

En-tête du fichier avec le nom que portera la fenêtre;	**$PBExportHeader$**<nom-fenêtre>**.srw** **forward**
Déclaration de l'objet fenêtre	**global type** <nom-fenêtre> **from Window** **end type**
Déclaration des objets interactifs (OI)de la fenêtre;	**type** <nom-OI> **from** Liste-OI **within** <nom-fenêtre> **end type** **end forward**
Déclaration des variables de la fenêtre;	**shared variables** Liste-type-de-variable <nom-varaible> **end variables**
Description de l'objet fenêtre (ce sera une fenêtre fille de la fenêtre principale);	**global type** <nom-fenêtre> **from Window** **int X=**<entier> **int Y=**<entier> **int Width=**<entier> **int Height=**<entier> **boolean TitleBar=**{true I false} **string Title=**<intitulé-fenêtre> **boolean ControlMenu=**{true I false} **boolean MinBox=**{true I false} **boolean MaxBox=**{true I false} **boolean Resizable=**{true I false} **WindowType WindowType=**child!
Liste des OI de la fenêtre;	<nom-OI> <nom-OI> **end type** **global** <nom-fenêtre> <nom-fenêtre>
Déclaration des fonctions locales à la fenêtre;	**Forward prototypes** **public subroutine** <nom-fonction> (type-de-variable ... ... <nom-paramètre>,) **end prototypes**
Instructions de création des différents OI de la fenêtre exécutées lors de la création de l'objet fenêtre (dès que l'on ouvre la fenêtre, l'objet est créé et par voie de conséquence, on exécute ces instructions);	**on** <nom-fenêtre>**.create** **this.**<nom-OI>**=create** <nom-OI> **this.Control[]={ this.**<nom-OI>(,**&** **this.**<nom-OI>) } **end on**

Instructions de desctruction des OI de la fenêtre lors de la desctruction de l'OF;

```
on <nom-fenêtre>.destroy
destroy(<nom-OI>)
end on
```

Définition des fonctions locales;

```
public subroutine <nom-fonction> (Liste-type-de-variable ...
                                                    ... <nom-paramètre>,
);
<corps-de-la-fonction / texte>
end subroutine
```

Description des OI de la fenêtre (la première partie des attributs de l'OI concernant sa position et sa dimension est commune à tous les objets; la deuxième partie dépendra du type des OI);

- Attributs propres à un bouton de commande
- Attributs propres à une liste-combo
- Attributs propres à une liste de sélection
- Attributs propres à un champ d'édition multiple
- Attributs propres à un bouton radio
- Attributs propres à un champ d'édition simple
- Attributs propres à un texte libellé

```
type <nom-OI> from Liste-OI within <nom-fenêtre>
int X=<entier>
int Y=<entier>
int Width=<entier>
int Height=<entier>
{ ⟨si commandbutton⟩
    int TabOrder=<entier>
    string Text=<string> |
  ⟨si dropdownlistbox⟩
    int TabOrder=<entier>
    [String Text=<string>]
    boolean VScrollBar={true | false}
    long BackColor=<entier-long>
    String Item[]={ <string>{,&
    <string>) } |
  ⟨si listbox⟩
    int TabOrder=<entier>
    boolean enabled={true | false}
    boolean VScrollBar={true | false}
    long BackColor=<entier-long>
    String Item[]={ <string>{,&
    <string>) } |
  ⟨si multilineedit⟩
    int TabOrder=<entier>
    boolean enabled={true | false}
    [String Text=<string>]
    long BackColor=<entier-long> |
  ⟨si radiobutton⟩
    string Text=<string>
    long BackColor=<entier-long>
  ⟨si singlelineedit⟩
    int TabOrder=<entier>
    boolean enabled={true | false}
    [String Text=<string>]
    boolean AutoHScroll={true | false}
    long BackColor=<entier-long> |
   ⟨si statictext⟩
    boolean Enabled={true | false}
    string Text=<string>
    Alignment Alignment={Center! | Left! | Right!}
    long BackColor=<entier-long> }
end type
```

Définition du script associé à un événement sur l'OI;

```
on <évènement-sur-OI>;
<script / texte>
end on
```

Comme la fenêtre sera une fenêtre fille de la fenêtre principale qui est de type MDI, il faut la déclarer comme une fenêtre cliente d'une fenêtre MDI.

```
type mdi_1 from mdiclient within <nom-fenêtre>
long BackColor=<couleur>
end type
```

---

type-de-variable = {boolean | date | integer | real | string | structure | time }

Liste-OI = {commandbutton | checkbox | dropdownlistbox | listbox | multilineedit | radiobutton | statictext | singlelineedit}

3.2. Génération de la fenêtre principale de l'application

Lorsque nous avions présenter la génération de la fenêtre principale de l'application, nous avions parlé de la définition de la structure du menu principal. Dans l'outil Powerbuilder, il faut définir le menu à part et l'inclure dans la fenêtre le contenant. Il faudra donc générer un fichier de défintion du menu principal et ensuite un fichier de définition de la fenêtre principale incluant ce menu.

3.2.1. Génération du menu principal

Voici le format d'un fichier de définition d'un menu Powerbuilder:

En-tête du fichier avec le nom que portera le menu principal;	**$PBExportHeader$**<nom-menu-principal>**.srm** **forward**
Déclaration de l'objet menu;	**global type** <nom-menu-principal> **from menu** **end type**
Déclaration de l'item Quitter;	**type** item-quitter **from menu within** <nom-menu-principal> **end type**
Déclaration du menu déroulalnt associé au groupement e phases <groupement-i>;	**type** <groupement-i> **from menu within** <nom-menu-principal> **end type**
Déclaration des items (<item-i>) du menu déroulant <groupement-i>;	**type** <item-menu-j-groupement-i> **from menu within** <groupement-i> **end type**
Liste des items du menu déroulant;	**type** <groupement-i> **from menu within** <nom-menu-principal> <item-menu-j-groupement-i> <item-menu-j-groupement-i> **end type** **end forward**
Description de l'objet menu (déclaration de tous les items de la barre de menu);	**global type** <nom-menu-principal> **from menu** item-quitter item-quitter <groupement-i> <groupement-i> **end type** **global** <nom-menu-principal> <nom-menu-principal>
Instructions de création des différents items de la barre de menu exécutées lors de la création de l'objet menu (création de l'item Quitter et des menus déroulants);	**on** <nom-menu-principal>**.create** <nom-menu-principal>**=this** **this.**item-quitter**=create** item-quitter **this.**<groupement-i>**=create** <groupement-i> **this.item[]={this.**item-quitter**, &** **(this.**<groupement-i>**, &) }** **end on**
Instructions de destruction des différents items de la barre de menu exécutées lors de la desctruction de l'objet menu;	**on** <nom-menu-principal>**.destroy** **destroy(this.**item-quitter) **destroy(this.**<groupement-i>**)** **end on**
Descritpion de l'item Quitter;	**type** item-quitter **from menu within** <nom-menu-principal> **end type**
Instructions exécutées lors de la création de l'item (définition de son texte et sa micro-aide);	**on** item-quitter**.create** **this.Text=**"Sortie" **this.MicroHelp=**"Sortie de l'application" **end on** **on** item-quitter**.destroy** **end on**

Descritpion du menu déroulant;	**type** <groupement-i> **from menu within** <nom-menu-principal> <item-menu-j-groupement-i> <item-menu-j-groupement-i> **end type**
Instructions exécutées lors de la création du menu déroulant (création des items de menu);	**on** <groupement-i>**.create** **this.Text=**<string> **this.Microhelp=**<string> **this.**<item-menu-j-groupement-i>**=create** <item-menu-j-groupement-i> **this.Item[]={this.**<item-menu-j-groupement-i>(, & **this.**<item-menu-j-groupement-i>) } **end on** **on** <groupement-i>**.destroy** **destroy(this.**<item-menu-j-groupement-i>**)** **end on**
Description des items de menu;	**type** <item-menu-j-groupement-i> **from menu within** <groupement-i> **end type**
Instruction exécutées lors de la création de ces items;	**on** <item-menu-j-groupement-i>**.create** **this.Text=**<string> **this.Microhelp=**<string> **end on** **on** <item-menu-j-groupement-i>**.destroy** **end on**
Instruction d'ouverture de la fenêtre <nom-fenêtre> correspondant à une phase en tant que fille de la fenêtre <nom-fenêtre-principale>.	**on clicked;** OpenSheet(<nom-fenetre>,<nom-fenetre-principale>,i,original!) **end on**

3.2.2. Génération de la fenêtre principale

Comme les fenêtres précédentes, le résultat de la génération de la fenêtre proprement dite ne sera qu'un seul fichier dans lequel seront intégrés les aspects présentation et dialogue. Voici le format du fichier de définition de la fenêtre principale:

En-tête du fichier avec le nom que portera la fenête principale;	**$PBExportHeader$**<nom-fenetre-principale>**.srw** **forward**
Déclaration de l'objet fenêtre;	**global type** <nom-fenetre-principale> **from Window** **end type**
Déclaration la zone cliente MDI (mdi_1) dans la fenêtre MDI[3];	**type** mdi_1 **from mdiclient within** <nom-fenetre-principale> **end type** **end forward**

[3] La zone cliente MDI est celle dans laquelle les fenêtres clientes de la fenêtre principale MDI vont s'ouvrir; comme la fenêtre possède un barre de menu et une 'micro-aide', cette zone est comprise entre la barre de menu, l'espace de 'micro-aide' et les bords horizontaux de la fenêtres.

Descritpion de la fenêtre principale;

```
global type <nom-fenetre-principale> from Window
int X=intger
int Y=integer
int Width=integer
int Height=integer
boolean TitleBar=true
string Title=<intitulé-fenêtre>
string MenuName="<nom-menu-principal>"
long BackColor=268435456
boolean ControlMenu=true
boolean MinBox=true
boolean MaxBox=true
boolean Resizable=true
WindowState WindowState=maximized!
WindowType WindowType=mdihelp!
mdi_1 mdi_1
end type
global <nom-fenetre-principale> <nom-fenetre-principale>
```

Instruction exécutées lors de la création de la fenêtre principale (création du menu et de la zone cliente MDI);

```
on <nom-fenetre-principale>.create
this.MenuID = create <nom-menu-principal>
this.mdi_1=create mdi_1
this.Control[]={ this.mdi_1}
end on
```

Instructions de destruction exécutées lors de la desctruction de la fenêtre principale;

```
on <nom-fenetre-principale>.destroy
destroy(MenuID)
destroy(mdi_1)
end on
```

Description de la zone cliente MDI.

```
type mdi_1 from mdiclient within <nom-fenetre-principale>
long BackColor=<couleur>
end type
```

Conclusion

L'enjeu de notre mémoire a été d'étudier la possibilité d'une génération automatique, dans les outils 4GL, des applications de gestion hautement interactives conçues suivant une architecture client/serveur. Cette génération s'intégrant dans un environnement d'aide au développement d'applications couplant un outil CASE et un 4GL.

Nous avons identifié les différents composants de ce type d'application et définit les modèles correspondant sur lesquels se basera le processus de génération qui aura pour résultat une base de données opérationnelle et un prototype de l'application cliente. Ce prototype comprend la signature des fonctions, la présentation et le dialogue de l'interface utilisateur de l'application.

Les données de l'application sont modélisées à l'aide du modèle entité-association. Nous avons tout d'abord expliqué comment transformer un schéma entité-association en un schéma relationnel standard. Ensuite.nous avons montré comment, sur base de ce dernier, le texte de définition des données de la base de données relationnelles pourra être dérivé et exécuté afin de créer les tables et leur index.

Les fonctions de l'application sont modélisées à l'aide du modèle de la statique des traitements. Pour chacune d'entre elles, une fonction exécutable vide comprenant sa description en commentaire est générée.

Nous avons défini une structure type d'interface qui se base sur la découpe en phases de l'application. La génération de l'interface utilisateur consiste à générer les différentes fenêtres correspondant aux phases de l'application ainsi que la fenêtre principale qui permettra d'accéder à celles-ci. Pour chacune d'entre elles, il faut générer la présentation et le dialogue. Concernant la fenêtre principale, la génération se basera sur le concept de groupement de phases que nous avons introduit. Concernant les autres fenêtres, le graphe d'enchaînement des fonctions d'une phase ainsi que les deux autres modèles précédents serviront de base à la génération. Des règles de sélection permettront de déduire les différents OI de chaque fenêtre sur base des informations contenues dans ces modèles. Le concept de Micro-dialogue que nous avons introduit permettra d'adapter le GEF au style de dialogue événementiel et servira de base à la génération des scripts.

Les bases de la génération automatique ayant été posées, un langage de spécification des différent modèles, DSL-II, a ensuite été défini. Nous avons également illustré pour un outil 4GL en particulier, Powerbuilder, le format des fichiers devant être générés et placés dans le répertoire de l'outil.

L'étendue du travail ne nous a pas permis de réaliser l'environnement de spécifications et les générateurs. Cette réalisation devra faire l'objet d'une prochaine étape. L'instanciation des modèles retenus dans l'outil générique Paquita permettra de réaliser l'outil d'aide à la conception permettant de spécifier l'application à l'aide du langage DSL-II. Des procédures d'extraction fournissant les informations nécessaires aux générateurs devront également être définies. Finalement, des générateurs, construisant des fichiers de définition suivant un format établi à l'avance(V. chapitre 8 Powerbuilder) en utilisant les informations extraites, devront être construits. Ces générateurs implémenteront les principes et la stratégie de génération que nous avons présentés tout au long du travail.

Bibliographie

[Bec,93] Beck A., *User participation in Systems Design - Result of a field study,* in CHI '93 - vol. A (ACM).

[Bod-Bea,88] Bodart F., Beat M., *La conception de base de données relationnelle,* in OUTPUT Nr.2, 1988.

[Bod-Pig,89] Bodart F., Pigneur Y., *Conception assistée des systèmes d'information*, Masson, 1989.

[Bod,94] Bodart F., Vanderdonckt J., Hennebert A.-M., Leheureux J.-M., Provot I., Vanderdonckt J., Zucchinetti G., *Dimensions clé pour une méthodologie de développement d'applications interactives,* 1994.

[Cou,91] Coutaz J., Bass L., *Developping software for the user interface, ...* ,1991.

[Dat,90] Date, C.J., *A contribution to the study of database integrity,* in Relational Database Writings 1985-1989, Addison-Wesley, 1990.

[Dat,90] Date, C.J., *An introduction to database systems - Vol. 1 - 5th Edition,* Addison-Wesley, 1990.

[DSL,89] *Manuel de référence DSL / Atelier logiciel IDA*, Metsi,1989.

[Fol,92] Foley J.D., Gieskens D.F., *Controlling user interface objects through pre- and postconditions*, in CHI '92 (ACM).

[Fol,92] Foley J.D., de Baar D. J.M.J., Mullet K.E., *Coupling application design and user interface design*, in CHI '92 (ACM).

[Hai,86] Hainaut J.L., *Conception assistée des applications informatiques - conception de base de données*, Masson, 1986.

[Hai,89] Hainaut J.L., *Base de données et base de connaissances en gestion des organisations,* 5ème Ecole d'Automne de Base de données, Port Barcarès (AFCET), Novembre 1989.

[Hai,92] Hainaut J.L., Cadelli M., Decuyper B., Marchand O., *Tramis : a transformation-based database CASE Tool,* in Proceedings of the 5th Conf. on Software Engineering and its Applications, Toulouse, Decembre 1992.

[Hai,91] Hainaut J.L., *Entity-generating schema transformation for entity-relationship models,* in 10th Conf. on E-R Approach, San Mateo (CA), October 1991.

[Hek,88] Hekmatpour S., Ince D., *Software prototyping, formal methods and VDM,* ..., 1988

[Her,93] Herczec J., Hohl H., Ressel M. (Research group DRUID), *A new approach to visual programming in user interface design*, in HCI '93 - vol. B (ACM).

[Jan,93] Janssen ., *iii,* , 1993.

[Joh,92] Johnson J., *Selectors: going beyond user interface widgets*, in CHI '92 (ACM).

[Kad,93] Kaddah M.M., *Interactive scenarios for the developpement of a user interface prototype*, in HCI '93 - vol. B (ACM).

[Lau,93] Lauesen S., Harning M.B., *Dialogue design through modified Dataflow and data modeling,* in HCI '93 - vol. B (ACM).

[Lin,94] Linthicum D., *4GLs: Productivity at what cost?*, DBMS, May 1994.

[Mei,91] Meinadier J.P., *L'interface utilisateur - pour une informatique plus conviviale*, Dunod, 1991.

[Mun,92] Muñoz R., Miller-Jacobs H.H., Spool J.M., Verplank B., *In search of the ideal prototype,* in CHI '92 (ACM).

[Paq,93] *Paquita : instanciateur - guide de méta-modélisation*, Metsi, 1993.

[Pet,87] Petoud I., *Applications de gestion hautement interactives en saisie : Vers un style d'interaction,* Université de Lausanne, 1987.

[Pet,90] Petoud I., Pigneur Y., *An automatic and visual approach for user interface design,* in Engineering for human-computer interaction, Cokton G. (editor), 1990.

[Sze,93] Szekely P., Luo P., Neches R., *Beyond Interface builders : model-based interface tools*, in INTERCHI '93 (ACM).

[Van,92] Vanderdonckt J., *Corpus ergonomique minimal des applications de gestion,* Rapport IHM/Règles/10-FUNDP, 1992

[Van,93] Vanderdonckt J., *A corpus of selection rules for choosing interaction objects,* Technical Report, Projet Trident-FUNDP, August 1993.