MASTER EN SCIENCES INFORMATIQUES

Application orientée objets: contribution à la modélisation et au prototypage

Bersez, François

Award date:
1993

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
Rue Grandgagnage, 21b
B-5000 NAMUR

Application orientée objets: contribution à la modélisation et au prototypage.

François Bersez

Promoteur : Jean-Luc Hainaut

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

Année académique 1992 - 1993

Résumé

Mots-clés :

Base de données - Orienté objets - Logiciel d'aide à la conception de base de données - Prototypage - Modélisation

Ce mémoire est consacré au développement d'une base de données orientée objets. Nous présentons donc les aspects du paradigme objet, les différentes méthodologies orientées objets qui peuvent être utilisées pour la conception d'applications orientées objets. Notre base de donnée orientée objets est construite à partir de l'atelier logiciel TRAMIS développé par les Facultés Universitaires Notre-Dame de la Paix de Namur. TRAMIS est un logiciel d'aide à la conception de bases de données. Une fois ce dernier présenté, nous expliquons comment nous pouvons spécifier, modéliser par prototypages successifs une base de données orientée objets sous TRAMIS et comment nous pouvons en générer le code exécutable C++.

Abstract

Keywords :

Database - Object-oriented - Database case tool - Prototyping - Modeling

This memoire is about the development an object-oriented database. Thus, we 'll expose the Object paradigm, the differents object oriented methodologies that can be used in the design of an object-oriented application. Our object-oriented database is built with the database tool case TRAMIS developped by the "Facultés Universitaires Notre-Dame de la Paix" of Namur (Belgium). After TRAMIS being presented, we'll expose how we give the specification of an object-oriented database using prototyping, and how we can generate the C++ code from it.

REMERCIEMENTS

Je tiens à remercier toutes les personnes ayant contribué, de près ou de loin à la réalisation de ce mémoire. Je tiens à remercier tout particulièrement :

Nick Hills, Damian Mehers et Robin Anderson de la société DEC, Ferney-Voltaire (FRANCE) qui, durant mon stage, m'ont aidé dans mon approche du paradigme objet.

L'équipe de développement de TRAMIS, Olivier Marchand, Jean Henrard et Jean-Marc Hick qui ont toujours été présents et sans qui le développement aurait été beaucoup plus pénible.

Jean-Luc Hainaut qui a toujours su faire preuve d'une grande disponibilité et d'une grande patience envers moi.

Marie Arnould qui m'a aidée dans la mise en forme de ce document.

Enfin, je remercie mes parents et mes amis qui ont toujours été à mes côtés, et qui m'ont toujours encouragé tout le long de ce mémoire et de mes études.

TABLE DES MATIERES

Introduction

Dans ce mémoire, il va être question de spécifier et de concevoir un système de gestion de base de données orientée objets. Concevoir un système de gestion de base de données n'est pas chose aisée.

Mais qu'est ce qu'une approche orientée objet de la programmation, quels sont les concepts de tels systèmes. Qu'entend-on exactement par "système de gestion de base de données orientée objets" par rapport à d'autres systèmes, relationnels ou autres ?

Après avoir répondu à toutes ces questions, nous allons tenter d'expliquer et de commenter toutes les étapes, les concepts et les outils nécessaires à la conception d'un système de gestion de bases de données orientées objets.

Nous aurons à faire des choix concernant la représentation des données du système orienté objets que nous allons implémenter. Devons nous nous inspirer d'une des nombreuses méthodologies d'analyse et de design orientées objets sur le marché ? Nous tenterons de dresser une typologie de ces différentes méthodologies.

Nous avons à notre disposition d'atelier logiciel TRAMIS. TRAMIS est logiciel d'aide à la conception de base de données. Nous aurons l'occasion de revenir sur une description de ce logiciel plus loin dans ce document.

Nous allons tenter, à partir d'une spécification d'une base de données orientée objets conceptuelle obtenue par prototypages successifs, sous TRAMIS, de générer le code exécutable d'une base de données orientée objets. Le code généré sera du C++.

Nous devrons faire des choix quant à la représentation physique de notre base de donnée. Elle devra être capable de représenter tous les concepts d'un système de gestion de base de données orienté objets. Nous devrons également réfléchir à la forme et au contenu du code exécutable généré. La forme du code est importante du pont de vue de la stabilité et de la cohérence de notre future base de données. Le contenu l'est tout autant puisqu'il devra fournir toutes les primitives d'accès et de manipulation des données, c'est à dire des objets, de la base de donnée.

Une fois le code généré, il devrait nous permettre de construire des applications complètes utilisant notre système de gestion de base de données orientée objets.

CHAPITRE I : Conception "Orientée Objets"

1.1 Objectifs de la conception "Orientée Objet" de la programmation

On peut considérer que l'objectif de la programmation objet est essentiellement d'améliorer la qualité des logiciels en programmant moins et mieux.

Pour **moins programmer**, il faut disposer de langages de plus haut niveau que les langages classiques, qui se rapprochent autant que possible des étapes d'analyse et de spécification des programmes. Il faut également que ces langages permettent d'éviter le re-développement complet des programmes existants lorsque seules quelques modifications y sont apportées.

Pour **programmer mieux**, il faut que ces langages offrent des structures permettant de garantir la validité, la fiabilité et la performance des logiciels développés.

Différents facteurs influent sur la programmation et permettant d'en diminuer le volume. En particulier :

- **La portabilité :** c'est la possibilité d'utiliser le même code pour réaliser une même application, mais sur des machines ou environnements différents. L'emploi de langages objets permet d'isoler les détails d'implémentation dans les niveaux les plus bas de spécification des objets.

- **La compatibilité :** de nouveaux programmes développés doivent pouvoir être intégrés au sein d'applications déjà existantes. Dans cette optique, de nombreux langages orientés objets ont choisi d'être associés à un précompilateur qui produit du code source dans un langage procédural classique. C++ produit du code source C. Il permet également l'intégration inverse de programmes C dans des programmes C++.

- **La validité :** Les programmes doivent assurer exactement les fonctions définies par le cahier des charges. Certains langages objets autorisent la

spécification explicite de postconditions, préconditions et axiomes qui sont des propriétés intrinsèques des données et des traitements associés[1].

- **La vérifiabilité :** Ce facteur est fortement lié au précédent. Il réside en la définition plus ou moins aisée des procédures de test et de recette des logiciels[2].

- **L'intégrité :** c'est la faculté d'un programme à protéger son code et ses données, d'accès non autorisés[3]. Toutefois, les langages orientés objets, par eux-mêmes, n'offrent pas vraiment encore de réponse à ce problème.

- **La fiabilité :** il s'agit d'assurer qu'en cas de défauts de fonctionnement d'un composant de l'application, les conséquences sur l'ensemble ne seront pas catastrophiques. Les trois facteurs précédents sont donc essentiels pour la satisfaction de cet objectif.

- **La maintenabilité :** Ici interviennent la lisibilité, la compréhension, la structuration des programmes. L'approche très modulaire des langages objets est une bonne réponse, avec toutefois, la réserve suivante : le compilateur doit être associé à un bon environnement de débogage[4].

- **La réutilisabilité :** Il s'agit de pouvoir réutiliser des développements réalisés pour une application dans d'autres contextes.

- **L'extensibilité :** c'est la faculté de pouvoir adjoindre des données ou des fonctionnalités supplémentaires sans corrompre l'intégrité du logiciel préexistant.

- **La facilité d'utilisation :** l'aspect structuré des langages orientés objets est un bon point, mais il n'est pas possible de dire que la syntaxe soit triviale, en particulier pour C++.

---

[1] C++ n'offre pas explicitement cette possibilité, c'est au programmeur d'insérer le code correspondant. La validité est à relier directement aux possibilités de vérification du programme, c'est-à-dire la facilité de produire pour chaque application des séquences de tests significatives du bon fonctionnement du programme.

[2] Peu de réponses satisfaisantes sont proposées aujourd'hui. La modularité du C++ semble toutefois un critère permettant une évaluation élément par élément, donc , une évaluation plus simple.

[3] De nombreux travaux apparaissent, en particulier en ce qui concerne les base de données orientées objet.

[4] De tels environnements commencent à apparaître pour C++.

- **L'efficacité :** c'est la possibilité d'exploiter au mieux les ressources offertes par la ou les machines où le logiciel sera implanté. Le C++ est en général considéré comme très efficace, mais, comme les autres langages objets ou non, il est loin d'offrir toutes les facilités d'exploitation de ressources matérielles souhaitables.

Pour compliquer le problème, certains de ces facteurs sont incompatibles entre eux, par exemple :

- **Intégrité - facilité d'utilisation :** pour être très intègre, il faut spécifier systématiquement toutes les conditions limites, toutes les propriétés invariantes, etc. ; pour être facile d'emploi, il ne faut pas avoir à écrire des pages de spécification pour une ligne de code "utile".

- **Efficacité - portabilité :** un programme très efficace prend en compte tous les avantages et les inconvénients de la machine cible ; un programme très portable se cantonne à tout formuler de manière très standard et générique, et bien souvent peu performante.

- **Efficacité - extensibilité :** c'est la même chose que ci-dessus mais, cette fois, l'efficacité prend en compte les caractéristiques spécifiques du cas traité alors que l'extensibilité voudrait rester à un stade très générique.

1.2 Les propriétés de la conception "Orientée Objet"

Les propriétés de l'orientation objet sont caractérisées par plusieurs clefs d'abstraction : [OO1]

1.2.1 L'encapsulation.

C'est le groupement d'un ensemble d'idées en une seule unité logique.

C'est la construction d'une partition qui rend des données privées pour un petit nombre de processus qui peuvent, eux, utiliser ces données directement, tandis que les autres devront le faire par une requête.

L'encapsulation est un mécanisme tel qu'un observateur externe[5] a connaissance de ce qu'un objet fait, mais ne connaît pas la façon dont il le fait, ni sa représentation interne. Elle permet de construire un objet qui ressemblera à une sorte de boîte noire pour un observateur externe.

---

[5] Nous considérons que les observateurs externes sont l'application et les objets qui la composent.

L'encapsulation

- Distingue la forme des données de leur contenu
- Isole le système des changements de la représentation physique des données
- Empêche des interventions frauduleuses ou non autorisées sur les objets

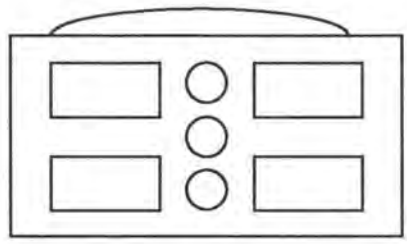

Figure 1.1 : l'objet, ensemble de processus (rectangles) et des données (cercles) encapsulés dans un seul ensemble.

Il est clair que le concept d'encapsulation va dans le sens de notre objectif de fiabilité décrit plus haut.

Nous reviendrons plus longuement sur le concept d'encapsulation au chapitre 3.

1.2.2 Dissimulation de l'implémentation et de l'information.

Une bonne encapsulation dissimule les décisions de programmation, au niveau de l'implémentation interne et/ou de l'information interne de l'objet. Encapsulation mène vers une vue interne (privée) et externe (publique) de l'objet.

Figure 1.2 : Les éléments internes de l'objet sont cachés.

```
class X {
public:
      // partie visible et accessible
      methode1() ;
      methode2() ;

private:
      // partie inaccessible ( informations internes )

};


methode1()
{
// implémentation interne de la méthode methode1()
// non visible
}

methode2()
{
// implémentation interne de la méthode methode2()
// non visible
}
```

Fig. 1.3 : Dissimulation de l'information et de l'implémentation en C++[6].

1.2.3 Objectification.

Un objet est une encapsulation de données et de procédures. Chaque étiquette[7] par laquelle il peut être référencé lui est unique. Son étiquette sera généralement le nom de la variable par laquelle il est référencé.

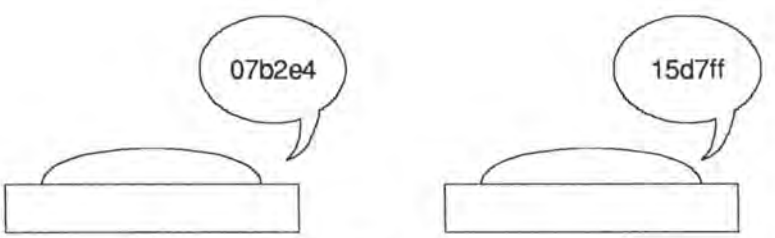

Figure 1.4: Les Objets et leur étiquette

Un objet est caractérisé par les actions qu'il subit, fait ou demande aux autres objets. Un objet ne présente que sa partie publique aux autres objets. Tout objet peut avoir des états bien définis par lesquels il passe.

---

[6] Généralement, en C++, la déclaration et l'implémentation des méthodes d'un objet sont dans des fichiers différents.

[7] J'ai appelé "étiquette" le concept anglais de "handle" d'un objet, c'est à dire le nom d'une variable le contenant. Ce concept est proche de celui de l'identité d'un objet sur laquelle nous reviendrons plus tard.

Figure 1.5 : Notation pour l'Objet AVION

```
class AVION {
public:
      tourner( );
      grimper( );
      accélérer( );
      atterrir( );
      ...
private:
      ...
};
```

Fig. 1.6 : Exemple C++.

1.2.4 Message.

Un message est le moyen par lequel un objet demande à un autre objet d'exécuter une opération sur lui-même ou sur l'objet en question. Pour envoyer un message, un objet doit avoir accès à l'étiquette de l'objet destination et doit connaître le nom de l'opération désirée.

Un message peut être informatif, interrogatif ou impératif

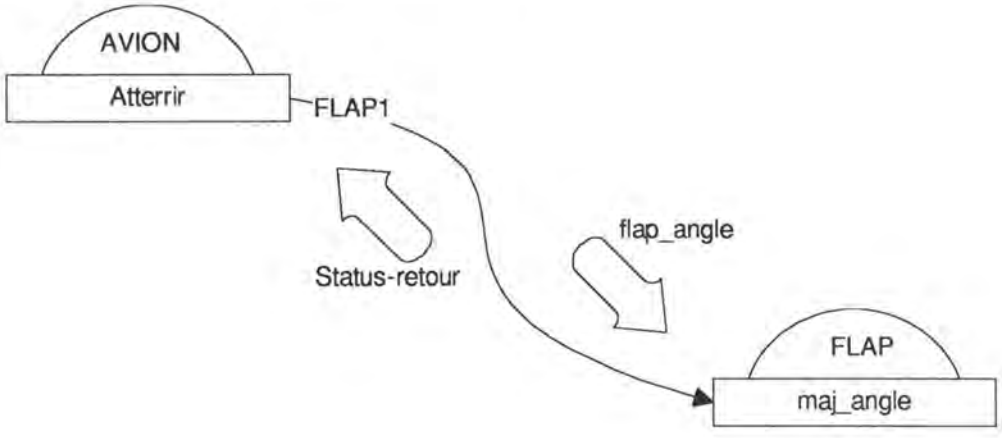

Figure 1.7 : Un AVION envoie un message à un objet FLAP
l'étiquette de l'objet FLAP est la variable FLAP1 connue localement.

En C++, un message constitue un appel à une méthode d'une autre classe à laquelle la classe appelante a accès.

```
Class AVION {
public:
      ...
      atterrir( ) { ... ; flap1.maj_angle (45) ; ... ; }
      ...

private:
      FLAP flap1;
};

Class FLAP {
public:
      ...
      status maj_angle (int ang) { angle = ang ; return ...}
      ...

private:
      ...
      int angle;
      ...
};
```

Fig. 1.8 Exemple C++.

1.2.5 Classification.

Les objets ayant le même comportement et la même structure appartiennent à la même classe. Un objet est une instance de sa classe. Le programmeur ou le concepteur construit des classes. A l'exécution, la classe peut être amenée à créer de nouveaux objets de sa classe. C'est ce qu'on appelle l'instanciation.

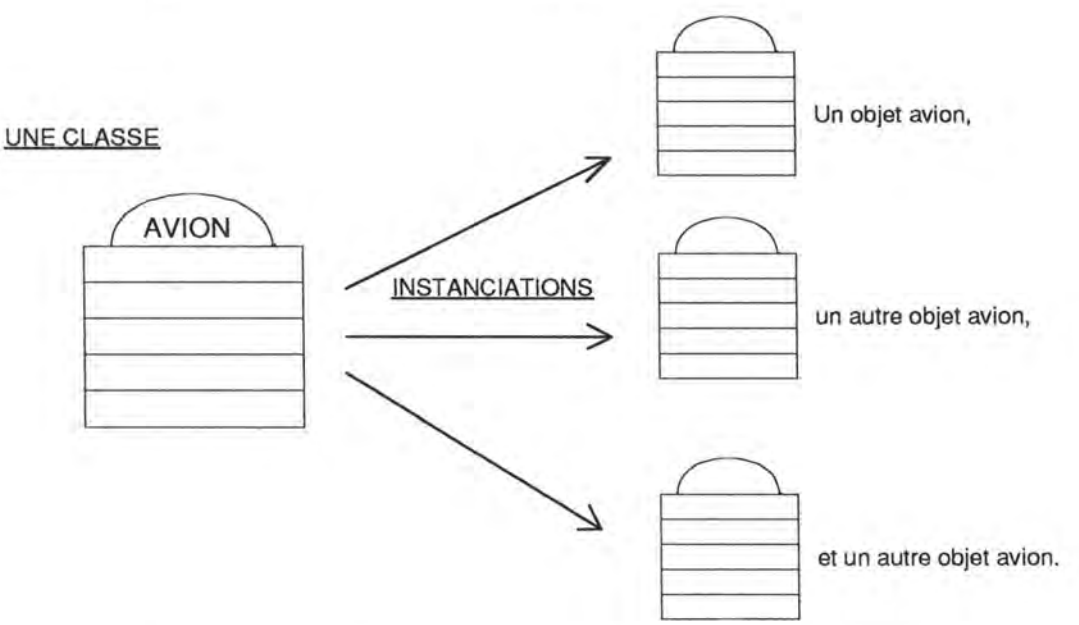

Fig. 1.9 : Les classes construisent des instances sur demande.

Une classe peut contenir des règles que toutes ses instances devront satisfaire. C++ possède les concepts de constructeurs et destructeurs de classe. Chaque instance de la classe sera générée par des règles définies dans le(s) constructeur(s) de la classe. Elles seront détruites par celles définies dans le destructeur dès que le programme n'aura plus besoin de ces instances.

```
class X {
public:

      // Un CONSTRUCTEUR
      X( ) { a = 0 ; b = "" ; ptr = NULL ; }

      // Un autre CONSTRUCTEUR
      X(int aa, string bb, string *p)
            {
                  a = aa;
                  b = bb;
                  ptr = p;
            }

      // Un DESTRUCTEUR
      ~X( ) { delete ptr ;}

private:
      int a;
      string b;
      string *ptr;
};


main ()
{
      // instanciations d'un objet instance_of_x de la classe X
      X instance_of_x;
      // instanciation d'une objet other_instance_of_x de la classe X
      X(0,"...",NULL) other_instance_of_x;
...
}
```

Fig. 1.10 : Exemple de constructeurs et destructeur de classe en C++.

Nous reviendrons sur ce concept de classification au chapitre 3. Nous nous attarderons en particulier sur le concept de **classe** par rapport à celui de **type**.

1.2.6 Structure interne d'une classe.

Les classes ont une structure interne :

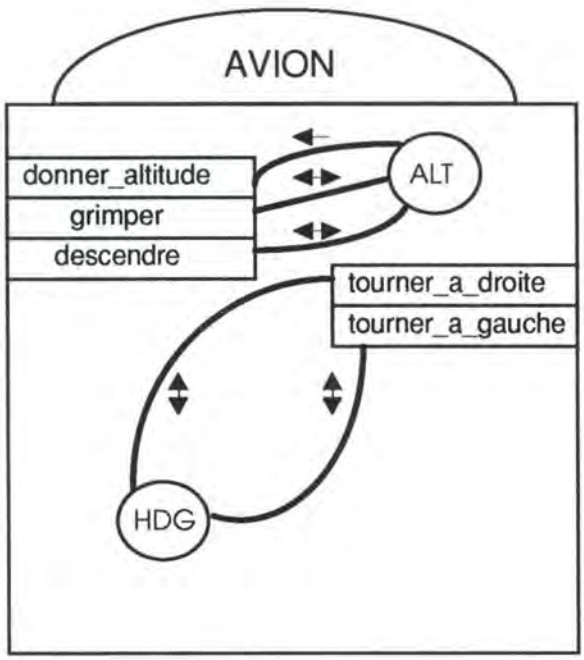

Figure 1.11 : structure interne de la classe AVION
Les méthodes (procédures), les variables (attributs) et leurs interactions.

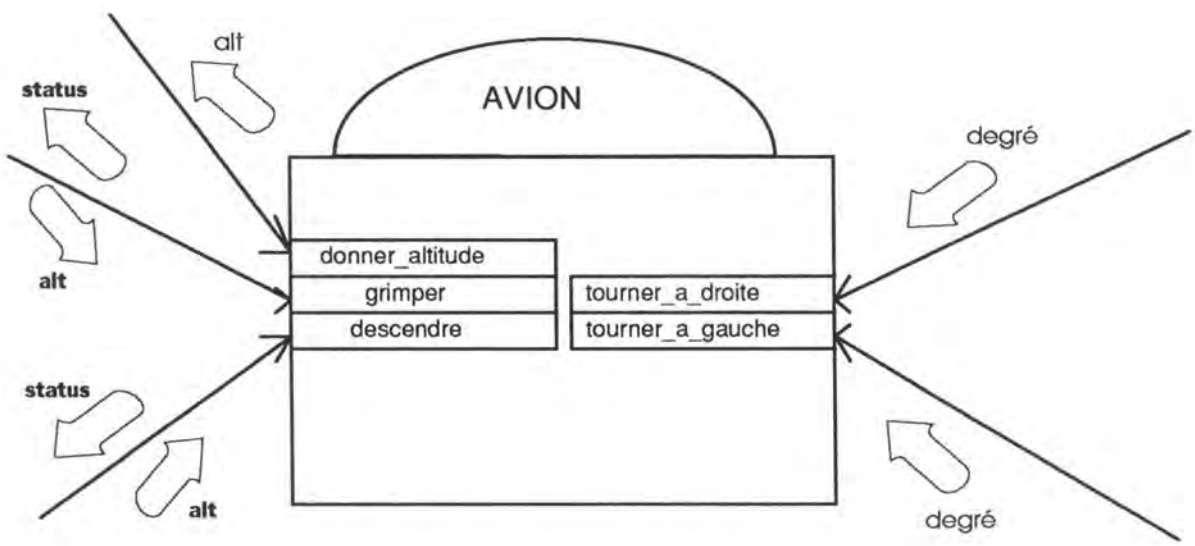

Fig. 1.12 : Structure externe de la classe AVION
Les méthodes et les messages qui les invoquent.

```
class AVION {
public:
      // ... constructeurs et destructeurs
      int donner_altitude( ) {return alt ;}
      grimper (int a) {alt = a ;}
      descendre (int a) {alt = a ;}
      tourner_a_droite( ) {...;}
      tourner_a_gauche() { ...;}

private:
      int alt;
      int hdg;
};

main ( )
{
      avion a;
      int altitude_actuelle = a.donner_altitude;
      a.tourner_a_droite;
      a.grimper (1000);
      ...
}
```

Fig. 1.13 : Illustration C++ de la structure interne et externe d'une classe.

1.2.7 Héritage.

L'héritage donne la possibilité à une instance d'une classe X d'utiliser, non seulement les méthodes et les variables définies pour X, mais aussi celles définies pour une classe désignée comme étant un "ancêtre" de X. L'héritage permet de construire de nouvelles classes à partir de classes déjà existantes moins spécialisées. (Programmation incrémentale).

Une nouvelle classe est spécifiée en désignant la manière dont elle est différente, dont elle est une extension, de celle de qui elle hérite.

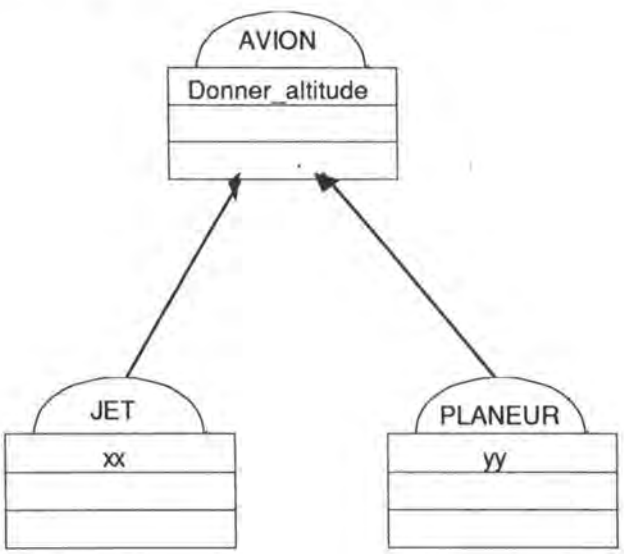

Fig. 1.14 : JET et PLANEUR héritent de la méthode Donner_altitude.

```
class AVION {
public:
      ...
      int donner_altitude ();
private:
      ...
}

class PLANEUR public AVION {
public:
      ...
      yy( ) ;
private:
      ...
};

class JET public AVION {
public:
      ...
      xx( ) ;
private:
      ...
};

main ( )
{
      AVION a;
      PLANEUR p;

      // OK
      int altitude_actuelle_planeur=p.donner_altitude();

      // ERREUR
      a.xx( );
}
```

Fig. 1.15 : Exemple C++ d'héritage.

L'objectif de réutilisabilité est clairement apparent dans le concept d'héritage. L'emploi de l'héritage pour spécifier et particulariser progressivement des modules en fonction d'une application particulière apparaît comme une solution très prometteuse.

Quant à l'objectif de l'extensibilité, il va de pair avec le concept d'héritage.

Nous reviendrons sur la notion d'héritage au chapitre 3.

1.2.8 Polymorphisme / surcharge.

Le polymorphisme est le moyen pour une variable de pointer vers un objet (c'est-à-dire qu'elle possède son étiquette) dont la classe n'est pas nécessairement connue. La surcharge donne la possibilité d'utiliser le même nom pour une méthode pour arriver à pouvoir exécuter une ou plusieurs actions différentes. Le polymorphisme va bien sûr de pair avec la surcharge.

Ces mécanismes sont largement supportés par le C++.

Nous reviendrons également sur cette notion au chapitre 3.

1.2.9 Généricité.

La généricité offre la possibilité de créer des structures similaires pour différents objets (réutilisation). Le C++ assume la généricité, nous verrons un exemple plus tard dans le cadre du développement.

Pour conclure, il faut quand même souligner que globalement la programmation objet, de part sa structuration plus fine, ses propriétés d'encapsulation et de généricité semble apporter une réponse nettement plus satisfaisante que celles de la programmation classique.

Chapitre II : Les méthodologies orientées objets.

2.1 Généralités

Une méthode d'analyse ou de design est une approche cohérente pour décrire un problème. Ce système peut être vu comme un système informatique, ou comme un processus de travail qu'un système informatique devra supporter. Un tel système est très complexe ; une approche classique est d'utiliser des techniques différentes, chacune d'entre elles soulignant certains aspects du système et négligeant certains autres. Par exemple, le modèle Entité-Relation (ER) se préoccupe des données du système, mais se désintéresse des processus sur ces données. Typiquement, une méthode va utiliser plusieurs techniques telles que celle-ci pour couvrir tous les aspects de la conception du système.

Habituellement, les techniques ne sont pas exclusives et la même technique peut être utilisée dans une grande variété de méthodes. Bien que la technique de base soit la même, chaque méthodologiste a l'habitude de modifier une technique commune en introduisant une nouvelle notation et des nouveaux concepts de modélisation. Donc les méthodes de Schlaer/Mellor et de Rumbaugh sont deux méthodologies utilisant des dialectes différents de la technique ER. Elles diffèrent dans certains concepts (il n'a pas le concept d'agrégation dans Schlaer/Mellor) et dans la notation (les notations de cardinalités sont différentes).

2.1.1 Les différentes vues d'un système

Chaque technique de modélisation souligne certains aspects du système et néglige certains autres. Elle représente donc une vue particulière d'un système. En général, on considère trois vues générales d'un système [MOO1] : les données (data view), les comportements (behavioural view), l'architecture (architectural view).

La "**data view**" décrit les parties statiques et structurelles du système et néglige les processus. Dans les systèmes de base de données, cela correspond au schéma de la base de données ; dans l'optique orientée objets, cela correspond aux types, aux attributs et aux relations entre eux. La technique des diagrammes ER est un exemple de "data view".

La "**behavioural view**" décrit les changements du système. Idéalement on pourrait lui construire un compilateur de description du système et donc en

théorie, être exécutable. Le diagramme de transition d'états est un exemple de technique utilisée pour la "behavioural view".

Entre la "data view" et la "behavioural view", on peut fournir une description complète d'un système devant être informatisé, mais cette description peut devenir trop complexe à utiliser. Une technique d' "**architectural view**" peut diviser un système en sous-systèmes. Le diagramme des flux est un exemple de l' "architectural view".

Cette division en trois vues n'est pas la seule option possible. Il est courant de voir une classification à deux niveaux : les données et l'architecture.

2.1.2 Deux aspects importants

Il existe deux aspects importants pour une technique : [MOO2] sa rigueur et sa compréhensibilité.

Une technique rigoureuse est celle qui ne permet pas d'ambiguïté dans son interprétation. La **rigueur** est poussée jusqu'à son extrême dans les méthodes formelles[8]. La rigueur est très importante puisque sans elle, les ambiguïtés d'un modèle devront être résolues à l'implémentation dans uns sens qui ne sera peut être pas cohérent avec le reste du modèle.

La **compréhensibilité** est souvent sous-estimée[9], mais est aussi importante.

La première raison est qu'une technique difficilement compréhensible sera difficile à apprendre ; il sera donc plus difficile pour de nouvelles personnes de comprendre les modèles. C'est important dans le sens ou cela diminue premièrement, les chances des utilisateurs de contribuer réellement au processus d'analyse et, deuxièmement, les chances d'un modèle pour sa réutilisation[10].

La deuxième raison, moins souvent avancée, mais aussi importante, d'utiliser des méthodes compréhensibles est que, pour produire un bon modèle, il est vital que le modélisateur aie une bonne vision du système. Si un modèle est difficile à comprendre, cela rend la vie plus dure à l'analyste et au concepteur avec pour conséquence un modèle de moins bonne qualité. Bien que la rigueur et la compréhensibilité soient considérées comme ayant des intérêts mutuellement

---

[8] La méthodologie orientée objets OBLOG se rapproche de telles méthodes formelles.

[9] Surtout par la communauté des méthodes formelles.

[10] Les modèles réutilisables sont un élément clé de la programmation réutilisable.

exclusifs, elles peuvent être résolues. La plupart des modélisateurs de données trouvent que les diagrammes ER sont une vue compréhensible des données ; de plus les diagrammes ER peuvent être définis de façon complètement formelle.

2.1.3 Conclusion

Donc, une méthode consiste en un nombre de techniques différentes qui se combinent pour former un modèle cohérent d'un système. Comme pour ces techniques de modélisation , un ensemble d'heuristiques[11] de modélisation sont nécessaires pour décrire comment construire ce modèle. La majorité du processus d'analyse et de design est impossible à décrire dans un livre ; l'analyse et le design tiennent autant de l'art que de la science.

2.2 Typologie et description des différentes méthodologies orientées objets

Les méthodes d'analyse et de design tombent dans deux catégories [MOO8]: d'un côté, celles qui imitent les méthodes structurées existantes qui ont trois notations séparées pour les données : les comportements, les processus et l'architecture, de l'autre celles qui affirment que, puisque les objets combinent les méthodes et les données de façon inhérente, une seule notation est nécessaire. Appelons ces deux approches respectivement les approches ternaires et unitaires.

L'**approche ternaire** a l'avantage que les adeptes des méthodes structurées existantes seront largement familiers avec la philosophie et les notations. Ceci assurera une transition plus aisée vers l'approche orientée objets pour les personnes déjà habiles dans les méthodes traditionnelles[12].

A l'opposé, les **approches unitaires** sont plus cohérentes avec la métaphore de la conception orientée objets, et plus facile à apprendre à partir de zéro[13].

En analysant les modèles d'analyse et de design orientés objets existants, on remarque que :

- Certaines méthodes comme celle de Coad/Yourdon sont simples mais manquent de moyens pour décrire la dynamique d'un système.

---

[11] Le terme "heuristique" est utilisé ici, car ils sont toujours décrit vaguement, mais donnent de bonnes lignes directrices quant à la façon de s'y prendre pour passer de l'état de la feuille blanche à celui du résultat final.

[12] Exemples : Rumbaugh, Ptech (Martin/Odell), Booch.

[13] Exemple : Coad/Yourdon.

- Certaines méthodes comme celle de Rumbaugh sont riches, mais très complexes à assimiler.

Nous allons décrire brièvement une méthodologie de chaque type. Il s'agira de la méthodologie de Coad/Yourdon pour le type unitaire, et de la méthodologie de Rumbaugh pour le type ternaire.

2.2.1 La méthodologie unitaire de Coad/Yourdon :

2.2.1.1 LES ACTIVITES DE L'ANALYSE.

Cette méthode est constituée de cinq activités : [MOO6]

- Identifier les "subject areas".
- Identifier les objets.
- Identifier les structures.
- Identifier les attributs.
- Identifier les méthodes.

Identifier les "subject areas"[14] est un processus top-down dont le but est de diviser le problème en sous-problèmes plus faciles à gérer. Ils sont utiles, mais ils ne constituent pas des objets en tant que tels et n'ont aucun statut formel ou sémantique dans le modèle.

On peut **identifier les objets** en donnant des noms dans une description textuelle du problème ou en utilisant des systèmes d'analyses conventionnels. C'est la même chose que de la modélisation d'entités excepté que des méthodes peuvent déjà être découvertes à ce niveau-ci.

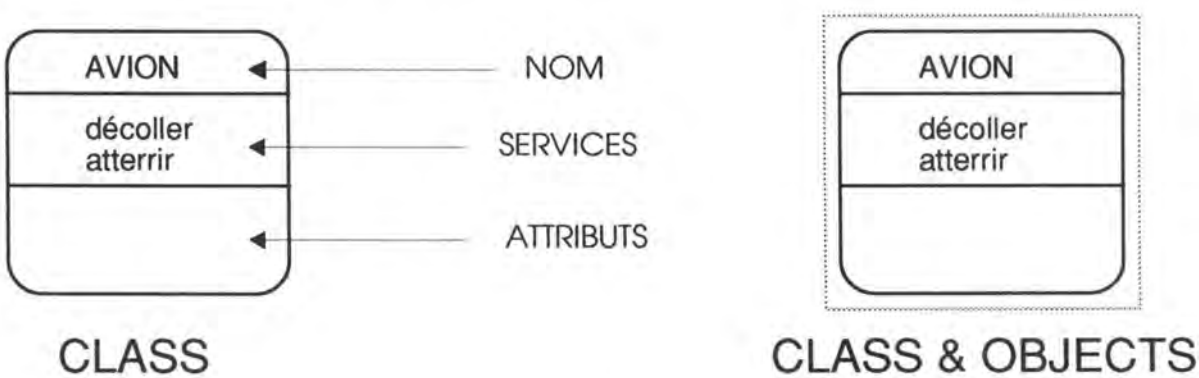

Fig. 2.1 : Notations pour une classe et pour une classe et ses occurrences (objets).

[14] Cela consiste en fait en un groupement d'objets formant un concept entre eux.

Identifier les structures consiste à grouper des objets dans des hiérarchies de classification et de composition. Dans la notation, l'héritage multiple est permis, mais la méthode ne supporte pas de technique pour les enlever l'ambiguïté des conflits qui peuvent survenir.

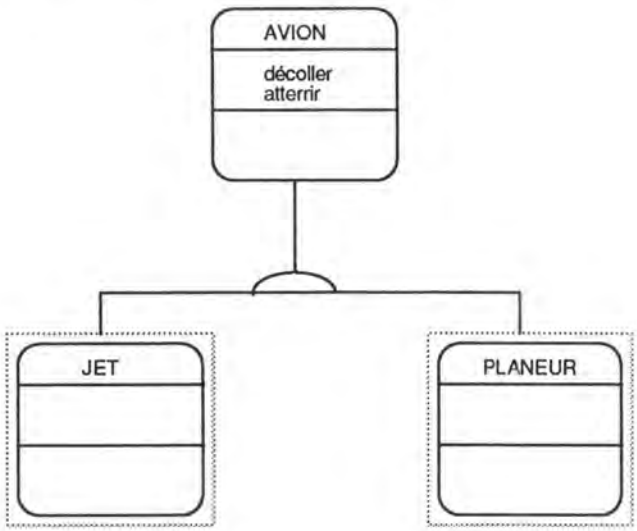

Fig. 2.2 : Structure de spécialisation/généralisation (relation IS-A).
Un AVION est un JET ou un PLANEUR.

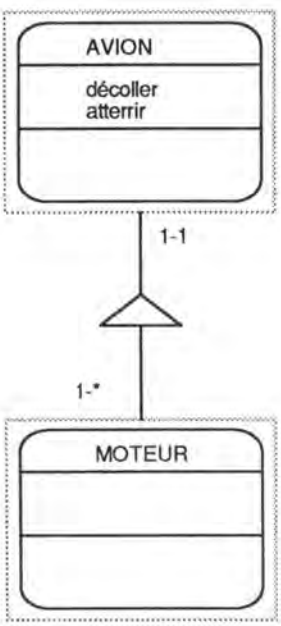

Fig. 2.3 : Structure part/whole.
Un MOTEUR fait partie d'un AVION (relation is-part-of).
Un AVION contient 1-n MOTEURS.

La façon d'**identifier les attributs** n'est pas différente de celle de la modélisation traditionnelle de données, excepté que les attributs hérités ne sont pas annotés.

Identifier les méthodes (appelées services dans cette méthodologie) repose aussi sur les techniques traditionnelles.

Remarque : la méthode fut la première à réintroduire, dans l'analyse orientée objets, l'idée que les relations sémantiques entre des objets doivent être retenues [MOO8] : l'état d'un objet est la collection d'associations que cet objet possède. Il est équivalent de dire que l'état d'un objet est défini par l'ensemble de ses types.

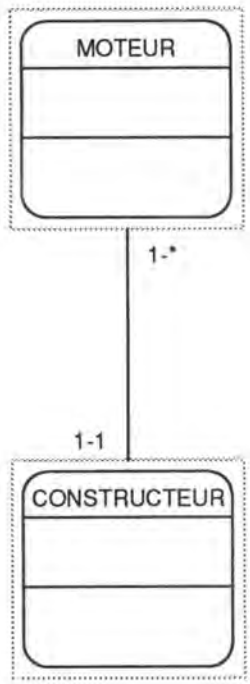

Fig. 2.4 : Relation entre objets : instance connection.
Un MOTEUR possède un CONSTRUCTEUR.
Un CONSTRUCTEUR construit 1-n MOTEURS.

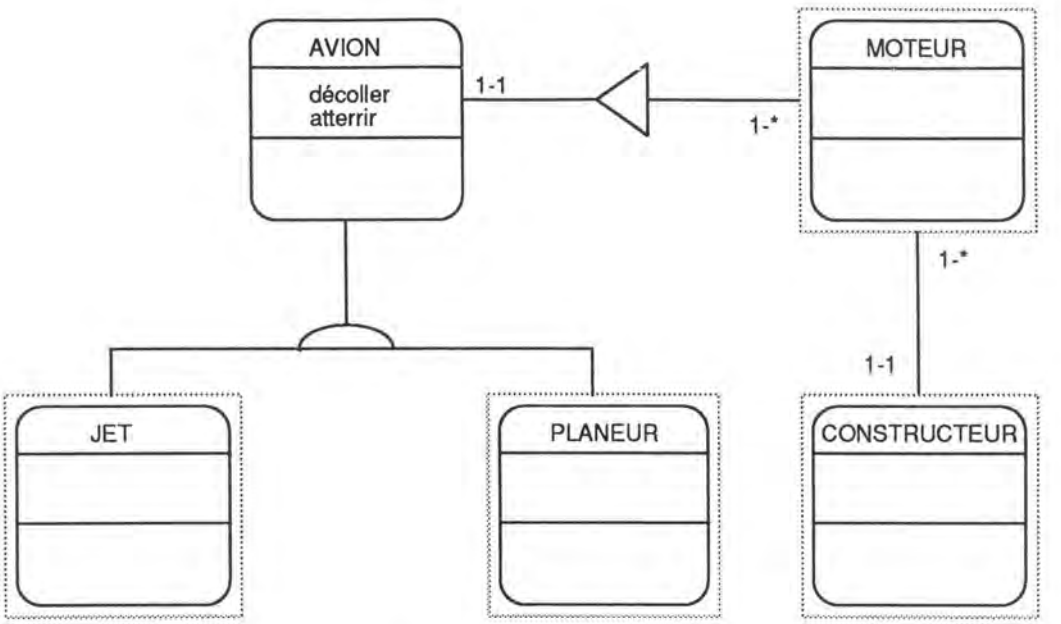

Fig. 2.5 : Exemple complet.

2.2.1.2 LES COMPOSANTS DU DESIGN.

La méthodologie ajoute à ces cinq activités, quatre composants [MOO7]. Le design consiste principalement :

- A raffiner le produit de l'analyse dans ce que les auteurs appellent "the problem domain component" du design.
- A ajouter trois nouveaux composants appelés

    1. the human interaction component
    2. the task management component
    3. the data management component

On insiste beaucoup sur la grande similarité de notation pour l'analyse et le design et sur la façon dont elle rend plus facile la transition entre l'analyse et le design[15].

2.2.1.3 EVALUATION

De vigoureuses critiques ont été faites sur le premier livre de Coad/Yourdon au sujet de l'analyse orientée objets [MOO5]. Le sentiment général était que cela manquait de rigueur, qu'il existait une confusion entre les "class" et les "class & objects" et n'avait que très peu de choses en plus que le modèle ER. Cependant, la seconde édition [MOO6], avait beaucoup à offrir.

---

[15] Cette transition est fortement ressentie dans les méthodes traditionnelles.

Le coeur de la technique reste dans une vue données avec les diagrammes ER. Ceux-ci montrent à présent les agrégations, les associations, le sous-typage.

Même si cette méthodologie fait partie de ce que nous avons appelé les méthodologies unitaires, elle souligne certains aspects du comportement et de l'architecture :

Le **comportement** est souligné

- En donnant des noms aux opérations sur les classes.
- En les définissant dans la classe par des tableaux de transition.

Le comportement inter-objets est mis en valeur par des messages dans le diagramme.

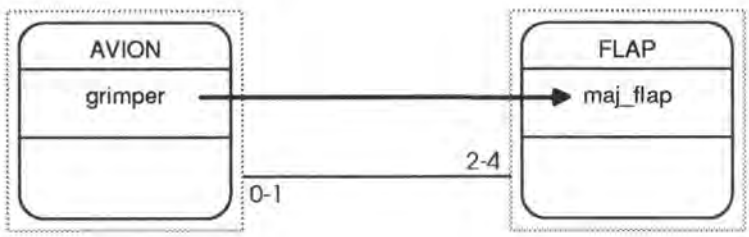

Fig. 2.6 : Représentation des messages.

L'**architecture** est définie en groupant les objets dans des "subject areas". Ces groupes sont également notés sur le diagramme.

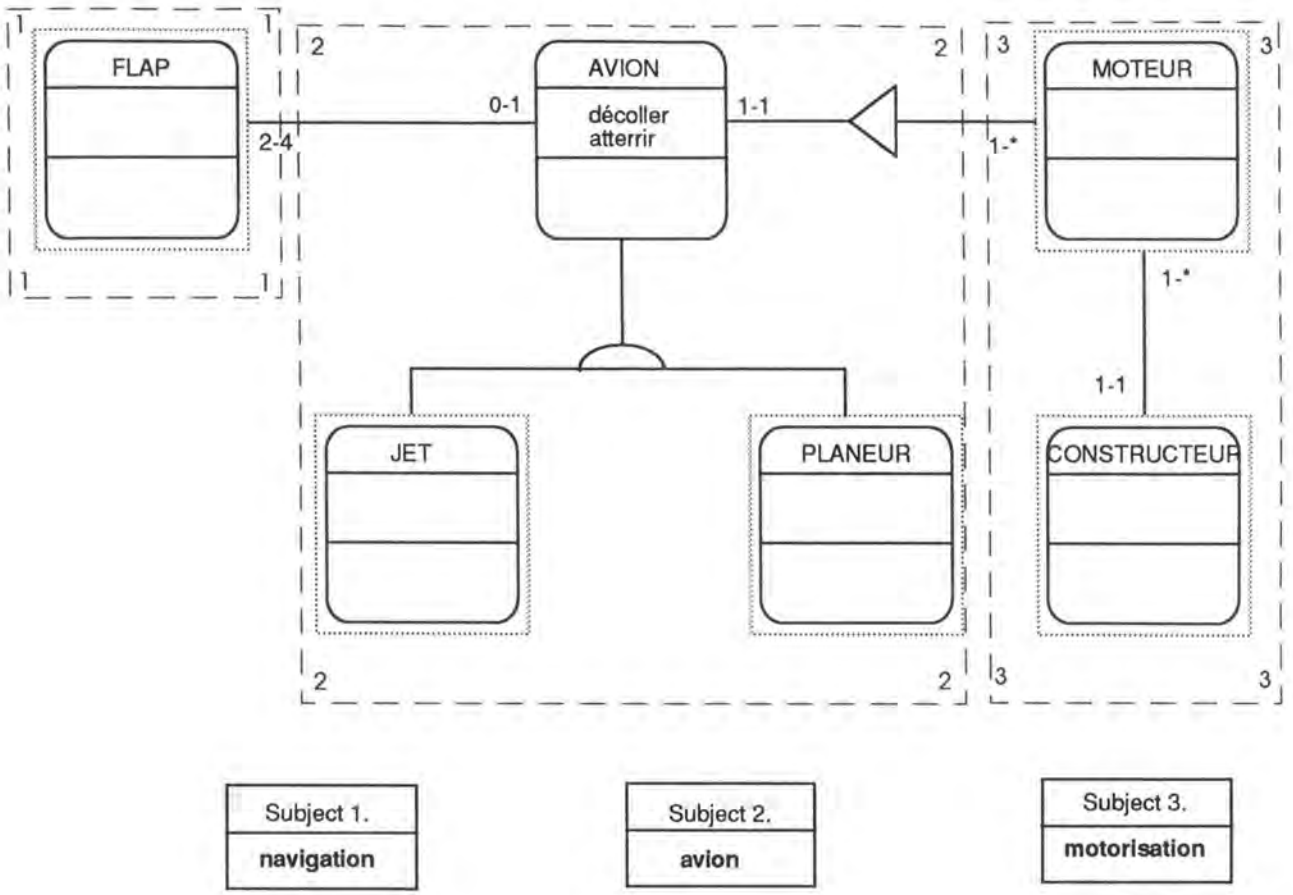

Fig. 2.7 : Illustration des "subject areas".
Le subject 1. Navigation est représenté par la surface entourée par des traits discontinus dont les coins sont numérotés par 1, etc.

La plus grande force de cette méthodologie est sa simplicité. Cependant, le sentiment général est que, bien qu'elle soit un bon point de départ, cette méthode ne serat pas efficace pour des projet trop conséquents Cette méthodologie devrait se limiter à de petits projets, car elle serait vite dépassée par des projets trop consistants.

Cette méthodologie reste cependant très attractive du point de vue des données, ce qui nous intéresse tout particulièrement dans ce mémoire.

2.2.2 La méthodologie ternaire de Rumbaugh

Le livre de Rumbaugh [MOO4] donne une définition claire d'un processus et des modèles associés, qui est similaire à un certain nombre d'autres méthodes moins documentées. Nous allons utiliser un exemple simple basé sur un problème réel ; le but est d'analyser une application simple de dessin qui permet à l'utilisateur de dessiner des lignes et des boîtes et de les déplacer :

"Un dessin est un graphe composé de boîtes rectangulaires de différentes tailles situées dans des positions différentes de l'écran, et de lignes. Une ligne peut être attachée à au plus deux boîtes, mais ne peut pas être attachée deux fois à la même boîte. Un ensemble de boîtes est connecté s'il y a un chemin entre elles. Pour simplifier le problème, nous demandons que tous les sous-graphes connectés soient acycliques. Les lignes peuvent, en option, soit avoir une couleur, soit être discontinues.
L'utilisateur devrait être capable de créer de nouvelles lignes et de nouvelles boîtes, d'attacher une ligne à une boîte, de déplacer une boîte avec toutes les lignes qui lui sont attachées et les boîtes connectées."

2.2.2.1 DATA VIEW.

La "data view" s'appelle le "the object model" chez Rumbaugh.. Il utilise une forme de diagramme ER pour capter la structure statique en soulignant les classes d'objet d'un système et les relations entre eux.

Les **classes** peuvent avoir des attributs contenus dans l'objet. Les instances de chaque classe ont un identifiant unique, qui n'est pas montré, mais laissé implicite.

Une **relation** entre n classes définit des "sets" de n-uplets, chaque n-uplet contenant l'identifiant d'un set d'objets reliés. Une instance d'une relation est appelée un lien (link). Les relations sont notées par des lignes dessinées entre les entités. Le nombre de participants dans une relation peut être spécifié :

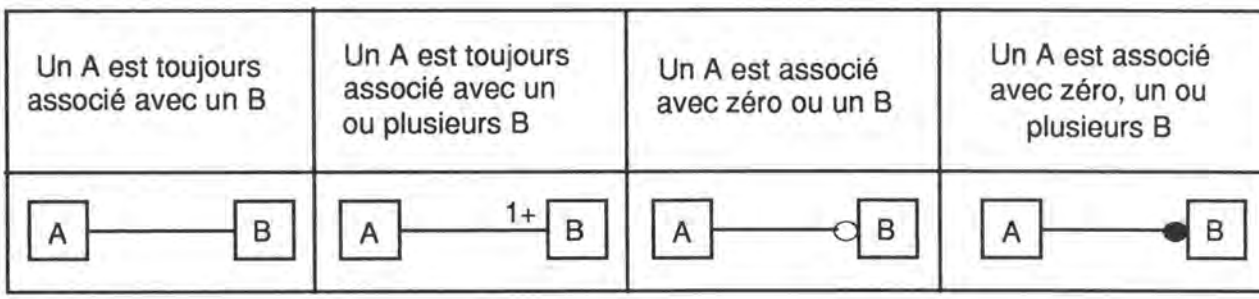

Fig. 2.8 : Les notations de connectivité utilisées par l'object model".

Ces notations peuvent sembler inhabituelles pour une personne habituée à travailler avec le modele ER, puisque les rôles, ou du moins leur équivalent, se situent à des places inverses par rapport au modèle ER. Notons que les contraintes de connectivité peuvent être notées simplement par un couple d'entier.

La **spécialisation** est une relation d'héritage entre des classes. Une sous-classe est une spécialisation d'une classe si elle participe à toutes les relations de cette classe. Une sous-classe peut avoir des attributs supplémentaires et/ou participer à des relations différentes de celles de la classe mère.

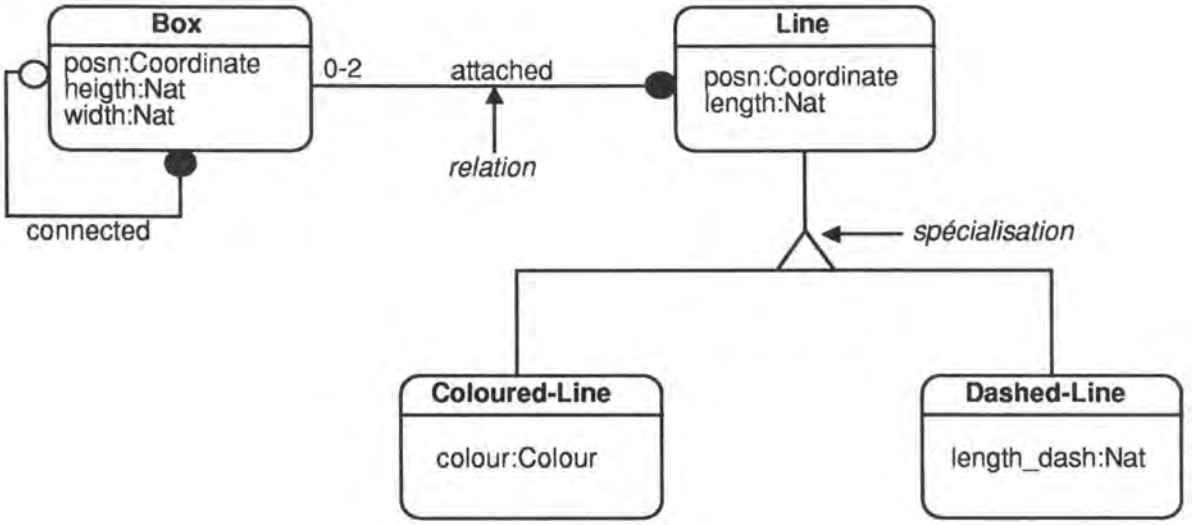

Fig. 2.9 : L' "object model" de Rumbaugh pour notre problème.

2.2.2.2 BEHAVIOURAL VIEW.

Dans la méthodologie de Rumbaugh, la "behavioural view" est appelée "the dynamic model". Ce modèle utilise des machines d'états finis étendues pour spécifier le comportement des classes. L'état d'une machine est une abstraction des valeurs d'attributs et des liens (links) d'un objet. Les machines d'états ont un modèle de communication par événements. Un **événement** est une unité de communication nommée, en sens unique et instantanée qui peut avoir des données qui lui sont associées. En recevant un événement, une machine peut changer d'état, créer un autre événement en sortie et/ou exécuter une action. Une **action** est une opération instantanée qui met à jour les valeurs d'attributs ou de liens.

La figure 2.10 montre la machine états pour la classe `Box`. Il y a seulement un état : READY. Un objet Boîte (Box) répond à deux événements, `move` et `attachline(l)`. L'événement `move` est invoqué par l'utilisateur : son effet est de créer des événements `move` pour tous les objets Ligne (Line) attachés et pour tous les objets Boîte connectés. L'événement `attachline(l)` ajoute `l` au lien attached.

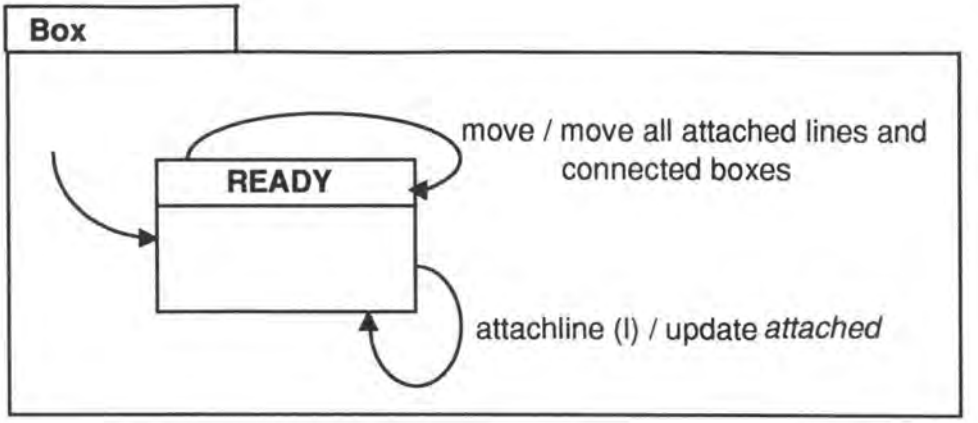

Fig. 2.10 Machine états pour la classe `Box`.

La machine d'état pour la classe Line illustre l'imbrication de diagrammes. Une Ligne va toujours répondre à un événement `move` en mettant à jour son attribut posn. L'état READY contient une combinaison OU des états DETACHED, ATTACHED_ONE et ATTACHED_TWO. Une ligne ne peut recevoir que deux événements `attachbox` ; par la suite, tous les événements `attachbox` resteront sans effet. `Attachbox(b)` ajoute la boîte au lien attached et envoie un message `attach(self)` à la boîte b.

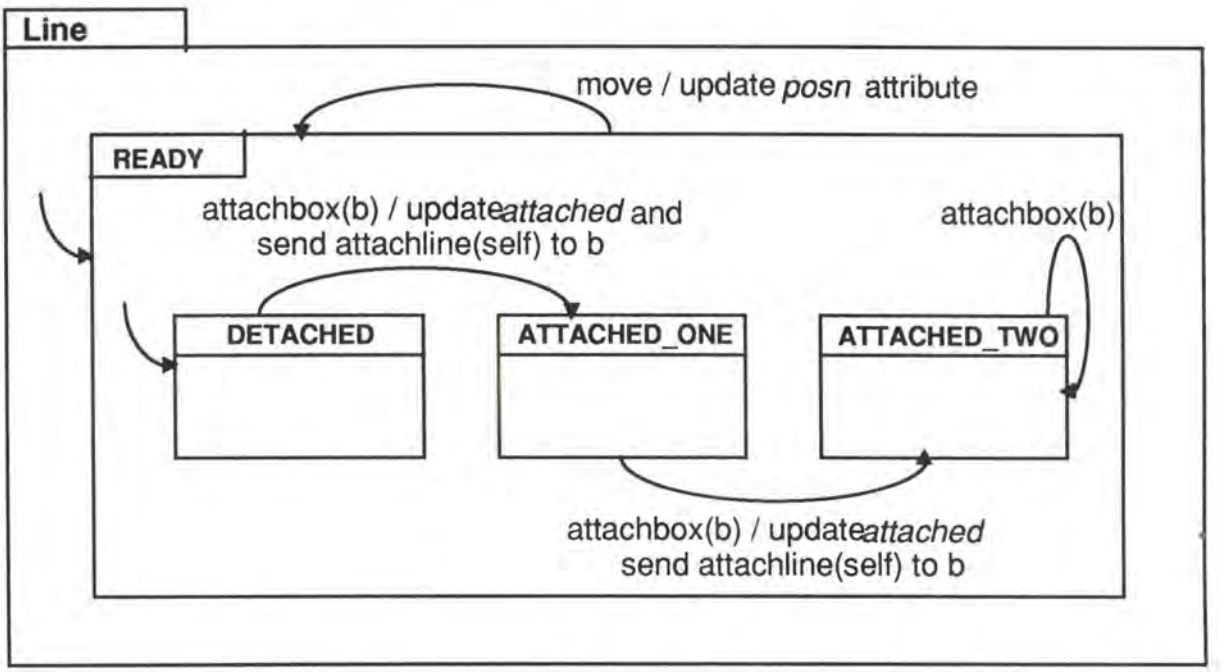

Fig. 2.11 Machine état pour la classe `Line`.

2.2.2.3 ARCHITECTURAL VIEW.

L' "architectural view" est appelée "the functional model". Il montre plus le comportement du système que celui des objets individuels. Dans la méthodologie de Rumbaugh, ce sont les "data flow diagrams" (DFDs) qui sont utilisés dans ce but. Un DFD montre le flux des valeurs de données de leurs sources vers leurs destinations via les processus qui les transforment. Un DFD ne montre pas les informations de contrôle comme les moments auxquels les processus sont exécutés ou les décisions parmi les chemins dynamiques.

Un DFD contient :

- des processus qui transforment les données
- des flux de données qui transportent les données
- des containers de données (data stores) qui sont des containers passifs
- des acteurs qui produisent et utilisent des données.

Bien que notre simple exemple ne le montre pas, les DFDs peuvent être structurés hiérarchiquement, chaque processus étant décomposé en un DFD des sous-processus. Le niveau le plus bas des processus sont spécifiés en langage naturel, en pseudo-code, par des tables de décision, etc.

La figure 2.12 spécifie le comportement de l'opération `move_box`. Elle montre que le traitement est effectué en déplaçant la boîte originale, et en déplaçant ensuite, chacune de ses lignes attachées. D'autres boîtes devront peut-être être déplacées récursivement. Le processus continue jusqu'à ce que les boîtes extrêmes du graphe aient été déplacées. Les containers de données (data stores) boîtes (boxes) et lignes (lines) contiennent les valeurs de Boîte et Ligne. Le container de données, position change, est utilisé pour retenir le changement de position requis Δposn.

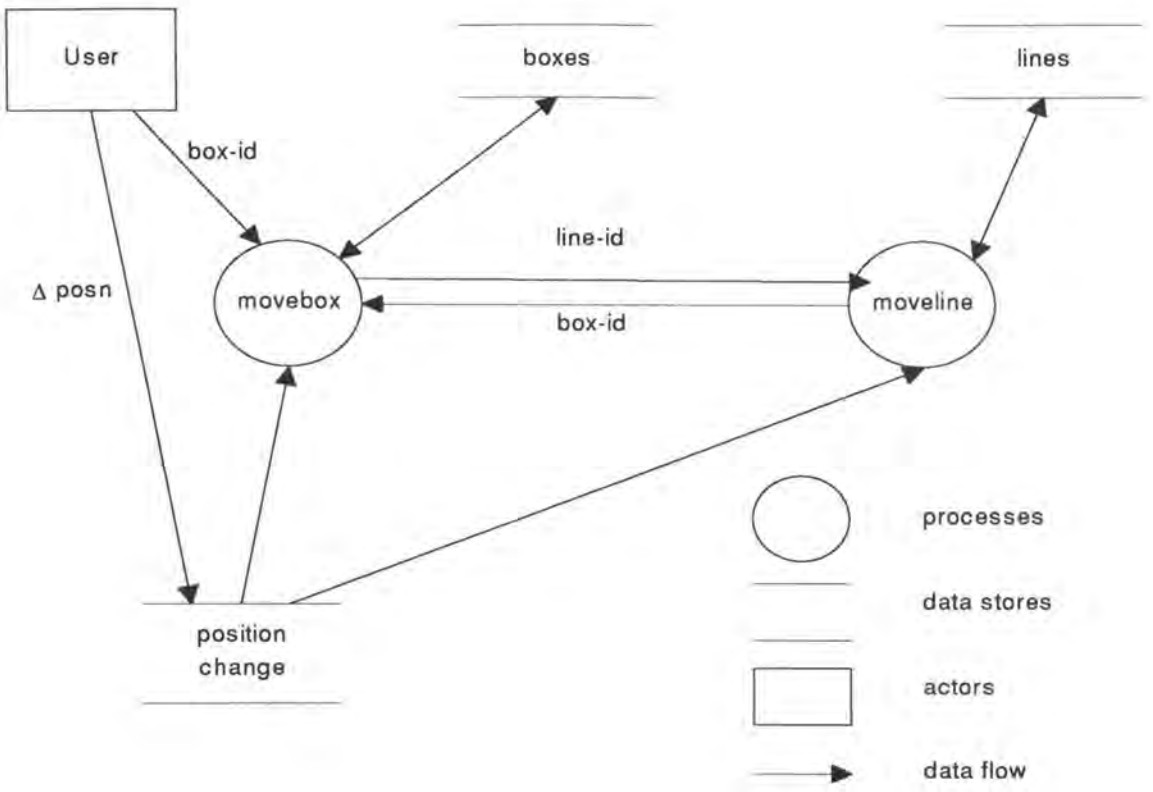

Fig. 2.13 DFD de l'opération `move_box`.

2.2.2.4 LE PROCESSUS D'ANALYSE.

Rumbaugh suggère que la procédure soit divisée en plusieurs étapes successives:

- Décrire le problème en langage naturel
- Construire un modèle objet accompagné de son dictionnaire de classes, d'attributs et de relations
- Construire un modèle dynamique accompagné de diagrammes donnant des exemples qui expliquent comment des événements parcourent le système en réponse aux opérations du système
- Construire un modèle fonctionnel accompagné des contraintes qui peuvent exister entre les objets
- Itérer les étapes supérieures à la recherche de classes, de relations, événements manquants, etc.

2.2.2.5 EVALUATION.

Nous avons vu que la méthode utilise trois techniques :

- un dialecte des diagrammes ER pour la "data view"
- les machines états pour le comportement

- les DFD pour l'architecture

La technique ER est particulièrement riche ; elle supporte les sous-types partitionnés et non partitionnés, les relations et les classes dérivées et l'agrégation. Les machines états apportent une modélisation de transition d'état très puissante. Le DFD force le lien aux objets par la décomposition, de telle sorte que les bulles de bas niveaux sont des opérations sur les classes.

Chapitre III : Les systèmes de gestion de base de données orientée objets

3.1 Introduction

Actuellement, les systèmes de gestion de bases de données orientées objets (SGBDOO) reçoivent une grande attention, tant du point de vue théorique que du point de vue expérimental [OODB1]. Il existe un débat considérable à propos de la définition de tels systèmes. Contrairement au papier original de Codd[16] qui donnait une spécification claire d'un système de base de donnée relationnel (modèle de donnée et langage de requête), aucune spécification de ce genre n'existe pour des systèmes de base de données orientée objets. Un avis se construit peu à peu sur les caractéristiques de la famille des SGBDOO. On remarque également l'absence d'une base théorique solide à ces systèmes. Ceci a pour conséquence que le consensus sur le modèle des données est presque impossible à atteindre.

La situation d'implémentation est similaire à celle des systèmes de bases de données relationnelles dans le milieu des années septante. Pour le modèle relationnel, même s'il y avait des désaccords sur quelques points spécifiques, comme la forme du langage de requête, ou si la forme des relations devraient être des "set" ou des "bag", ces distinctions étaient dans la plupart des cas, superficielles et il existait un modèle sous-jacent commun. Aujourd'hui, on choisit simultanément la spécification du système et la production de la technologie pour supporter son implémentation. Donc, en respect avec la spécification du système, nous prenons une approche darwinienne : nous espérons qu'une viable implémentation technologique pour un modèle va évoluer simultanément.

Malheureusement, dans l'agitation de l'expérimentation, on risque qu'un système s'impose comme étant le système, non pas parce qu'il est le meilleur, mais parce qu'il est le premier à apporter une partie des fonctionnalités demandées par le marché. C'est un exemple classique et malheureux du milieu de l'informatique de voir un produit précoce devenir un standard de facto, et ne jamais disparaître. Cet exemple est vrai, du moins pour les langages et les systèmes d'exploitation[17].

---

[16] E.F. Codd, "A relational model for large shared data banks", Communication of the ACM, Volume 13, Number 6, (June 1970), pp 377-387.

[17] Fortran, Cobol, Lisp et SQL sont de bons exemples de cette situation.

Cependant, il est à noter que notre but, ici, n'est pas de standardiser des langages, mais d'affiner la terminologie.

Il est important de se mettre d'accord sur une définition d'un système de base de données orientées objets. Nous allons suggérer les caractéristiques que de tels systèmes devraient posséder. Nous avons séparé ces caractéristiques en trois catégories :

- Les caractéristiques obligatoires : celles que le système doit satisfaire pour en avoir la dénomination.
- Les caractéristiques optionnelles : celles qui peuvent être ajoutées pour améliorer le système, mais qui ne sont pas obligatoires.
- Les caractéristiques ouvertes : où le concepteur choisit d'autres solutions aussi acceptables.

3.2 Les caractéristiques obligatoires

Un système de base de données orienté objets doit satisfaire à deux critères :

- Il devrait, premièrement, être un système de gestion de base de données (SGBD).
- Deuxièmement, il devrait être un système orienté objets.

A la limite, il devrait être cohérent avec la foule de langages de programmation orientés objets actuels.

Le premier critère se traduit dans cinq caractéristiques :

- La persistance
- La gestion de mémorisation secondaire
- La concurrence
- La récupération
- Un langage de requête adéquat.

Le deuxième critère se traduit en sept critères :

- Les objets complexes
- Identité d'objet
- Encapsulation
- Les types et les classes
- L'héritage

- Surcharge combinée avec une exécution dynamique
- Extensibilité

3.2.1 Les objets complexes.

Les objets complexes sont construits à partir d'objets plus simples en leur appliquant des constructeurs. Les objets plus simples sont des objets comme des entiers (integers), des caractères (characters), des chaînes de caractères (strings) de longueurs variables, des booléens (booleans) et des nombres décimaux (floats). Il existe de nombreux constructeurs d'objets complexes comme les listes, les n-uplets, les "bag", les "set", les tableaux (arrays), etc. L'ensemble minimum de constructeurs que le système devrait posséder sont les "set", les n-uplets et les listes. Les "**set**" sont importants parce qu'ils sont un moyen naturel de représenter des collections du monde réel. Les **n-uplets** sont importants parce qu'ils sont un moyen naturel de représenter les propriétés d'une entité. Bien sûr, les "set" et les n-uplets sont importants parce qu'ils ont été largement acceptés comme des constructeurs d'objets via le modèle relationnel. Les **listes** ou les tableaux sont importants puisqu'ils capturent la notion d'ordre parmi les événements qui se passent dans le monde réel ; de plus, ils apparaissent dans de nombreuses applications scientifiques où l'on a besoin de matrices ou des séries de données temporelles.

Les constructeurs de l'objet doivent être orthogonaux : un constructeur quelconque devrait s'appliquer à n'importe quel objet. Les constructeurs du modèle relationnel ne sont pas orthogonaux parce que le constructeur du "set" peut seulement être appliqué aux n-uplets et le constructeur du n-uplet ne peut être appliqué qu'aux valeurs atomiques.

Il est à noter que le fait de supporter les objets complexes requiert de posséder les opérateurs appropriés pour manipuler de tels objets comme étant un tout, quelle que soit leur composition : une opération sur un objet complexe doit pouvoir se propager transitivement à tous ses composants[18]. Par exemple, il doit être possible d'effacer, de retirer un objet complexe en entier. Il en est de même pour la production de copies[19]. Bien entendu, des opérations supplémentaires sur les objets complexes peuvent être définies par les utilisateurs du système[20]. Cependant, cette possibilité demande certaines dispositions fournies par le

---

[18] C'est pourquoi nous utiliserons la forme canonique de classe orthodoxe en C++ décrite au chapitre 6.

[19] On parle ici de "deep copy" différente de la "shallow copy" où les composants ne sont pas dupliqués mais sont référencés par la copie de l'objet racine uniquement.

[20] Cfr. la règle d'extensibilité ci-dessous.

système comme la distinction entre deux type de références différentes ("is-part-of" et "general").

3.2.2 Identité de l'objet.

L'identité d'un objet existe depuis longtemps dans les langages de programmation. Le concept est plus récent en ce qui concerne les bases de données[21]. L'idée est la suivante : dans un modèle avec une identité d'objet, un objet a une existence indépendante de sa valeur. Donc, deux notions équivalentes de l'objet existent :

- Deux objets peuvent être identiques (ils sont le même objet).
- Deux objets peuvent être égaux (ils ont la même valeur).

Cela a deux implications : d'une part on a le partage d'objets (object sharing) et de l'autre, les mises à jour d'objets (object updates).

Partage d'objets : dans un modèle basé sur l'identité, deux objets peuvent partager un composant. Donc, la représentation graphique d'un objet complexe est un graphe, alors qu'il est limité à un arbre dans un système sans identité d'objets. Prenons l'exemple suivant : une personne a un nom, un âge et un ensemble d'enfants. Supposons que *Peter* et *Susan* ont tous les deux un enfant de quinze ans du nom de *John*. Dans le monde réel, deux situations peuvent surgir : *Susan* et *Peter* sont les parents du même enfant ou il s'agit de deux enfants différents. Dans un système sans identité, *Peter* est représenté par :

(Peter, 40, {(John, 15, { })})

et *Susan* est représentée par :

(Susan, 41, {(John, 15, { })})

Donc, il n'y a aucun moyen d'exprimer que *Susan* et *Peter* sont les parents du même enfant. Dans un modèle basé sur l'identité, ces deux structures peuvent partager la partie commune (John, 15, { }) ou non et donc, exprimer une situation ou l'autre.

Mises à jour d'objets : supposons que *Peter* et *Susan* sont effectivement les parents du même enfant. Dans ce cas, toutes les mises à jour concernant le fils de *Susan* seront appliquées sur l'objet *John* et donc aussi sur le fils de *Peter*. Dans

---

[21] S. Khoshafian and G. Copeland, "Object identity". Proceedings of the 1st ACM OOPSLA conference, Portland, Oregon, September 1986.

un système basé sur la valeur des objets, les deux composants doivent être mis à jour séparément. L'identité d'objets est également une primitive de manipulation de données puissante.

Supporter l'identité d'objets implique qu'il faudra offrir des opérations telles que l'assignation, la copie d'objets[22] et des tests concernant l'identité d'un objet et l'égalité d'objets[23].

Bien sûr, il est possible de simuler l'identité d'objet dans un système basé sur la valeur des objets en introduisant des identifiants explicites d'objets. Cependant, cette approche alourdit la tâche de l'utilisateur qui devra s'assurer de l'unicité de l'identifiant des objets et de l'intégrité référentielle.

Il est à noter que les modèles basés sur l'identité des objets sont la norme dans les langages de programmations impératifs : chaque objet manipulé dans un programme possède une identité et peut être mis à jour. Cette identité vient soit du nom d'une variable, soit de la localisation physique dans la mémoire. Ce concept est assez nouveau dans les systèmes purement relationnels, où les relations sont basées sur les valeurs.

3.2.3 Encapsulation

L'idée de l'encapsulation vient, premièrement, du besoin de distinguer proprement la spécification de l'implémentation d'une opération et, deuxièmement, du besoin de la modularité. La modularité est nécessaire pour structurer des applications complexes et implémentées par une équipe de programmeurs. C'est aussi nécessaire comme outil de protection et d'autorisation d'accès aux objets.

Il y a deux vues de l'encapsulation :

- La vue du langage de programmation (qui est la vue originale puisque ce concept en est issu).
- L'adaptation de cette vue aux bases de données.

L'idée de l'encapsulation **dans les langages de programmation** vient des types de données abstraits. Dans cette vue, un objet possède une partie interface et une partie implémentation. La partie interface est la spécification de l'ensemble des opérations qui peuvent être appliquées sur l'objet. C'est la seule partie visible de l'objet. La partie implémentation possède une partie données et une partie

[22] Aussi bien la "deep copy" que la "shallow copy".

[23] Aussi bien la "deep equality" que la "shallow equality".

procédurale. La partie données est la représentation de l'état de l'objet, la partie procédurale décrit, dans un certain langage de programmation, l'implémentation de chaque opération.

```
class EMPLOYE {
public:
      // partie interface
      augmenter_salaire( );
      ...
private:

      // partie données de l'implémentation
      string nom;
      Int âge;
      int salaire
};

EMPLOYE::augmenter_salaire( )
{
      //  partie procédurale de l'implémentation
      //  opération augmenter_salaire
      //  de la classe d'objet EMPLOYE
      ...
}
```

Fig. 3.1 : Exemple d'encapsulation avec le langage de programmation C++

La **transposition du principe aux bases de données** est qu'un objet encapsule à la fois les données et les programmes. Dans le monde des bases de données, le fait que la partie structurelle du type fasse partie ou non de l'interface n'est pas clair, tandis que dans le monde des langages de programmation, la structure des données fait clairement partie de l'implémentation, en non de l'interface.

Considérons par exemple un employé. Dans un système relationnel, un employé est représenté par un n-uplet. Il est requis en utilisant un langage relationnel. Plus tard, un programmeur écrit des programmes pour mettre à jour ce "record", par exemple pour augmenter le salaire de cet employé ou le licencier. Ils sont généralement écrits, soit dans un langage de programmation impératif avec des instructions DML[24] imbriquées, soit dans un langage de quatrième génération, et sont mémorisés dans un système de fichiers traditionnel, et pas dans la base de données. Donc, dans cette approche, il existe une forte distinction entre le programme et les données, et entre le langage de programmation et le langage de requête.

Dans un système orienté objet, on définit l'employé comme un objet qui possède une partie données[25], et une partie opération qui contient les opérations

[24] Data Management Language.

[25] Probablement très similaire au "record" qui aurait été défini pour la modèle relationnel.

`Licencier` et `Augmenter_salaire` ainsi que d'autres opérations pour accéder aux données d'un employé. Lorsque l'on a enregistré un ensemble d'employés, les opérations et les données sont enregistrées dans la base de données.

Donc, nous avons un seul modèle pour les données et les opérations, et les informations peuvent être cachées. Aucune opération, autre que celles spécifiées dans l'interface, ne peut être utilisée. Cette restriction est valable, aussi bien pour les opérations de mise à jour que celles de suppression.

L'encapsulation nous fournit une forme "d'indépendance logique des données" : on peut changer l'implémentation d'un type sans modifier un seul des programmes utilisant ce type. Donc, les applications sont protégées des changements d'implémentation dans les couches inférieures du système.

Une encapsulation propre est obtenue quand uniquement les opérations sont visibles, et quand les données et l'implémentation des opérations sont cachées dans les objets.

Cependant, il y a des cas où l'encapsulation n'est pas nécessaire. L'utilisation du système peut être fortement simplifiée s'il permet de violer l'encapsulation sous certaines conditions[26]. Un mécanisme d'encapsulation doit être fourni par un SGBDOO, mais il existe des cas où son application n'est pas appropriée.

3.2.4 Types et classes.

Il existe deux catégories principales de systèmes orientés objets : celle qui supporte la notion de classe et celle qui supporte la notion de type. Dans la première catégorie, il y a les systèmes tels que Smalltalk[27], Gemstone et plus généralement tous les systèmes de la famille Smalltalk ou du Lisp. Dans la seconde catégorie, il y a les systèmes comme C++ [C++4] et Simula[28].

Un **type**, dans un système orienté objets, résume les spécificités communes d'un ensemble d'objets ayant les même caractéristiques. Cela correspond au concept de type abstrait de donnée. Il possède deux facettes : l'interface et l'implémentation. Seul la partie interface est visible aux utilisateurs de ce type,

---

[26] C++ possède la notion d'amis de classe. Peuvent être déclarés comme amis d'une classe , des fonctions, une autre classe.

[27] A. Goldberg and D. Robson, "Smalltalk-80 : the language and it's implementation", Addison-Wesley, 1983.

[28] "Simula 67 Reference Manual"

l'implémentation de l'objet est vue uniquement par le concepteur du type. L'interface est constituée d'une liste d'opérations avec leurs signatures[29].

L'implémentation du type est constituée d'une partie donnée et d'une partie opération. Dans la partie donnée, on décrit la structure interne des données de l'objet. La structure de cette partie donnée peut être plus ou moins complexe selon la puissance du système. La partie opération est constituée des procédures qui implémentent les opérations de la partie interface.

Dans les langages de programmation, les types sont des outils pour augmenter la productivité des programmeurs, en leur assurant des programmes corrects. En forçant les utilisateurs à déclarer les types des variables et des expressions qu'ils manipulent, le système raisonne sur l'état correct des programmes basés sur ce typage des informations. Si le système de types est construit avec attention, le système peut effectuer le contrôle de type à la compilation ;s'il en est autrement, certains d'entre eux devront peut-être être différés durant la compilation. Donc, les types sont principalement utilisés à la compilation pour vérifier l'état correct des programmes. En général, dans les systèmes typés, un type n'est pas un citoyen de première classe et a un statut particulier et ne peut être modifié durant l'exécution.

La notion de **classe** est différente de celle de type. Sa spécification est identique à celle du type, mais elle est plus une notion d'exécution. Elle contient deux aspects : une fabrique d'objet (object factory) et un entrepôt d'objets (object warehouse). La fabrique d'objets peut être utilisées pour créer de nouveaux objets, en appliquant l'opération `New` sur la classe, ou en produisant un clone à partir d'un prototype d'objet représentatif de la classe. L'entrepôt d'objets attaché à une classe, est l'ensemble des objets qui sont des instances de cette classe. L'utilisateur peut manipuler l'entrepôt en appliquant des opérations sur tous les éléments de la classe. Les classes ne sont pas utilisées pour vérifier qu'un programme est correct, mais plutôt pour créer et manipuler des objets. Dans la plupart des systèmes qui utilisent le mécanisme de classe, les classes sont les citoyens de première classe et, en tant que tels, peuvent être manipulés à l'exécution, c'est à dire mises à jours et passées en paramètre. Dans la plupart des cas, pendant qu'il fournit aux systèmes une plus grande flexibilité et une plus grande uniformité, cela rend le contrôle de type à la compilation impossible.

---

[29] C'est à dire le type des paramètres d'entrée et le type du résultat.

Bien sûr, il existe de nombreuses similitudes entre les classes et les types, les noms ont été utilisés avec les deux significations ; les différences peuvent être subtiles dans certains systèmes.

Il ne semble pas intéressant de choisir une des deux approches, le choix entre les deux doit être laissé au concepteur du système. Nous demandons cependant que système offre une certaine forme de mécanisme de structuration des données, qu'il s'agisse des classes ou des types.

3.2.5 Hiérarchie de classes ou de types.

L'héritage possède deux avantages : c'est un outil de modélisation puissant, car il donne une description précise et concise du monde réel et aide à la construction de spécifications et d'implémentations partagées dans les applications.

Un exemple va nous aider à illustrer l'intérêt d'avoir un système fournissant le mécanisme d'héritage. Supposons que nous ayons des Employés et des Etudiants. Chaque EMPLOYÉ possède un nom, un âge de plus de 18 ans et un salaire ; il ou elle peut mourir, se marier et être payé(e). Chaque ÉTUDIANT possède un âge, un nom et un ensemble de grades ; il ou elle peut mourir, se marier et avoir une bourse.

Dans un système relationnel, le concepteur de la base de données définit une relation pour EMPLOYÉ, une relation pour ETUDIANT et écrit le code des opérations `mourir`, `se_marier` et `être_payé` pour la relation EMPLOYÉ, et écrit le code de opérations `mourir`, `se_marier` et `avoir_une_bourse` pour la relation ETUDIANT. Le programmeur de l'application doit donc écrire six programmes.

Dans un système orienté objets, en utilisant la propriété d'héritage, on remarque que, EMPLOYE et ETUDIANT sont des PERSONNES ; ils ont donc à la fois quelque chose en commun et des traits spécifiques. Introduisons le type PERSONNE, qui possède les attributs nom et âge et les opérations `mourir` et `se_marier`. Nous déclarons ensuite que EMPLOYE est un type particulier de PERSONNE qui hérite des attributs et des opérations, et possède un attribut spécifique salaire. Il possède également une opération spécifique `être_payé`. De la même manière, déclarons qu'un ETUDIANT est un type particulier de PERSONNE avec un attribut spécifique grades et une opération spécifique `avoir_une_bourse`. Dans ce cas, nous avons une description plus concise et mieux structurée du schéma et nous avons écrit seulement quatre programmes. L'héritage aide à la réutilisabilité car chaque programme est au niveau où le plus grand nombre d'objets peut l'utiliser.

Il existe au moins quatre types d'héritage [OODB1] :

- L'héritage de substitution
- L'héritage d'inclusion
- L'héritage de contrainte
- L'héritage de spécialisation

Dans l'**héritage de substitution**, on dit qu'un type T hérite d'un type T' si on sait appliquer plus d'opérations sur les objets de type T que sur les objets de type T'. Donc, partout où on peut avoir un objet de type T', on peut lui substituer un objet de type T. Cette forme d'héritage est basée sur le comportement et non sur la valeur.

L'**héritage d'inclusion** correspond à la notion de classification. Il dit que T est un sous-type de T' si tout objet du type T est aussi un objet du type T'. Ce type d'héritage est basé sur la structure et pas sur les opérations. Prenons comme exemple l'objet CARRE avec les méthodes `get`, `set(size)` et l'objet CARRE_PLEIN avec les méthodes `get`, `set(size)` et `fill(color)`.

L'**héritage de contrainte** est un cas particulier de l'héritage d'inclusion. Un type T est un sous-type d'un type T' s'il est constitué de tous les objets du type T qui satisfont à une contrainte donnée. Exemple : Un ADOLESCENT est une sous-type de PERSONNE. ADOLESCENT ne possède pas plus de champs ou d'opération que PERSONNE mais il doit obéir à une contrainte spécifique (son âge est compris entre 13 et 19 ans).

Dans l'**héritage de spécialisation**, un type T est un sous-type du type T' si les objets du type T sont des objets de type T' qui contiennent en plus des informations spécifiques. Exemple : considérons les objets PERSONNE et EMPLOYE où les informations concernant EMPLOYE sont celles de PERSONNE plus certains champs spécifiques.

3.2.6 Surcharge et liaison dynamique.

En contraste avec l'exemple précédent, il existe des cas où l'on veut que plusieurs opérations différentes possèdent le même nom. Considérons par exemple l'opération `afficher` : elle prend un objet en entrée et l'affiche à l'écran. En fonction du type de l'objet en entrée, nous voudrions utiliser des mécanismes d'affichages différents. Si l'objet est un DESSIN, nous voulons qu'il apparaisse à l'écran. Si l'objet est une PERSONNE, nous voulons qu'une représenation de son n-uplet apparaisse à l'écran. Enfin, si l'objet est un GRAPHE, nous voulons sa

représentation graphique affichée à l'écran. Considérons maintenant le problème d'afficher un "set" d'objets dont on ne connaît pas les types à la compilation.

Dans une application utilisant un système conventionnel, nous avons trois opérations `afficher _ personne`, `afficher _ dessin`, `afficher _ graphe`. Le programmeur va tester le type de chaque objet du "set" pour utiliser l'opération d'affichage correspondante. Ceci à pour conséquence que le programmeur devra avoir connaissance de tous les types possibles des objets du "set" et leur opération d'affichage associée.

```
for x in X do
begin
    case of type(x)
        personne: afficher_personne (x);
        dessin: afficher_dessin (x);
        graphe: afficher_graphe (x);
    end
end
```

Dans un système orienté objets, nous définissons l'opération d'affichage au niveau du type de l'objet. L'opération d'affichage a un seul nom et peut être utilisée indifféremment par les personnes, les dessins ou les graphes. Par contre, nous allons redéfinir l'implémentation de l'opération pour chaque type d'objets. Cette redéfinition est appelée la surcharge. Pour afficher un ensemble d'objets de types différents, on applique tout simplement l'opération `afficher` sur chacun des objets. On laisse donc le soin au système de choisir l'opération d'affichage appropriée à l'exécution .

```
for x in X do  afficher (x) ;
```

L'avantage est ici d'une nature différente : les implémenteurs écriront toujours le même nombre de programmes. Par contre, le programmeur de l'application ne doit pas s'occuper des trois programmes différents. De plus, le code est plus simple puisqu'il n'y a plus d'instruction de test concernant les types. Enfin, le code est devenu plus maintenable : lorqu'un nouveau type est introduit, il suffira de définir l'opération d'affichage pour ce dernier, l'application pourra fonctionner sans problème ,et cela, sans modification.

Pour fournir cette fonctionnalité, le système n'est pas capable de faire le lien entre le nom de l'opération et les programmes à la compilation. C'est pourquoi les noms des opérations doivent être résolus (c'est à dire traduits en adresses) à l'exécution. Cette traduction différée est appelée liaison dynamique.

Il est à noter que, même si la liaison dynamique à l'exécution rend le contrôle de type plus difficile[30], elle ne l'exclut pas totalement.

3.2.7 Complétude des traitements.

Du point de vue d'un langage de programmation, cette propriété est évidente : cela signifie tout simplement que l'on peut exprimer n'importe quelle fonction exécutable en utilisant le DML du système de gestion de base de données. Du point de vue des bases de données, c'est une nouveauté : SQL, par exemple, n'est pas complet.

Nous ne demandons pas que les concepteurs de SGBDOO créent de nouveaux langages de programmation : la complétude des traitements peut être obtenue à partir de langages de programmation existants.

Tout cela est différent d'être capable d'accéder à toutes les ressources du système à partir de ce langage. C'est pourquoi, même si le langage est complet du point de vue des traitements, il ne pourra pas nécessairement exprimer une application complète. C'est cependant plus puissant qu'une base de données qui ne se borne qu'à retirer, enregistrer des données, et exécuter des traitements sur des valeurs atomiques.

3.2.8.Extensibilité.

La base de donnée existe avec un ensemble de types prédéfinis. Ces types peuvent être utilisés selon le bon vouloir des programmeurs pour écrire leurs applications. Cet ensemble de types doit pouvoir être extensible : il est possible de définir de nouveaux types et il n'existe aucune distinction d'usage entre les types prédéfinis du système et ceux définis par l'utilisateur. Bien sûr, il peut y avoir des différences dans la façon dont le système supporte les types prédéfinis du système et ceux définis par l'utilisateur, mais cela devrait rester invisible pour l'application et son concepteur.

3.2.9 Persistance.

C'est évident du point de vue des bases de données, mais c'est une nouveauté du point de vue des langages de programmation. La persistance est la possibilité pour le programmeur d'avoir ses données qui survivent à l'exécution d'un processus pour pouvoir éventuellement être réutilisées par un autre processus. La persistance devrait être orthogonale : chaque objet, indépendamment de son type,

[30] Elle est parfois impossible.

est autorisé à devenir persistant tel quel, sans traduction, transformation explicite. L'utilisateur ne devrait pas déplacer ou copier explicitement des données pour les rendre persistantes.

3.2.10 Gestion de mémoire secondaire

La gestion de mémoire secondaire est un composant classique des SGBD. C'est généralement supporté par un ensemble de mécanismes. Cela comprend la gestion des index, le "clustering" des données, le "buffering" des données et l'optimisation des requêtes.

Ces derniers ne sont pas visibles pour l'utilisateur, ce sont simplement des composants pour la performance. Cependant, ils sont si critiques en terme de performance que leur absence empêcherait le système d'effectuer certaines tâches[31]. Ce qui est important, c'est qu'ils soient invisibles. Le programmeur ne devrait pas avoir à écrire du code pour gérer les index, ni à allouer de la mémoire sur disque ou déplacer des données entre le disque et la mémoire centrale. Il devrait exister une indépendance entre le niveau physique et logique du système.

3.2.11 Concurrence.

En respect avec la gestion de plusieurs utilisateurs concurrents qui travaillent sur le système, ce dernier devrait offrir le même niveau de service que celui offert par les systèmes de base de données actuels. C'est pour cela qu'il devra assurer une coexistence harmonieuse entre les utilisateurs qui travaillent simultanément sur la base de donnée. Le système devra supporter la notion standard d'atomicité d'une séquence d'opérations et de partage contrôlé.

3.2.12 Reconstruction.

Ici aussi, le système devrait fournir le même niveau de service que les bases de données actuelles. En cas de défaillance hardware ou software, le système devra revenir lui-même dans un état cohérent des données. Les défaillances hardware comprennent à la fois les défaillances du processeur et des disques.

3.2.13 Facilités de requête adéquates.

Le principal problème est de fournir la fonctionnalité d'un langage de requête *ad hoc*. Nous ne demandons pas que ce soit fait sous la forme d'un langage de requête, mais uniquement que ce service soit fourni. Par exemple, un "browser"

---

[31] Simplement parce que cela prendrait trop de temps.

graphique est suffisant pour remplir cette fonctionnalité. Le service consiste à permettre à l'utilisateur de demander des requêtes simples sur la base de données. L'étalon de mesure est bien sûr le système relationnel ; le test consiste à prendre un certain nombre de requêtes relationnelles représentatives, et vérifier si elles peuvent être effectuées avec la même quantité de travail. Il est à noter que cette facilité pourrait être supportée par le DML ou un sous-ensemble de celui-ci.

Ces facilités de requête devraient satisfaire les trois critères suivants :

- Elles devraient être de haut niveau, c'est à dire que l'on devrait être capable d'exprimer, en quelques mots ou quelques mouvements de souris, des requêtes non triviales de manière concise.
- Elles devraient être efficaces : la formulation des requêtes devrait mener elle-même vers une certaine forme d'optimisation de requête.
- Elles devraient être indépendantes de l'application, c'est à dire qu'elles devraient fonctionner pour n'importe quelle base de donnée. Ce dernier critère élimine les facilités de requête spécifique qui dépendent de l'application, ou qui demandent d'écrire des opérations supplémentaires sur chaque type défini par l'utilisateur.

3.2.14 Conclusion.

Nous avons terminé de citer les caractéristiques obligatoires. La distinction entre les bases de données traditionnelles et les base de données orientées objets devrait être claire.

Les systèmes relationnels ne satisfont pas aux règles de 1 à 8. Les bases de données CODASYL satisfont partiellement aux règles 1 et 2. Certaines personnes disent que les SGBDOO ne sont rien d'autre que des systèmes CODASYL. Notons simplement que :

- Les systèmes CODASYL ne satisfont pas pleinement à ces deux règles. Les constructeurs d'objets ne sont pas orthogonaux, l'identité d'objet n'est pas traitée uniformément puisque la connectivité des relations est limitée à 1-n.
- Ils ne satisfont pas aux règles 3, 5, 6, 8 et 13.

3.3 Les caractéristiques optionnelles

Nous allons passer en revue toutes les caractéristiques qui pourraient clairement améliorer le système, mais qui ne sont pas obligatoires pour en faire un SGBDOO.

Certaines de ces caractéristiques sont d'une nature orientée objets. Elles sont inclues dans cette catégorie car, même si elles rendent le systèmes "plus" orienté objets, elles n'appartiennent pas au noyau des exigences.

D'autres caractéristiques sont simplement liées aux bases de données. Ces caractéristiques améliorent généralement la fonctionnalité des SGBD, mais elles ne font pas partie du noyau des exigences des SGBD et n'ont rien à voir avec l'aspect orienté objets.

3.3.1 Héritage multiple.

Le fait de fournir l'héritage multiple ou non est une option. Puisque la communauté de l'orientation objet ne s'est pas encore mis d'accord concernant l'héritage multiple, nous considérons qu'il est optionnel.

3.3.2 Contrôle de type et inférence de type.

Le degré de contrôle de type que le système va utiliser à la compilation est variable, mais le plus élevé est le meilleur. La situation optimale est celle où un programme qui a été accepté par le compilateur ne peut pas produire d'erreurs à l'exécution. Le nombre d'inférence de type est aussi laissé à la discrétion du concepteur du système : la situation idéale est celle dans laquelle seulement les types de base doivent être déclarés, le système inférant les types temporaires.

3.3.3 Distribution.

Il devrait être clair que cette caractéristique est orthogonale à la nature orienté objets du système. Le SGBD pourra être distribué ou non.

3.4 Les caractéristiques ouvertes.

Chaque système satisfaisant aux règles 1 à 13 peuvent avoir la dénomination de système de gestion de base de données orientée objets. Durant la conception de tels systèmes, il reste de nombreux choix à faire. C'est le degré de liberté laissé

au concepteur des SGBDOO. Ces caractéristiques sont différentes des caractéristiques obligatoires dans le sens où aucun consensus n'a encore abouti dans la communauté scientifique les concernant. Elles diffèrent également des caractéristiques optionnelles dans le sens où l'on ne sait pas laquelle des alternatives est plus ou moins orientée objets.

3.4.1 Le paradigme de programmation.

Nous ne voyons aucune raison d'imposer un paradigme de programmation plutôt qu'un autre : le style de programmation logique, fonctionnel ou impératif pourraient être tous choisis comme paradigme de programmation. Une solution est d'avoir un système indépendant du style de programmation qui supporte de multiples paradigmes.

3.4.2 Représentation du système.

La représentation du système est définie par l'ensemble des types atomiques et l'ensemble des constructeurs. Même si on a donné un ensemble minimal de types atomiques et de constructeurs[32] disponibles pour décrire la représentation des objets, ils peuvent être étendus de différentes manières.

3.4.3 Types du système.

La seule facilité de formation d'objet que nous exigeons est l'encapsulation. Il peut y avoir d'autres formateurs de types comme les types génériques ou des générateurs de types. Exemple : un set [T], ou T est un type arbitraire.

3.4.4 Uniformité.

Il existe un débat important concernant le degré d'uniformité auquel on pourrait s'attendre de la part de tels systèmes. Est-ce qu'un type est un objet ? Est-ce qu'une méthode est un objet ? Est-ce que ces trois notions doivent être traitées de manières différentes ? On peut voir ce problème sous trois niveaux différents :

- Le niveau de l'implémentation.
- Le niveau du langage de programmation.
- Le niveau de l'interface.

---

[32] Ce sont les types élémentaires des langages de programmation et les constructeurs de "set", de n-uplets et de listes.

Au **niveau de l'implémentation**, on doit décider si les informations sur les types doivent être enregistrées comme des objets, ou si un système *ad hoc* doit être implémenté. C'est le même problème rencontré par les concepteurs de bases de données relationnelles lorqu'ils doivent décider d'enregistrer le schéma. La décision devra être prise par rapport aux performances et les exigences de l'implémentation. Quelque soit la décision prise, elle est indépendante de celle prise au niveau suivant.

Au **niveau du langage de programmation**, la question est la suivante : est-ce que les types sont des citoyens de première classe dans la sémantique du langage ? La plus grande partie du débat est concentré sur cette question. Il y a probablement plusieurs styles d'uniformité (syntaxique ou sémantique).

Enfin, au **niveau de l'interface**, une décision indépendante des autres doit être faite. On voudra, d'un côté, présenter une vue uniforme des types, des objets et des méthodes à l'utilisateur, même si dans le sémantique du langage de programmation, il existe des notions de différentes natures. De l'autre, on pourrait les présenter comme des entités différentes, même si le langage de programmation les considère comme équivalents.

CHAPITRE IV : Proposition de représentation des données.

4.1 Introduction :

Nous avons présenté au chapitre 2 un bref aperçu des méthodologies orientées objets utilisées dans le cadre de l'analyse et de la conception d'un système informatique. Nous n'allons cependant pas utiliser l'une de ces méthodologies dans la conception de base de données orientées objets. La raison essentielle est que nous sommes surtout intéressés par la "data view". Nous écarterons les autres aspects.

Nous allons tenter, dans ce chapitre, de proposer une représentation des données, qui soit stable et complète. Pour que notre modèle soit complet, il devra être capable de représenter tous les concepts dont nous aurons besoin :

- Les classes d'objets et leurs instances
- Les relations que les classes d'objets ont entre elles
- Les attributs des classes d'objets
- Les identifiants des classes d'objets
- Les méthodes des classes d'objets
- L'héritage

Paradoxalement, nous n'insisterons pas sur certaines caractéristiques des systèmes orientés objets. Nous estimons que ces concepts seront représentés de manière implicite. Cela concerne :

- La surcharge et l'exécution dynamique
- L'encapsulation (vues externes et internes des objets)

Tous ces aspects qui resteront implicites, seront matérialisés dans l'implémentation de la base de données :

- C'est l'implémentation de l'héritage qui entraînera une éventuelle surcharge et/ou une éventuelle exécution dynamique.
- C'est l'implémentation des méthodes d'accès aux données des objets qui créera leur encapsulation.

Le modèle proposé est une extension du modèle ER classique [ER1].

4.2 Les concepts représentables :

4.2.1 Les classes d'objets et leurs instances.

Une classe d'objets représente une classe ou un type d'objets. Ces notions ont été présentées au chapitre 3. La classe d'objets contient à la fois la description des caractéristiques et des propriétés communes des instances de cette classe, mais également l'ensemble de ces instances. Cela semblera d'autant plus logique lorsque l'on sait que l'on modélise une base de données.

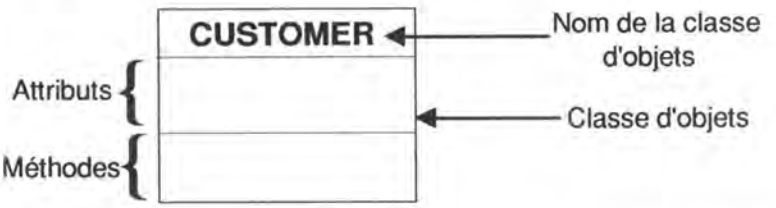

Fig. 4.1 : Représentation graphique de la classe d'objets CUSTOMER.

Par rapport à la représentation graphique du concept de "type d'entité" du modèle ER, notre classe d'objets possède une partition supplémentaire où les méthodes de la classe sont contenues.

4.2.2 Les relations entre les classes d'objets.

Les relations enrichissent la sémantique de la base de données. Dans la figure 4.2 ci-dessous, la relation représente :

- le fait qu'un client est associé à aucune, une ou plusieurs commandes
- le fait qu'une commande est toujours associée à un et un seul client

Les connectivités minimum et maximum des rôles sont exprimées tout simplement par un couple d'entier, N représentant un nombre quelconque.

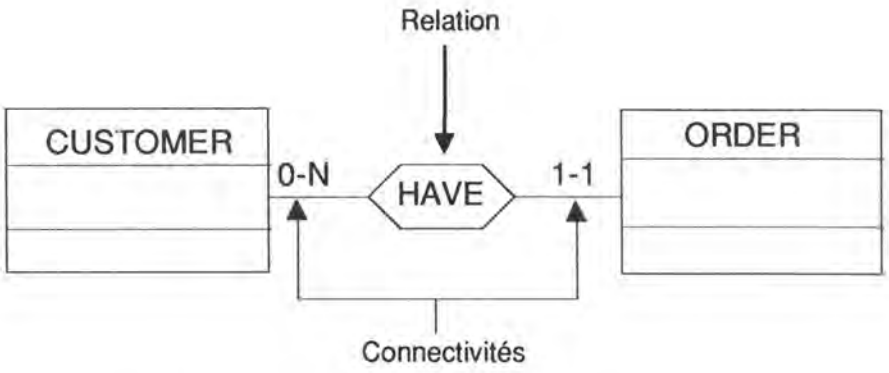

Fig. 4.2 : Représentation graphique d'une relation.

Il est évident que dans le cadre d'une conception orientée objets, le degré[33] de ces relations ne pourra être supérieur à 2. Les relations ne possèdent pas d'attributs.

4.2.3 Les attributs d'une classe d'objets.

Les attributs d'une classe d'objets font partie des caractéristiques communes des instances de la classe d'objets. Ils possèdent un nom et un type.

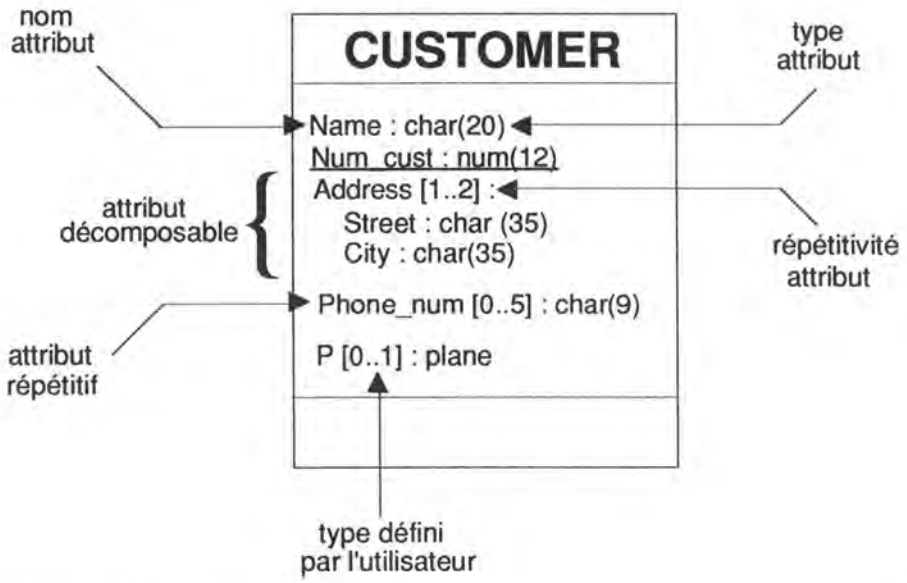

Fig. 4.3 : Représentation graphique des attributs d'une classe d'objets.

Pour créer des objets complexes décrits au chapitre 3, les attributs doivent pouvoir être d'un type quelconque, autre que ceux des types de bases du système (entiers, caractères, etc.) :

[33] Le degré d'une relation représentant le nombre de classes d'objets participants à cette relation.

- **types décomposables :** un client peut posséder une adresse, elle même décomposée en une série d'attributs : la rue et la localité.
- **types répétitifs :** un client peut posséder une ou deux adresses.
- **types définis par l'utilisateur** : un client peut posséder une voiture, voiture étant une classe d'objets distincte. Les types définis par l'utilisateur constituent des classes d'objets à part entière.

4.2.4 Les identifiants des classes d'objets.

Un identifiant d'une classe d'objet est un groupe d'attributs et/ou de rôles tel qu'à chaque combinaison de valeurs prises par ce groupe correspond au plus un objet de cette classe. Une classe d'objets peut être doté de plus d'un identifiant. Lorsqu'il n'y a pas d'ambiguïté, on soulignera les éléments identifiants d'une classe d'objets. Dans la figure 4.3, l'attribut `Num_cust` est l'identifiant de la classe d'objets `customer`. Dans la figure ci-dessous, une commande peut être identifiée par le client concerné et par la date de la commande.

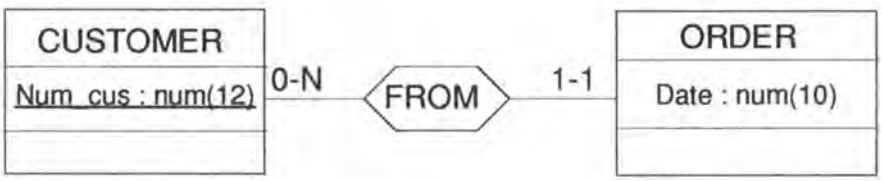

Fig. 4.4 : Illustration du concept d'identifiant.

4.2.5 Les méthodes des classes d'objets.

Les méthodes associées à une classe d'objets sont indiquées par leur nom.

Les méthodes peuvent être classées en trois types :

- **Les méthodes "d'accès internes"** : celles qui sont utilisées pour accéder ou modifier la valeur des attributs des objets de la classe.
- **Les méthodes "d'accès externes"** : celles qui sont utilisées pour accéder à des objets d'une autre classe d'objet, cette dernière étant associée via une relation à la classe d'objets en question.
- **Les méthodes "définies par l'utilisateur"** pour des traitements plus complexes sur la base de données qui, dans leur implémentation, utiliseront probablement des méthodes des deux types précédents.

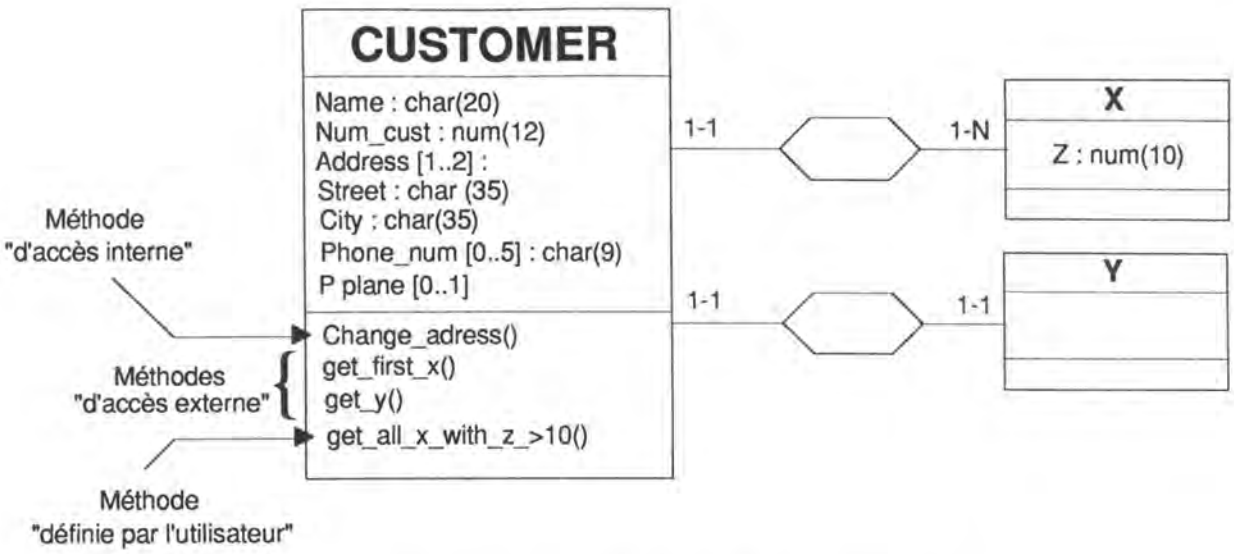

Fig. 4.5 : Représentation des méthodes de classes d'objets.

Remarque : nous considérons que le(s) constructeur(s), le destructeur et les autres méthodes destinées à la gestion même des objets d'une classe sont présents de manière implicite.

4.2.6 L'héritage.

Une classe, appelée super-classe, peut avoir plusieurs sous-classes. La super-classe est appelée la classe générique et les sous-classes les classes spécifiques. Un cluster est un groupe de classes spécifiques. Il s'agit en quelque sorte d'un concept "relais" entre la classe générique et les classes spécifiques. La "relation d'héritage" entre un type générique A et un des ses types spécifiques B peut se lire : tout objet de la classe B est un (IS-A) objet de la classe A.

Il y a couverture entre différents types spécifiques d'un cluster si et seulement si l'union des populations des classes spécifiques est égale à celle de la classe générique. Dans la figure ci-dessous, le cluster est couvrant, ce qui signifie que tout client est, soit un client public, soit un client privé, soit un ... .[34]

[34] La couverture est notée d'un C à l'intérieur du triangle représentant un cluster.

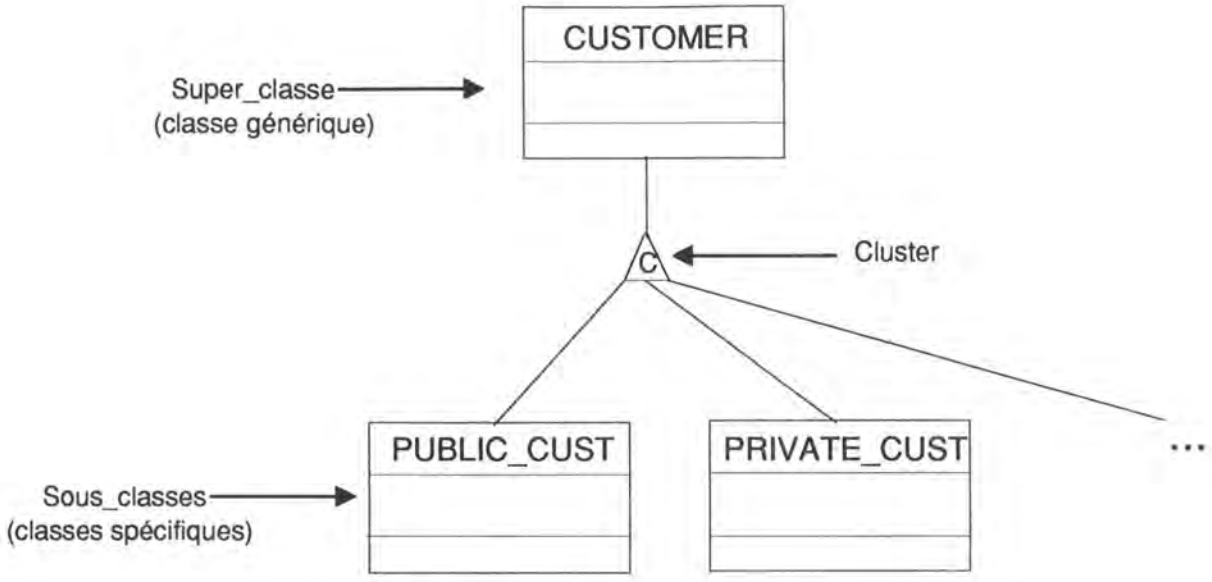

Fig. 4.6 : Représentation graphique de l'héritage.

Une classe peut posséder plusieurs clusters. Une classe d'objets peut être à la fois la classe générique d'un cluster et une des classes spécifiques d'un autre cluster.

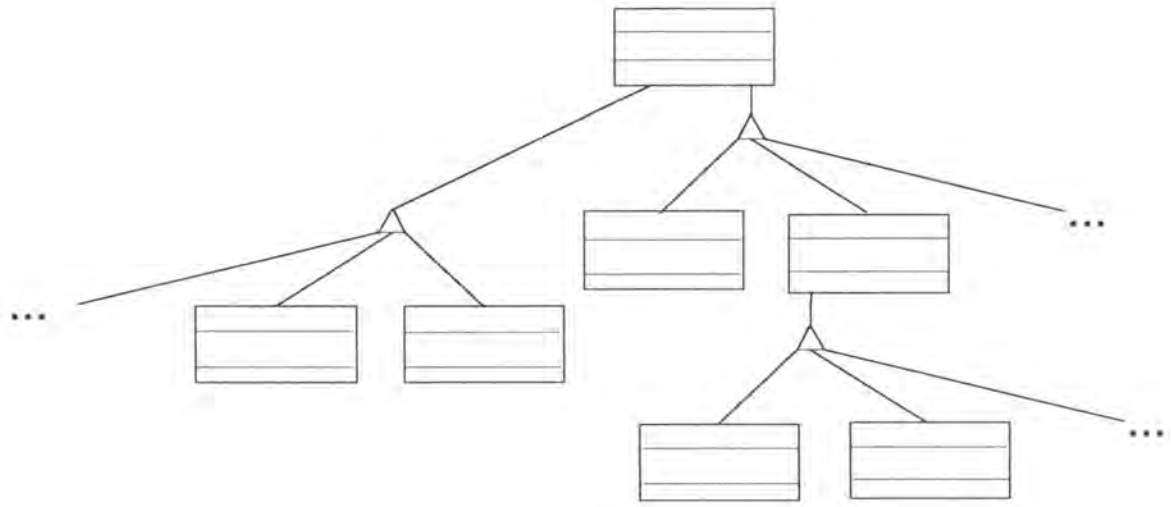

Fig. 4.6 : Illustration.

4.3 Conclusion.

Nous avons essayé que cette méthode soit la plus rigoureuse et la plus compréhensible possible. Le peu de concepts utilisés et la simplicité de notation va dans ce sens. De plus, puisque notre méthode est une extension du modèle ER, son utilisation n'en sera que plus aisée pour les personnes déjà aguerries à la conception de base de données.

Le choix de notre proposition n'est pas un hasard. Si nous proposons au concepteur de base de données orientée objets d'utiliser ce modèle, c'est qu'elle nous permettra une transposition aisée du schéma obtenu au modèle générique utilisé par l'atelier logiciel TRAMIS dont il sera question dans le chapitre suivant.

Nous allons dorénavant utiliser l'exemple classique de la gestion des commandes d'un client. En utilisant la méthode proposée, nous pouvons le représenter comme ceci :

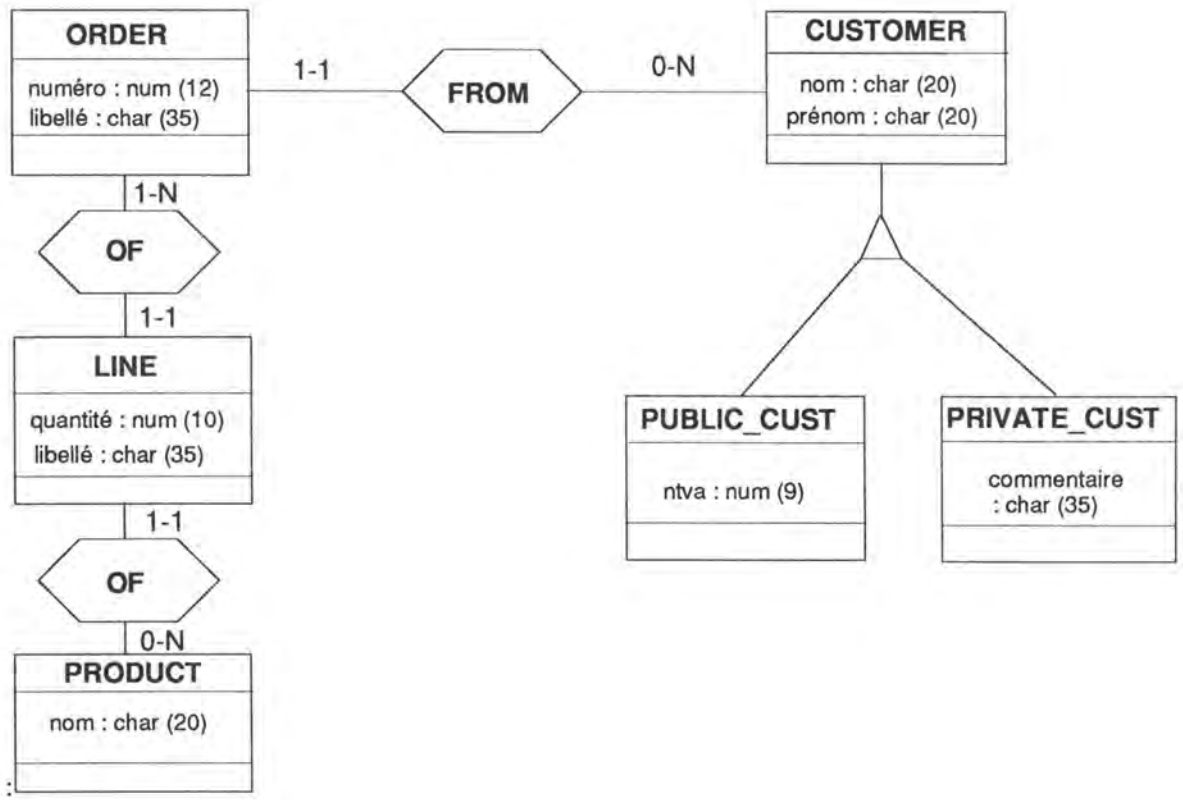

Fig. 4.7 : La représentation d'un exemple classique avec notre modèle.

CHAPITRE V : Le contexte.

5.1 Introduction.

Après avoir décrit notre base de données orientée objets conceptuelle à l'aide du modèle proposé dans le chapitre précédent, il va être temps de penser à la façon d'implémenter physiquement cette dernière. Nous allons utiliser pour cela l'atelier logiciel TRAMIS développé par les Facultés Notre-Dame de la Paix de Namur.

Ce chapitre va donc être consacré à la présentation de ce logiciel, tant du point de vue de ses principes méthodologiques, que du point de vue de son architecture. Nous verrons ensuite dans quelle mesure un module pourra être construit afin de supporter une conception orientée objet des bases de données.

5.2 TRAMIS, un atelier logiciel destiné à la conception de bases de données.

5.2.1 Introduction.

La plupart des ateliers de conception de bases de données fournissent quatre familles de fonctions à leurs utilisateurs :

- L'acquisition de spécifications conceptuelles
- La gestion de spécifications conceptuelles
- La validation de spécifications conceptuelles
- La génération de code exécutable

Ils se concentrent généralement sur l'étape de la spécification conceptuelle en laissant le problème de produire des schémas physiques efficaces pratiquement pas résolu. Ils sont basés sur des stratégies simples et rigides qui donnent l'illusion que la conception de base de données est un processus direct et déterministe une fois que le schéma conceptuel a été développé. Des stratégies plus sophistiquées qui permettraient la satisfaction d'exigences plus réalistes comme l'efficacité de temps et d'espace, la distribution, la modularité, la sécurité, les contraintes au niveau hardware, sont impossibles. Ces faiblesses poussent les concepteurs à des pratiques inacceptables :

- ils intègrent ces exigences dans le schéma conceptuel.
- ils modifient le DDL généré à l'aide d'un éditeur de texte.

TRAMIS est atelier logiciel expérimental qui propose une structure fonctionnelle différente qui peut supporter plusieurs stratégies de design selon la compétence du concepteur et des exigences à satisfaire par le produit final. Il est basé sur une approche transformationnelle sachant que la plupart des activités de conception de base de données sont basées sur des transformations de schémas formellement définies. Cette approche, qui jusqu'à présent, était surtout théorique, reçoit un accueil pratique de plus en plus grand.

5.2.2 Les principes méthodologiques de TRAMIS.

Actuellement, [TRAMIS1] les méthodes standards de conception de bases de données proposent une approche stricte par étapes selon laquelle le schéma conceptuel doit être construit à partir des exigences de l'utilisateur (analyse conceptuelle), ce schéma étant traduit en un schéma logique (conception logique) qui à son tour est transformé en un schéma physique (conception physique).

Les hypothèses de base, qui sont sous-jacentes à l'architecture de TRAMIS, concernent la façon dont les concepteurs se comportent réellement lorsqu'ils construisent des bases de données. Utilisent-ils des méthodes standards ou non ? La conception de programmes est une activité de résolution de problèmes qui peut être caractérisée par ce qui suit :

- Il existe un **ensemble de problèmes à résoudre limité.**
- Chaque problème est relié à des **exigences** spécifiques, fonctionnelles (orientées utilisateur) ou non-fonctionnelles (surtout techniques).
- La résolution d'un problème est effectuée par un **processus de design**, dont le but est de permettre à l'état actuel des spécifications, de **satisfaire aux exigences correspondantes** (normaliser un schéma, ou optimiser un schéma de base de données par rapport à l'espace disque).
- Un problème peut être **résolu indépendamment** des autres grâce à un ensemble limité de règles, de raisonnements et d'heuristiques[35].
- La plupart des processus de design peuvent être vus comme des **spécifications de transformations** qui produisent des produits en sortie à partir de produits en entrée.
- Une **méthode de design spécifique**, comme MERISE, peut être décrite comme un arrangement de processus de design, appliqués sur des produits spécifiques selon des ensembles spécifiques d'exigences.
- Dans la pratique, la façon dont les concepteurs construisent des systèmes de programmes ne suit pas toujours les méthodes standards, même lorsqu'une méthode est suggérée ou imposée ; de là viennent les notions de

[35] Cet ensemble n'est pas toujours petit.

méthode formelle et de méthode heuristique. Un concepteur qui suit aveuglément une méthode comme MERISE, utilise une méthode formelle ; un programmeur qui définit une base de données en encodant directement le schéma SQL utilise une méthode heuristique. Cependant, tous les deux sont supposés résoudre, explicitement ou implicitement, les mêmes problèmes associés à la future base de données. Ces observations sont très importantes quand on parle de "reverse engineering".

TRAMIS se propose de supporter, du moins en partie, ces comportements de design de résolution de problèmes grâce à quatre principes de base qui permettent de définir et d'appliquer à la fois des méthodes heuristiques et formelles dans la conception de bases de données. Les principes sont :

- Un modèle de spécification générique unique qui permet la définition d'une large variété de produits spécifiques.
- Des fonctions de transformation comme la plupart des ateliers d'aide à la conception de bases de données.
- Une architecture de boîte à outils permettant une indépendance maximum entre les fonctions.
- La définition de plusieurs modèles par la paramétrisation du modèle générique unique.

5.2.3 Le modèle générique.

Un schéma de base de données, quel que soit son état et son niveau d'abstraction, est décrit dans TRAMIS par un schéma entité-association. Les schémas conceptuels sont décrits dans un schéma unique, générique du modèle entité-association étendu ; il en est de même pour les schémas physiques. L'avantage d'utiliser un formalisme unique durant le cycle de vie d'une base de données est triple :

- Premièrement, cela diminue l'effort cognitif pour le concepteur, puisqu' un seul modèle doit être compris et maîtrisé au lieu de deux ou trois. Par exemple, plusieurs méthodes utilisent un modèle ER au niveau conceptuel, un modèle de type CODASYL ou Bachman au niveau logique et un modèle relationnel au niveau physique.

- Deuxièmement, c'est le support idéal pour un ensemble d'outils de transformations globales. En effet, les transformations peuvent être utilisées à n'importe quel niveau d'abstraction. Par exemple, la même transformation de schéma peut être utilisée pour normaliser un schéma conceptuel et pour optimiser un schéma CODASYL. De plus, des

transformations intra-modèle sont plus faciles à définir, à comprendre et même à implémenter que des transformations inter-modèles.

- Enfin, puisqu' aucun ensemble fixe de produits de design ne sont définis a priori, la définition de méthodes de conception standards ou non est faite assez facilement.

Nous allons illustrer brièvement les concepts[36] du modèle générique en supposant que le modèle entité-association est connu du lecteur.

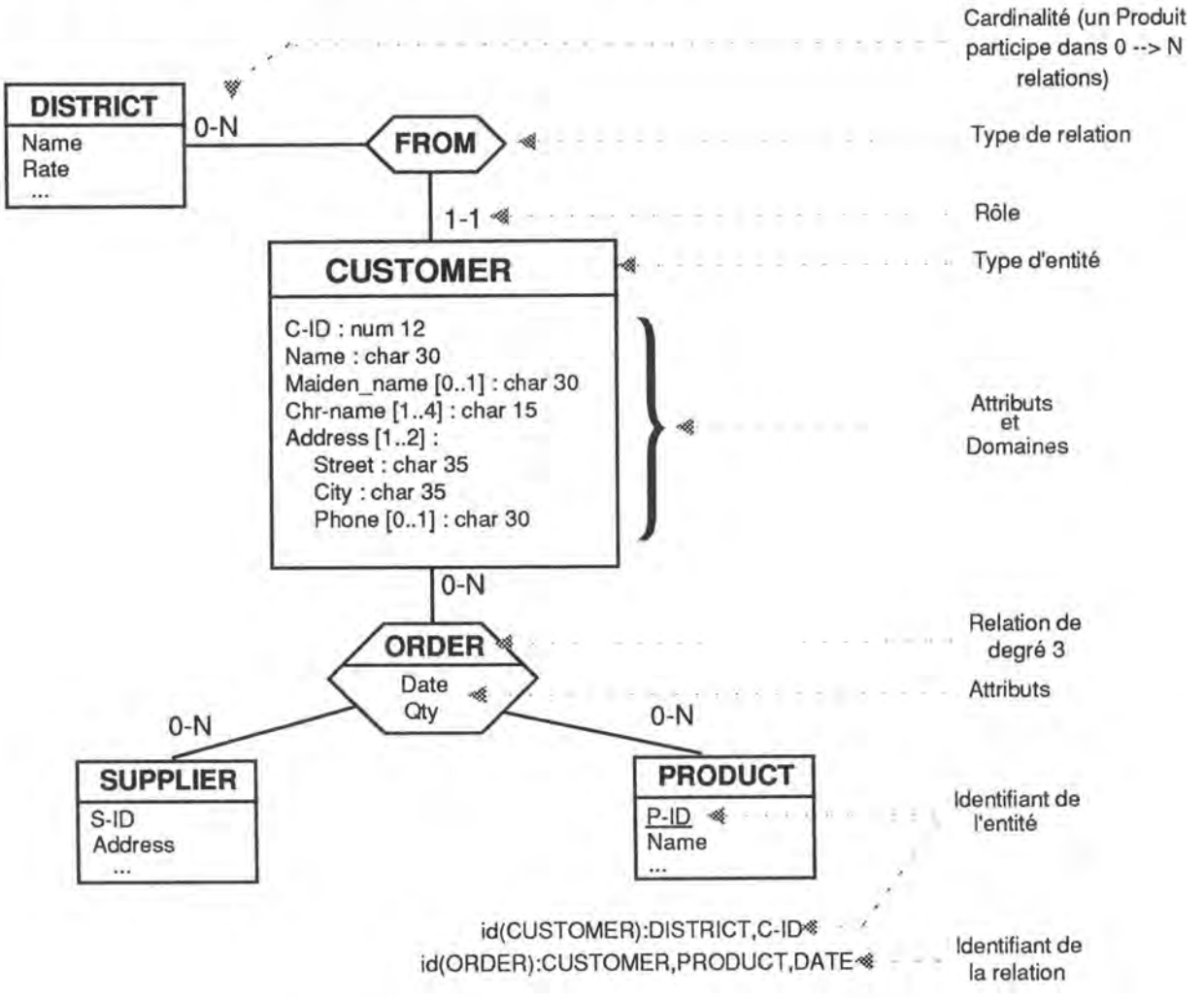

Fig. 5.1 : Illustration des concepts du modèle générique.

Nous allons insister quelque peu sur le concept de généralisation/spécialisation. La **généralisation** est un processus d'abstraction où un ou plusieurs ensembles

---

[36] Les concepts en question sont : les types d'entités, les types d'associations, les attributs, les rôles, les contraintes de cardinalités, les identifiants d'un type d'entité ou d'un type d'association.

Notons que les attributs peuvent être simple ou multivalués, optionnel ou obligatoire, atomique ou composé.

d'objets sont considérés à un niveau d'abstraction comme un ensemble le(s) contenant. Exemple : les ensembles EMPLOYE et OUVRIER peuvent être généralisés par l'ensemble PERSONNEL. La relation de généralisation est souvent rebaptisée en relation **IS-A**.

Supposons un type d'entité TE1. Le concepteur peut vouloir modéliser un type d'entité TEG plus général que TE1, c'est-à-dire : décrivant d'autres entités que celles de TE1. Nous venons d'opérer une généralisation sur TE1. TEG sera appelé type d'entité générique de TE1. A tout moment, la population de TE1 est incluse dans celle de TEG : $pop(TE1) \subseteq pop(TEG)$. A l'inverse, parmi les entités d'un type d'entité TE1, il est parfois possible d'en sélectionner certaines qui ont des propriétés communes ou plus précises et de les grouper pour constituer un type d'entité TES. TE1 a été spécialisé. TES est appelé type d'entité spécifique de TE1.

En parallèle de cette théorie de type ensembliste, nous pouvons y adjoindre des contraintes d'intégrité portant sur les relations IS-A créées en spécialisant un type d'entité générique ou en généralisant des types d'entités spécifiques. Nous dégagerons trois types de contraintes : la disjonction, la couverture et la partition.

Si on spécialise un type d'entité générique (TEG) en n types d'entités spécifiques (TES1, TES2, ..., TESn), on peut se retrouver face à une contrainte d'intégrité qui se vérifie à tout moment :

$$pop(TESi) \cap pop(TESj) = \varnothing \quad \forall\, i, j \in 1..n \text{ et } i \neq j$$

Exemple : soit le type d'entité PERSONNEL_ADMINISTRATIF et ses deux types d'entités spécifiques CADRE et SECRETAIRE. Il n'existe pas de cadre qui soit secrétaire et réciproquement. Il n'existe pas non plus un autre type d'entité spécifique PERSONNEL_ADMINISTRATIF disjoint des deux premiers. On peut donc spécifier que CADRE et SECRETAIRE forment un groupe d'entités disjoints de PERSONNEL_ADMINISTRATIF.

Graphiquement, nous représenterons la disjonction des types d'entités spécifiques TES1, TES2, ..., TESn du type d'entité TEG de la manière suivante :

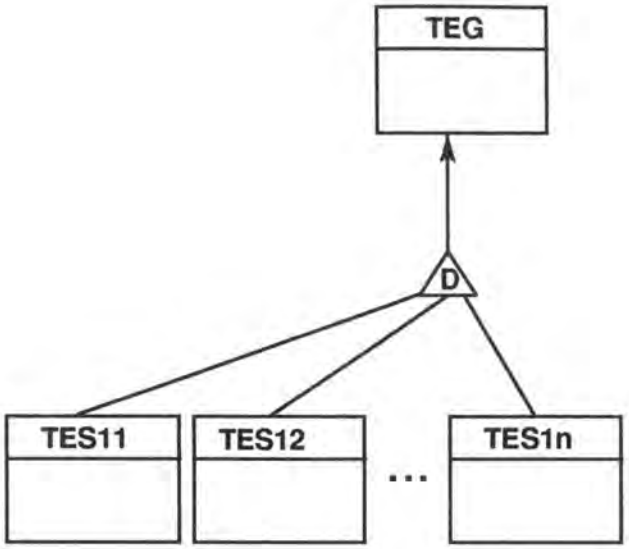

Fig. 5.2 : La disjonction au sein de la généralisation/spécialisation.

Si nous disposons de plusieurs types d'entités spécifiques (TES1, TES2, ..., TESn et n>1) et d'un type d'entité générique (TEG), nous dirons qu'il y a couverture si et seulement si, à tout moment,

$$\text{pop(TES1) U pop (TES2) U ... U (TESn)} = \text{pop (TEG)}$$

Une entité de TEG appartient au moins à un des types d'entités spécifiques et au plus à n.

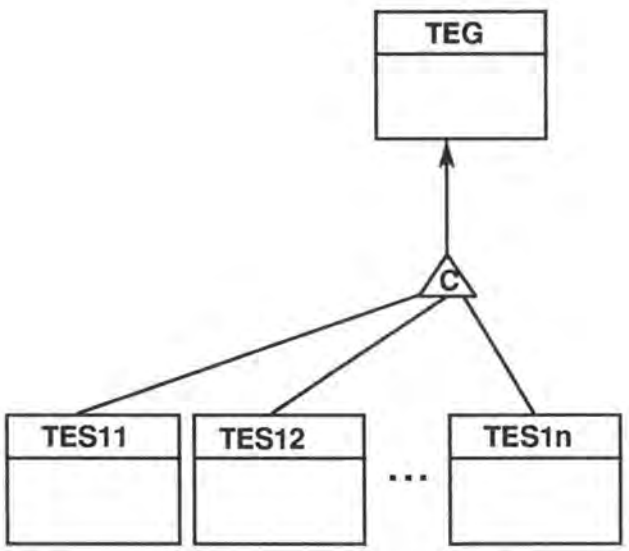

Fig. 5.3 : La couverture au sein de la généralisation/spécialisation.

Il y a **partition**
du types d'entité générique par ses types d'entités spécifiques
ssi
il y a disjonction et couverture
des types d'entité spécifiques

Nous apporterons à cette notion la notation graphique suivante :

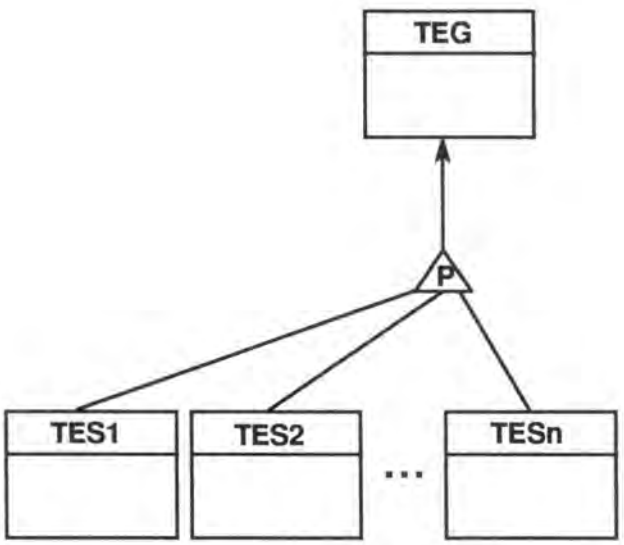

Fig. 5.4 : La partition au sein de la généralisation/spécialisation.

Lorsque des spécifications sont ajoutées concernant les aspects techniques des structures de données, un schéma TRAMIS correspond à une sorte de schéma logique. Il peut comprendre des constructions additionnelles telles que les clefs d'accès[37], des chemins d'accès[38], des espaces et des contraintes d'inclusion.

Ce modèle générique de spécification peut être spécialisé en une grande variété de sous-modèles, par exemple selon les niveaux de conception d'une méthode à standard ou définie par l'utilisateur, ou selon le système de gestion de base de données cible. La spécialisation du modèle générique vers un sous-modèle spécifique peut être effectuée en établissant une liste de règles que le schéma doit satisfaire. Par exemple, un schéma TRAMIS peut être un schéma de type MERISE s'il satisfait aux règles suivantes:

---

[37] Mécanismes abstraits d'accès basés sur la valeur des objets (indices,...).

[38] Exprime qu'un type de relation peut être utilisé par une application pour parcourir la base de données.

- Les types d'entités possèdent au moins un attribut.
- Les attributs sont simples, obligatoires et atomiques.
- Les types d'entités ont un et un seul identifiant.
- Un identifiant est composé d'un attribut.
- Les contraintes de cardinalités autorisées sont 0-1, 1-1, 0-N et 1-N.
- Les types d'associations possèdent un et un seul attribut.
- Un identifiant d'un type d'association est composé uniquement de rôles.
- Il n'y a pas de clef d'accès, ni de chemin d'accès, ni d'espace (c'est à dire que le schéma est purement conceptuel).

TRAMIS propose environ une centaine de règles paramètrables qui peuvent être utilisées pour définir des sous-modèles.

L'atelier logiciel TRAMIS possède quelques définitions de sous-modèles intégrés : ER standard, relationnel, CODASYL, fichiers COBOL, RDB, SYBASE, ORACLE, etc. De plus, les utilisateurs peuvent définir un nombre quelconque de sous-modèles spécifiques en construisant un fichier ASCII contenant une liste de règles sous un format particulier ; ces définitions sont automatiquement intégrées dans l'outil à l'exécution.

5.2.4 Les transformations de schémas.

Le concept de transformation est particulièrement attrayant dans le domaine des bases de données. Une transformation de schéma est généralement considérée comme un opérateur par lequel une structure de données S1 est remplacée par une autre structure S2 qui possède une certaine équivalence avec S1. La transformation de schéma est un concept omniprésent dans la conception de bases de données. Les activités de conception de base qui peuvent être effectuées en choisissant soigneusement des transformations de schémas sont les suivantes :

- Fournir une équivalence entre des schémas.
- Affiner un schéma conceptuel.
- Intégrer deux schémas partiels.
- Produire un schéma cohérent avec un certain type de système de gestion de bases de données à partir d'un schéma conceptuel.
- Restructurer un schéma physique.
- Reverse engineering de bases de données.

La notion de **réversibilité** est une caractéristique importante d'une transformation. Si une transformation est réversible, alors les schémas sources et destinations ont la même puissance de description, et décrivent le même univers,

bien qu'ils utilisent des présentations différentes. Les transformations réversibles conservent aussi la sémantique. Les transformations fournies par TRAMIS conservent, non seulement la sémantique du schéma source, mais aussi les autres aspects de la spécification : les structures d'accès, les noms, les informations statistiques et informelles. Par exemple, les structures d'accès dans le schéma source S1 sont transformées en structures d'accès différentes, mais sont fonctionnellement équivalentes dans le schéma destination S2 ; les statistiques de S1 sont recalculées selon la nouvelle structure de S2 ; les spécifications informelles des objets qui disparaissent de S1 sont transférées, avec des adaptations possibles, dans les nouveaux objets de S2. Les transformations de TRAMIS ne conservent pas seulement la sémantique mais également la spécification.

TRAMIS offre une large gamme de transformations qui peuvent être utilisées à différents niveaux du processus de conception. Elles ne sont pas orientées vers un but précis[39] et peuvent donc être utilisées dans n'importe quelle stratégie ou processus de conception.

Les transformations concernent :

- Les types d'entités. Exemple : décomposition d'un type d'entité en plusieurs autres en extrayant des attributs et des rôles.
- Les types de relations. Exemple : transformation de types d'associations en types d'entités.
- Les attributs. Exemple : transformation d'attributs composés en types d'entités.
- Les identifiants et les clés d'accès.
- Les noms donnés aux types d'entités et d'associations.
- Les aspects non-conceptuels.

Nous n'allons cependant pas parcourir l'ensemble des transformations possibles offertes par TRAMIS.

TRAMIS propose un ensemble d'outils de transformations à trois niveaux qui peuvent être utilisés librement selon l'expérience de l'utilisateur, l'efficacité du schéma exécutable souhaité et le temps disponible pour la production de ce schéma :

- **Les transformations élémentaires :** une transformation est appliquée à un objet ; avec ces outils, l'utilisateur garde le contrôle total de la

[39] Sauf pour les transformations guidées par un modèle.

transformation du schéma puisque des situations similaires peuvent être résolues par différentes transformations ; par exemple, un attribut répétitif peut être transformé de la façon suivante :

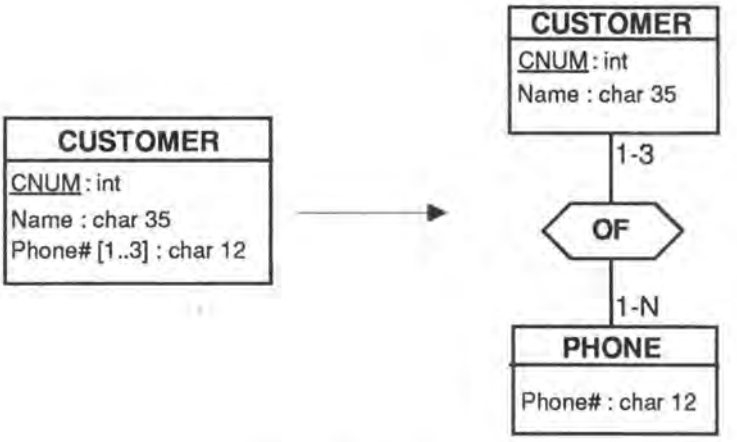

Fig. 5.5 : Illustration.

- **Les transformations globales :** une transformation est appliquée à tous les objets d'un schéma ; exemples : remplacer toutes les relations 1-N par des clés secondaires et des contraintes référentielles, remplacer tous les attributs multivalués par des types d'entités et des types d'associations 1-N.

- **Les transformations basées sur un modèle :** toutes les structures d'un schéma qui ne satisfont pas aux règles d'un sous-modèle donné sont transformées ; ces transformations ne nécessitent pas le contrôle de l'utilisateur.

Il est important de noter que les transformations des trois niveaux peuvent être utilisées quel que soit l'état dans lequel se trouve un schéma. Par exemple, une transformation basée sur un modèle peut être utilisée pour produire rapidement un schéma SQL à partir d'un schéma conceptuel pur[40]. Elle peut être également utilisée pour donner la touche finale à un schéma qui a déjà été optimisé, et qui a déjà été, en partie, rendu conforme au modèle SQL par des transformations élémentaires et globales[41].

5.2.5 L'architecture de TRAMIS.

L'architecture de TRAMIS est basée sur deux composants principaux : le répositoire et la structure de boîte à outils.

---

[40] C'est le cas d'un concepteur novice ou le cas d'un prototypage rapide.

[41] C'est le cas d'un concepteur expérimenté.

Le **répositoire** de TRAMIS contient l'état courant du schéma en développement. Il est implémenté avec MDBS-3, un système de gestion de base de données de type CODASYL compact et très performant. Un module d'accès a été construit pour donner une interface entité-association aux fonctions, fournissant un atelier avec des exigences d'indépendance dans le développement et d'indépendance vis-à-vis du SGBD. Le répositoire, est en lui-même, une base de données de complexité moyenne puisque son schéma contient environ 30 types d'entités. Ces 30 types d'entités forment un méta-schéma dont la "valeur" représente l'état de la base de donnée.

Attardons nous un peu sur la structure et l'utilisation de ce méta-schéma. Il est implémenté en C++. Chaque type d'entité est donc une classe au sens C++. Il existe des classes comme ENTITY_TYPE, ATTRIBUTE, REL_TYPE, CLUSTER,etc. Toutes ces classes représentent un concept du modèle générique de TRAMIS. Toutes ces classes héritent s'une super-classe appelée GENERIC_OBJECT. Cette super-classe permettra une manipulation aisée des instances de n'importe quelle classe.

Ces classes sont reliées entre-elles par des relations. Exemple : une instance de la classe ENTITY_TYPE est associée à 0, 1, ou plusieurs instances de la classe ATTRIBUTE. Ces relation sont matérialisées par une liste de pointeurs vers des GENERIC_OBJECT.

Chaque classe possède des attributs qui les caractérisent. Exemple : la classe ATTRIBUT possède les attributs nom et type dans sa classe C++.

Chaque classe possède une série de méthodes qui :

- permettent ou non l'accés à la valeur des attributs des instances d'une classe. Exemple : la méthode `get_name()` de la classe ATTRIBUTE retourne le nom de l'attribut associé.
- permettent ou non la modification de ces valeurs.
- permettent d'accéder aux instances de la classe. Exemple : la méthode `get_first_entity_type()` retourne un pointeur vers la première instance de la classe ENTITY_TYPE.
- sont utlisées pour accèder aux objets associés. Exemple : la méthode `get_first_attribute()` de la classe ENTITY_TYPE retourne un pointeur vers la premiere instance d'ATTRIBUTE associée.
- qui effectuent des opérations plus complexes dans le méta-schéma.

Les fonctions de l'atelier travaillent directement sur les données contenues dans le schéma, une performance qui n'aurait pas été réaliste sur des stations de travail PC avec un SGBD relationnel.

La **structure de boîte à outils** donne aux utilisateurs une liberté maximum au niveau de leur comportement dans la conception. En effet, on peut appliquer aisément des stratégies de type incrémentales, d'essai & erreur, de prototypages. TRAMIS fournit un ensemble d'outils qui peuvent être utilisés librement. Les outils sont indépendants les uns des autres et ne communiquent que par le répositoire. Aussi bien les concepteurs expérimentés que les novices peuvent trouver leur compte : certains utilisateurs auront besoin d'outils très fins sur lesquels ils doivent avoir un contrôle total, tandis que d'autres auront besoin d'outils simples et automatisés pour de la production rapide et des descriptions exécutables.

Les outils sont groupés dans des ensembles qui peuvent être accessibles par un gestionnaire de dialogue commun fournissant une interface de type WIMP.

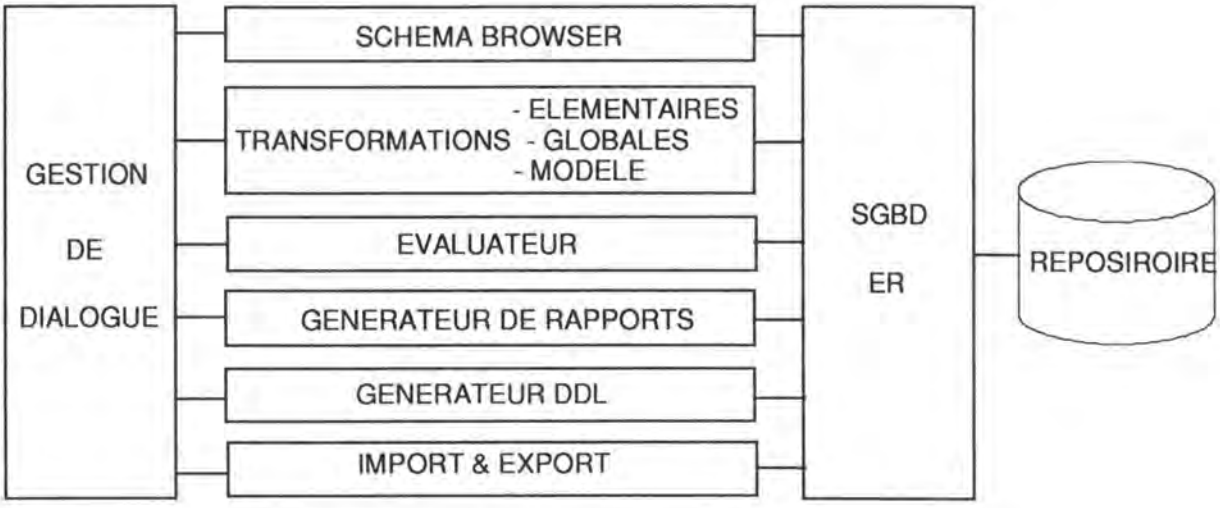

Fig. 5.6 : Architecture de l'atelier logiciel TRAMIS.

Le "browser" de schéma (schema browser) : il permet le parcours de l'état actuel de la spécification de la base de données, la sélection d'un objet par son nom ou dans une liste, et la navigation d'un objet vers les objets qui lui sont associés. Il permet la consultation et la mise à jour des différents aspects de l'objet courant avec la création et la suppression d'objets.

La transformation de schémas : permet l'application d'une transformation élémentaire sur l'objet courant, l'application d'une transformation sur tous les objets du schéma, et l'application de toutes les transformations nécessaires sur les objets du schéma pour le transformer en un nouveau schéma qui est cohérent

avec le sous-modèle choisi. Toutes les facettes des objets transformés sont reconstruites pour préserver les informations contenues.

L'évaluateur : à tout moment, l'utilisateur peut valider ses spécifications par rapport à un sous-modèle choisi. TRAMIS compile les règles définissant le sous-modèle et analyse les spécifications courantes par rapport à ces règles. TRAMIS génère des diagnostiques qui seront examinés par l'utilisateur.

Le générateur de rapports : permet la production d'un rapport détaillé pour tous les objets ou pour un objet particulier, d'un rapport statistique, ou encore d'un rapport contenant une liste structurée de noms d'objets avec des traductions de noms naturels vers des noms techniques et inversement (le dictionnaire).

La génération DDL : c'est la traduction d'un schéma TRAMIS dans des descriptions exécutables selon un SGBD choisi. La génération est double : premièrement la génération d'un texte DDL, et deuxièmement, la génération d'un document supplémentaire qui concerne toutes les contraintes d'intégrités, qui n'ont pas été traduites dans le texte DDL avec les procédures de validation de ces contraintes. Ce rapport contient les générateurs pour les gestionnaires de base de données suivants : CODASYL, RDB, SYBASE, ORACLE, les fichiers COBOL standards, etc.

Import/export : TRAMIS est doté d'un langage de spécification externe dans lequel toute spécification peut être exprimée. Ce langage permet la communication entre les outils, un backup sélectif et même des entrées de données.

TRAMIS a été implémenté en C++ dans l'environnement MS-Windows 3.1.

5.3 Un module de spécification orientée objets.

Il va être temps à présent d'imaginer la création d'un module supplémentaire pour TRAMIS concernant les bases de données orientées objets.

Il est nécessaire de définir un sous-modèle supplémentaire qui possède les règles suivantes par rapport au modèle générique utilisé par TRAMIS :

- Un type d'association a un degré = 2.
- Un type d'association n'a pas d'attributs.
- Les contraintes de cardinalités autorisées sont 0-1, 1-1, 0-N et 1-N.

- Si un type d'entité possède un rôle avec une contrainte de cardinalité 1-N, l'autre de rôle du type d'association concerné est autorisé à posséder les contraintes de cardinalités suivantes : 0-1, 1-1.
- Un attribut ne peut pas être composé.
- Un type d'entité ne peut avoir qu'un seul cluster.
- Un type d'entité ne peut être membre que d'un seul cluster (pas d'héritage multiple).

Une fois le sous-modèle défini, voici les différentes étapes que va suivre le concepteur de la base de données orientées objets :

1. Premièrement, il va concevoir et spécifier sa base de données en utilisant le modèle proposé au chapitre précédent.

2. Il va ensuite utiliser TRAMIS grâce auquel il va pouvoir exprimer la conception de sa base de données via le modèle générique. Dans le cas contraire, les outils de l'atelier logiciel TRAMIS seront là pour supporter les éventuelles transformations nécessaires pour répondre aux exigences de notre sous-modèle. C'est ce qui est exprimé dans la vue "TRAMIS" de la figure 5.6. L'utilisateur procède par prototypage.

3. Enfin, dès que la représentation de la base de données sera conforme au sous-modèle, notre module générera un fichier contenant le code exécutable à partir des informations contenues dans le méta-schéma de TRAMIS. Ce code exécutable (représenté dans la vue "code généré" de la igure 5.6) sera constitué de deux parties :

    - celle qui contient le code nécessaire à la gestion de la base de donnée (définitions de la base de données, des objets utilisés, des accès aux objets, ...)
    - celle qui contient le code nécessaire pour la gestion des dialogues avec la base de données. Elle est constituée essentiellement par la surcharge des opérateurs d'entrée et de sortie.

Le code exécutable sera du C++. Il est clair que l'interface n'est qu'un artifice et que l'essentiel du code "utile" se situe dans la partie gestion de la base de données. Un utilisateur de la base de données, c'est à dire du code généré, pourra construire une application complète à l'aide de celui-ci. C'est ce qui est exprimé par la vue "application" de la figure 5.6.

La figure ci-dessous illustre le module supplémentaire développé pour TRAMIS.

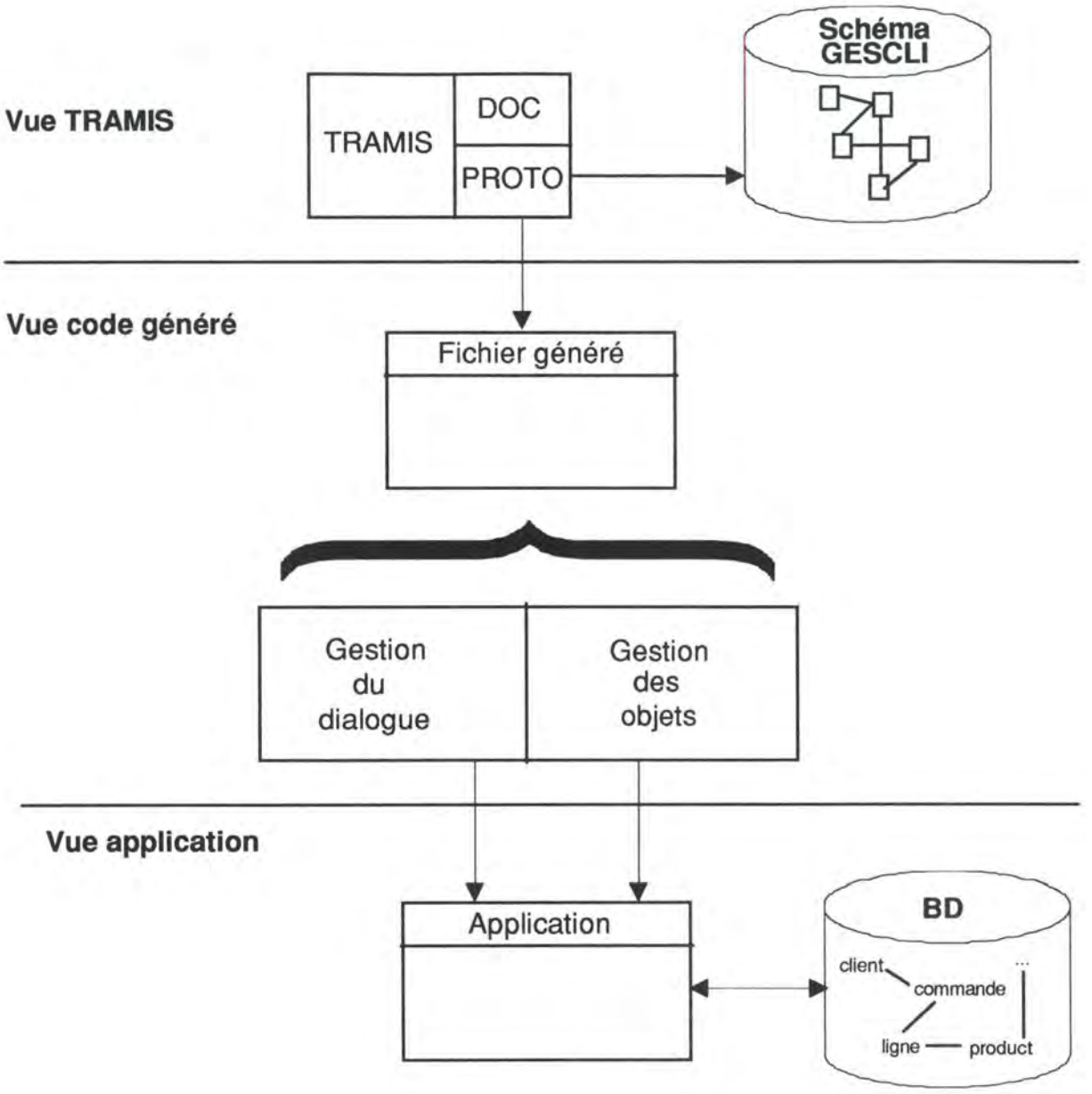

Fig. 5.6 : Illustration du nouveau module de TRAMIS.

CHAPITRE VI : L'implémentation.

6.1 La représentation physique de la base de données.

Il s'agit des classes d'objets, de leurs instances, de leurs associations et de leurs relations d'héritage.

6.1.1 Les classes d'objets et leurs instances.

La base de données physique est constituée de l'ensemble des instances de ses classes d'objets. A chaque classe d'objets correspond une liste de ses instances. L'ensemble de ces listes constitue la base de données. Ces listes seront matérialisées par une liste de pointeurs vers les intances de la classe d'objets concernée.

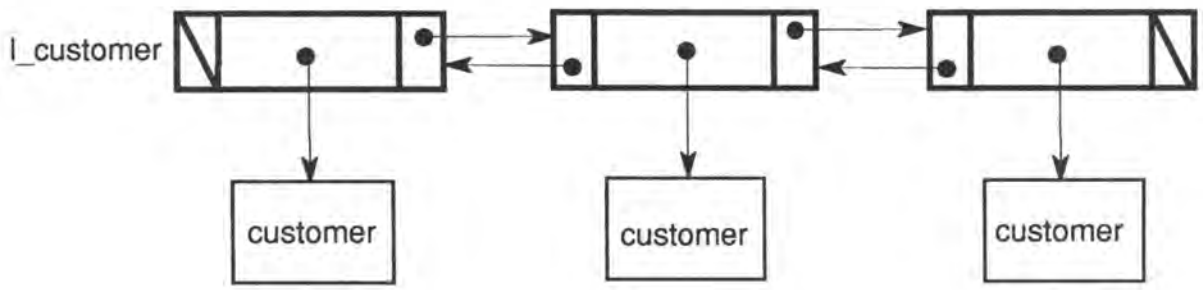

Fig. 6.1 : Illustration de la liste d'instances de la classe d'objets `CUSTOMER`.

6.1.2 Les classes d'objets, leurs instances et leurs associations.

Pour chaque instance d'une classe d'objets, il existe une série d'objets associés. Ces associations seront représentées physiquement, soit par un cluster de pointeurs vers ces objets, soit par un pointeur isolé, selon le nombre d'objets associés d'une classe d'objets particulière. Ces clusters sont matérialisés par le même type de listes de pointeurs que celles qui sont utilisées dans la figure 6.1.

Commentons la figure 6.2 : le cluster signifie qu'à une instance de la classe d'objets `customer` correspond une ou plusieurs instances de la classe `order`. Le pointeur signifie qu'à une instance de la classe d'objets `order` correspond une instance de la classe d'objets `customer`. Nous avons bien là une matérialisation du rôle 1-N de la relation de `customer` vers `order` et du rôle 1-1 de la relation de `order` vers `customer`.

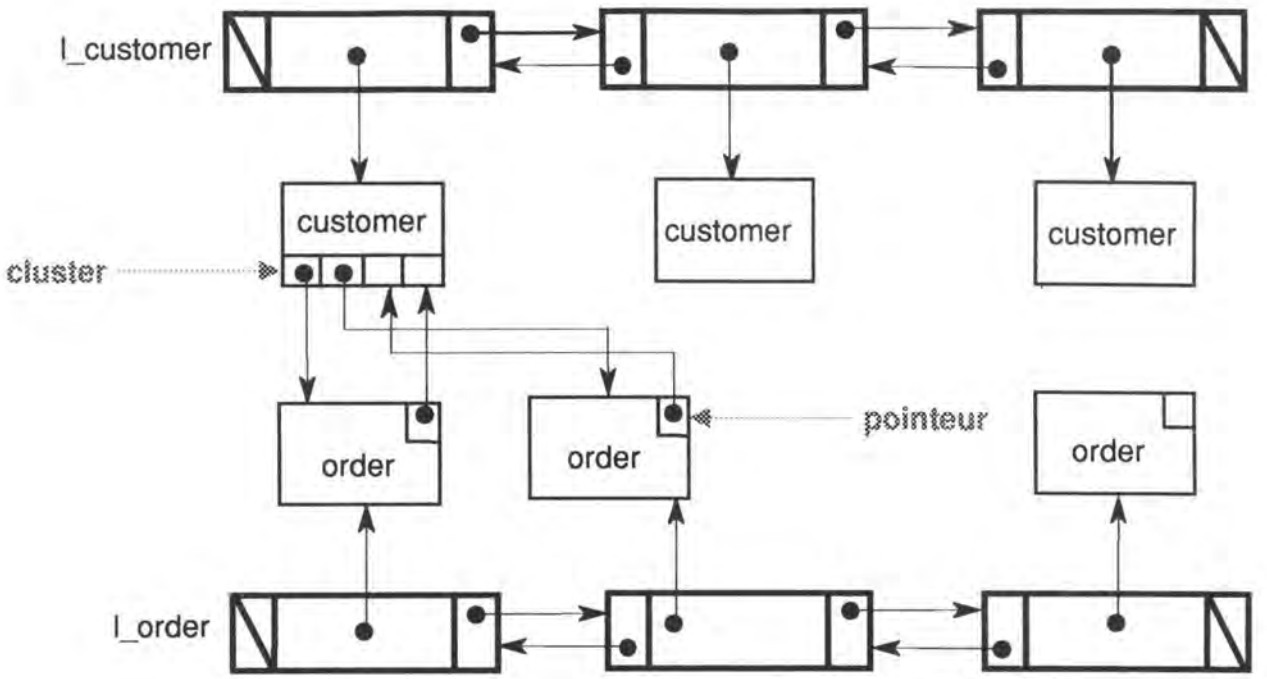

Fig. 6.2 : Illustration de la représentation physique des objets associées.

6.1.3 Les classes d'objets et leurs relations d'héritage.

Lorsqu'une classe d'objets A hérite d'une classe d'objets B, on dit que "A est un B" (IS-A). Dans notre exemple, `priv_cus` est un `customer`, `pub_cus` est un `customer`. Si on considère la relation de couverture ou pas, entre les différentes classes d'objets spécifiques, membres du cluster :

- un `customer` peut être un "simple" `customer`, un `pub_cus` ou un `priv_cus` ; c'est le cas de la non couverture.
- un `customer` est, soit un `pub_cus`, soit un `priv_cus` ; c'est la cas de la couverture.

Quel que soit le cas considéré, toutes des instances des classes d'objets spécifiques feront partie de la liste des instances de la classe d'objets générique. Cette représentation physique est décrite dans la figure suivante.

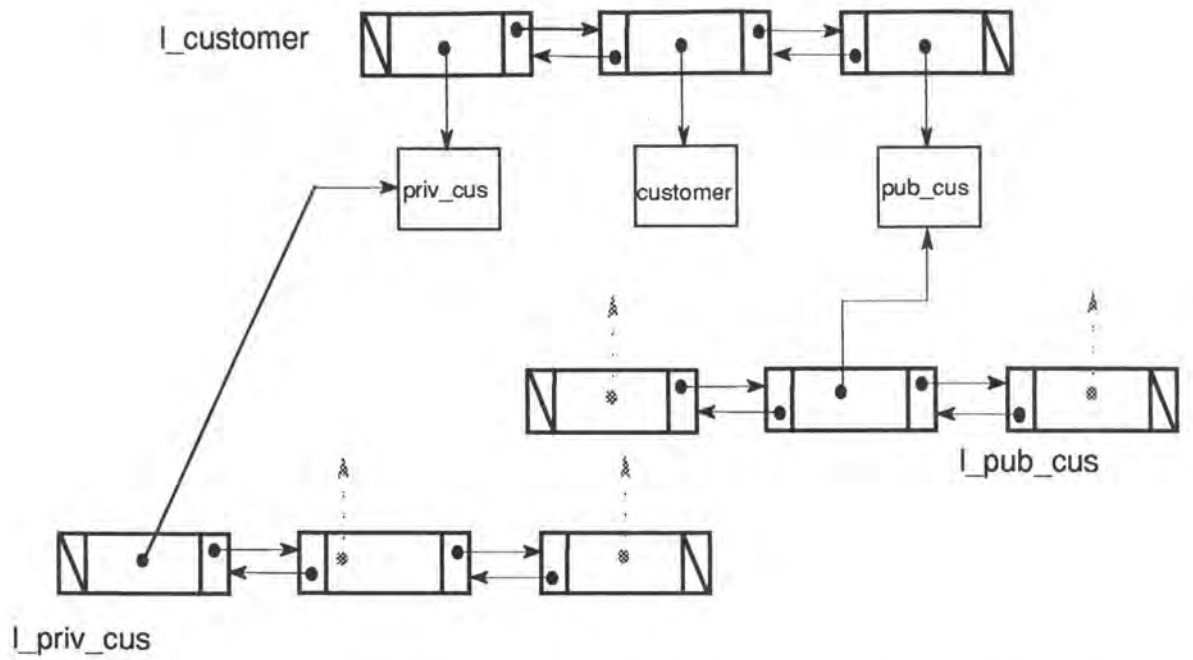

Fig. 6.3 : Illustration de la représentation physique des relations d'héritage.

6.1.4 Remarques concernant l'intégrité de la base de données.

Soulignons déjà les conséquences importantes pour l'**intégrité** que la base de données devra conserver avec une telle représentation physique :

- Nous ne devons pas oublier qu'un objet ne pourra être associé à un autre que si ce dernier existe déjà dans la liste des instances de sa classe d'objet. Remarquons aussi que nous n'aurons aucune redondance ni de duplication d'objets.
- Les connectivités des relations entre différentes classes d'objet peuvent être plus au moins restrictives selon qu'on oblige par exemple, une instance de `customer` à être associé à au moins une instance de la classe d'objets `order` ou non. L'utilisateur de la base de données devra être conscient de ces restrictions lors de la manipulation des données.
- Une instance d'une classe spécifique devra être à la fois présente dans la liste des instances de sa classe d'objets, mais également dans la liste des instances de sa classe générique.

6.2 la représentation des types de données.

Les différents types des données (attributs) proposés par TRAMIS devront être supportés par notre base de données. Nous allons passer en revue les différents types et exposer notre représentation physique choisie.

Le **type alphanumérique** sera représenté par une chaîne de caractères de type "string". Il en sera de même pour le **type date**. Ce type de données "string" sera défini dans une librairie que nous décrirons plus tard.

Le **type naturel** sera représenté par les type C++ `int`, `short int` ou `long int` selon la longueur de l'attribut concerné.

Le **type booléen** sera également défini par une librairie prédéfinie.

Concernant le **type composé**, nous avons vu qu'il n'était pas supporté par notre sous-modèle. Il pourra cependant être matérialisé par la définition d'une nouvelle classe d'objets.

Il en sera de même pour les **types définis par l'utilisateur**. Ils forment des classes d'objets à part entière.

6.3 Les librairies utilisées pour la représentation de la base de données et de ses données.

Nous avons conçu des librairies nécessaires

1. à la représentation physique de notre base de données (les listes de pointeurs génériques "liste_pt" dans la librairie `l_pt.h`).
2. à la représentation de ses données (la librairie `strings.h` pour la classe d'objet string et la librairie `boolean.h`).

6.3.1 La classe d'objets "liste_pt".

Cette classe est une classe d'objets générique. Dans la terminologie du C++, c'est une "classe template". Elle permet de définir de façon tout fait générique une liste de pointeurs vers des objets de type entier, caractère ou encore de tout autre type défini par l'utilisateur. Exemples :

- `liste_pt<int> liste_entiers;`
- `liste_pt<char> liste_de_caractères;`
- `liste_pt<customer> liste_de_customer;`

- `liste_pt<T> liste_de_T;`

Une fois définie, une liste de pointeurs manipule les objets vers lesquels elle pointe quel que soit le type de T.

Important : une seule condition est requise pour qu'un type d'objets puisse être manipulé par la classe liste_pt : son implémentation et sa déclaration doivent être de la **forme canonique orthodoxe.** Nous reviendrons sur la notion de déclaration de classe canonique orthodoxe plus tard dans ce chapitre.

REPRESENTATION PHYSIQUE

La **représentation physique** de la classe l_pt est une liste classique doublement chaînée, dont les éléments sont des pointeurs vers un type générique quelconque. Elle possède un pointeur sur le premier et le dernier élément de sa liste. Elle possède également un pointeur sur un de ses éléments dit courant.

Important : concernant le pointeur sur l'élément courant, nous considérons qu'il pointe toujours sur un des éléments de sa liste. Sa valeur n'est donc jamais égale à `NULL`, excepté si la liste est vide.

A tout moment, une liste pointe vers un des ses élément dit **courant**

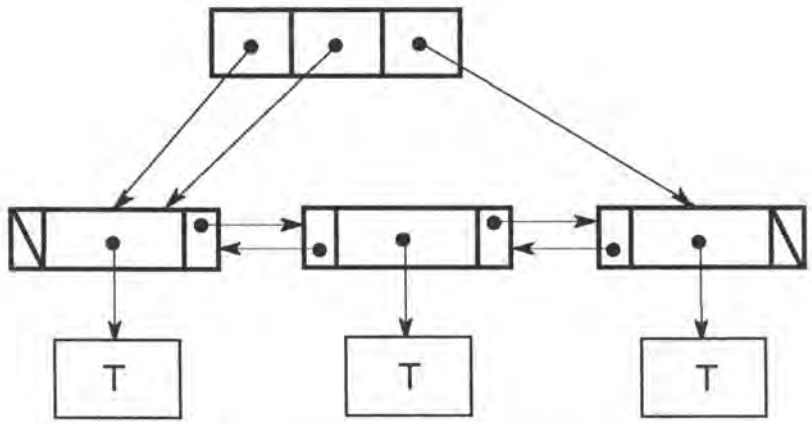

Fig. 6.4 : Représentation physique de la classe d'objets L_PT.

DESCRIPTION DES METHODES

L'interface de la classe est décrite dans l'interface 6.1. Nous allons parcourir ensuite les différentes méthodes en les commentant.

```
#ifndef LISTE_PT
#define LISTE_PT

#include "boolean.h"

template <class T> class elem;
template <class T> class liste_pt;

// Classe ELEM
//_____________

template <class T> class elem {

friend class liste_pt;
friend ostream& operator<< (ostream&,liste_pt<T>&);
friend ostream& operator<< (ostream&,elem<T>&);

public:
      elem() : next(0),valeur(0),previous(0) {}
      elem(const T &typ) : next(0),previous(0)
            { valeur = new T (typ);}
     ~elem(){delete valeur;}

      elem<T> *next;
      T *valeur;
      elem<T> *previous;
};

            //**************************************

template <class T>
class liste_pt {

friend ostream& operator<< (ostream&,liste_pt<T>&);

public:

    liste_pt() : start(0),current(0),end(0),len(0) {}
    liste_pt(const liste_pt<T> &q)
            : start (0),current(0),end(0),len(0)
    ~liste_pt() {empty()

    int length(void) const ;
    atbegin() ;
    atend() ;
    boolean is_empty(void) const ;

    void append (T &val);
    retrieve(void);
    void insert(T &val);
    void insert(T* ptr);
    void insert(T *ptr,T &val);
    remove();
    void del(T *ptr);

    T *getfirst();
    T *getcurrent();
    T *getnext();
    T *getnext(T *ptr);
    T *getprevious();
    T* getprevious(T *ptr);
    T *getlast();

    liste_pt<T>& operator+= (const liste_pt<T> &b);
    liste_pt<T>& operator= (const liste_pt<T> &b);

    void empty();
```

```
private:

    elem<T> *start;
    elem<T> *current;
    elem<T> *end;
    int len;
};

#endif
```

Interface. 6.1 : Interface de l'objet L_PT.

- **liste_pt() :** constructeur par défaut de la classe.
- **liste_pt(const liste_pt<T> &q) :** constructeur de copie de la classe.
- **~liste_pt() :** destructeur de la classe.
- **int length(void) const :** retourne le nombre d'éléments de la liste.
- **atbegin() :** positionne le pointeur de l'élément courant sur le premier élément de la liste.
- **atend() :** positionne le pointeur de l'élément courant sur le dernier élément de la liste.
- **boolean is_empty(void) :** renvoie `TRUE` si la liste est vide, `FALSE` si elle ne l'est pas.
- **void append (T &val) :** ajoute à la fin de la liste un élément de type T passé par adresse.
- void append(T *ptr) : ajoute à la fin de la liste un élément de type T pointé par `ptr`.
- **retrieve(void) :** enlève le dernier élément de la liste;
- **void insert(T &val) :** insère un élément de type T passé par adresse avant l'élément courant. L'élément courant devient le nouvel élément.
- **void insert(T* ptr) :** insère un élément de type T pointé par `ptr` avant l'élément courant. L'élément courant devient le nouvel élément créé.
- **void insert(T *ptr, T &val) :** insère un élément de type T passé par adresse dans la liste après l'élément pointé par `ptr`.
- **remove() :** supprime l'élément pointé par le pointeur sur l'élément courant. L'élément courant devient l'élément qui précède celui qui va être supprimé.
- **void del(T *ptr) :** supprime l'élément de la liste pointé par `ptr`. L'élément courant devient l'élément qui précède celui qui devait être supprimé si il était l'élément en question.
- **T *getfirst() :** retourne un pointeur vers le premier élément de la liste ; s'il n'existe pas, retourne `NULL`. L'élément courant devient ce dernier.
- **T *getcurrent() :** retourne un pointeur vers l'élément courant de la liste ; s'il n'existe pas, retourne `NULL`.
- **T *getnext() :** retourne un pointeur vers l'élément qui suit l'élément courant ; si il n'existe pas, retourne `NULL`. L'élément courant devient ce dernier.

- **T *getnext(T *ptr) :** retourne un pointeur vers l'élément qui suit celui pointé par `ptr` ; s'il n'existe pas, retourne `NULL`.
- **T *getprevious() :** retourne un pointeur vers l'élément qui précède l'élément courant ; s'il n'existe pas, retourne `NULL`. L'élément courant devient ce dernier.
- **T *getprevious(T *ptr) :** retourne un pointeur vers l'élément qui précède celui pointé par `ptr` ; s'il n'existe pas, retourne `NULL`.
- **T *getlast() :** retourne un pointeur vers le dernier élément de la liste ; si il n'existe pas, retourne `NULL`. L'élément courant devient ce dernier.
- **liste_pt<T>& operator+= (const liste_pt<T> &b) :** surcharge de l'opérateur +=.
- **liste_pt<T>& operator= (const liste_pt<T> &b) :** surcharge de l'opérateur =.
- **void empty() :** vide une liste.

DESCRIPTION DES PROCEDURES GENERALES

- **friend ostream& operator<< (ostream&,liste_pt<T>&) :** surcharge de l'opérateur de sortie.

6.3.2. La classe d'objets "string".

DESCRIPTION PHYSIQUE

Un string est une chaîne de caractères pointée par un pointeur sur le premier de ses caractères, d'une longueur définie et terminée par le caractère spécial "\0".

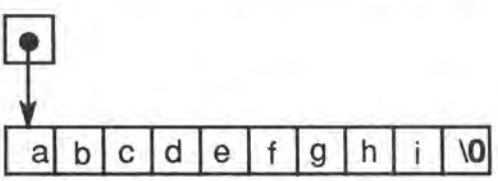

Fig. 6.5 : Représentation physique d'un string de longueur 9.

DESCRIPTION DES METHODES

```
#ifndef STRING
#define STRING

class string {
```

```
    friend string operator+ (const string&,const string&);
    friend string operator+ (const string&,const char*);
    friend string operator+ (const char*,const string&);

    friend int operator== (const string&,const string&);
    friend int operator== (const string&,const char*);
    friend int operator== (const char *,const string&);

    friend int operator!= (const string&,const string&);
    friend int operator!= (const string&,const char*);
    friend int operator!= (const char*,const string&);

    friend int operator< (const string&,const string&);
    friend int operator< (const string&,const char*);
    friend int operator< (const char*,const string&);

    friend int operator> (const string&,const string&);
    friend int operator> (const string&,const char*);
    friend int operator> (const char*,const string&);

    friend int operator>= (const string&,const string&);
    friend int operator>= (const string&,const char*);
    friend int operator>= (const char*,const string&);

    friend int operator<= (const string&,const string&);
    friend int operator<= (const char*,const string&);
    friend int operator<= (const string&,const char*);

    friend ostream& operator<<(ostream& os,const string& str);
    friend istream& operator>>(istream& is,string &str);

public:

    string ();
    string (const char*);
    string (const string&);
    string (const long int);
    string (const char);

    int length() const;
    string operator() (int) const;
    string operator() (int,int) const;
    int index (char) const;

    string& operator=(const string&);
    string& operator=(const char*);

    string& operator+=(const string&);
    string& operator+=(const char*);

    ~string();

private:

    char *value;
    long int ln;
};

#endif
```

Interface 6.2 : L'interface de la classe d'objets `STRING`.

- **string () :** constructeur par défaut.
- **string (const char*) :** constructeur.

- **string (const string&) :** constructeur de copie.
- **string (const long int) :** constructeur.
- **string (const char) :** constructeur
- **int length() const :** retourne la longueur du string.
- **string operator() (int i) const :** à partir retourne une partie string délimitée par i et la taille maximum du string.
- **string operator() (int i,int j) const :** retourne un segment de string délimité par les positions i et j
- **string& operator=(const string&) :** surchage de l'opérateur =.
- **string& operator=(const char*) :** surchage de l'opérateur =.
- **string& operator+=(const string&) :** surcharge de l'operateur +=.
- **string& operator+=(const char*) :** surcharge de l'operateur +=.

DESCRIPTION DES PROCEDURES GENERALES

Elles sont constituées par la surcharge classique des opérateurs <, >, >=, <=, ==, !=, et + avec des combinaisons différentes des types d'arguments :

- friend string operator+ (const string&,const string&);
- friend string operator+ (const string&,const char*);
- friend string operator+ (const char*,const string&);
- friend int operator== (const string&,const string&);
- friend int operator== (const string&,const char*);
- friend int operator== (const char *,const string&);
- friend int operator!= (const string&,const string&);
- friend int operator!= (const string&,const char*);
- friend int operator!= (const char*,const string&);
- friend int operator< (const string&,const string&);
- friend int operator< (const string&,const char*);
- friend int operator< (const char*,const string&);
- friend int operator> (const string&,const string&);
- friend int operator> (const string&,const char*);
- friend int operator> (const char*,const string&);
- friend int operator>= (const string&,const string&);
- friend int operator>= (const string&,const char*);
- friend int operator>= (const char*,const string&);
- friend int operator<= (const string&,const string&);
- friend int operator<= (const char*,const string&);
- friend int operator<= (const string&,const char*);

Nous avons également surchargé les opérateurs d'entrée et de sortie.

- friend ostream& operator<<(ostream& os,const string& str);
- friend istream& operator>>(istream& is,string &str);

6.3.3 Le type "boolean".

```
#ifndef BOOLEAN
#define BOOLEAN

//Type BOOLEEN
//_____________

enum boolean {true=1,false=0};

#endif
```

Interface 6.3 : Le type boolean.

6.4 Le méta-schéma.

La génération de code C++ de notre base de données se base sur le meta-schéma de l'atelier logiciel TRAMIS. Ce dernier sera la source unique d'informations nécessaires à la génération du code C++. Nous n'allons pas entrer dans la description de ce meta-schéma ; retenons simplement qu'il représente l'état courant de la base de donnée conceptuelle dans le modèle générique de TRAMIS. Nous allons cependant utiliser notre propre meta-schéma logique. Ce dernier, dans son implémentation, fait appel aux méthodes des classes d'objets du méta-schéma de TRAMIS

Notre méta-schéma est décrit conceptuellement dans la figure ci-dessous. Le formalisme utilisé est celui du modèle proposé au chapitre 4.

- Un méta-schéma possède un nom.
- Un méta-schéma est un ensemble d'objets.
- Un objet possède un ou plusieurs attributs.
- Un objet possède éventuellement des associations.
- Un objet possède éventuellement un cluster (c'est à dire qu'il est un type générique).
- Un attributs possède un nom, une longueur et un type.
- Une association a une connectivité maximale et un objet source et un objet destination.
- Un cluster possède plusieurs objets spécifiques.

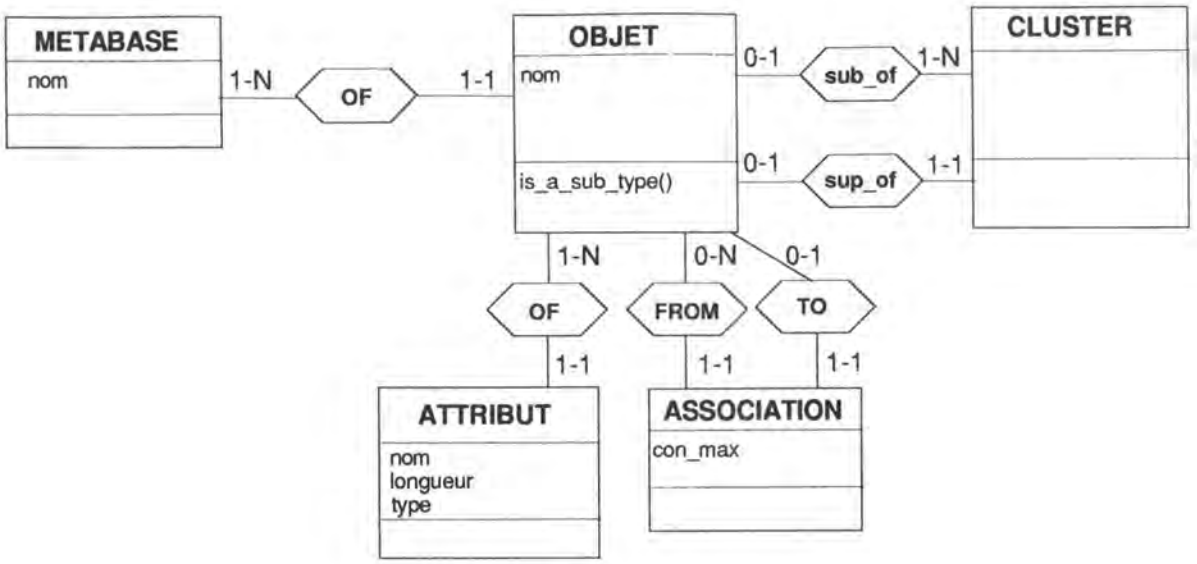

Fig. 6.6 : le méta-schéma conceptuel.

Une librairie supplémentaire à été crée afin de supporter cette "méta-base". Elle utilise une autre librairie `listestr.h` qui définit une classe d'objets semblable à `liste_pt`, mais spécialisée dans la liste de pointeurs vers des strings. Voici l' interface de notre "méta-base":

```
#include "string.h"
#include "l_pt.cpp"
#include "listestr.h"

class metabase;
class objet;
class attribut;
class association;


//***************
//Classe ATTRIBUT
//***************

class attribut {
public:
      attribut() : nom(""),type(""), ln(0) {;}
      attribut(const attribut&)
      ~attribut() ;

                attribut& operator=(const attribut&);

      put_nom(string) ;
      put_type(string) ;
      string get_nom() ;
      string get_type() ;
      put_ln(int l) ;
      int get_ln() ;

private:
      string nom;
      string type;
      int ln;
};

//******************
```

```
//Classe ASSOCIATION
//******************

class association {
public:
      association() : destination(""),con_max(0) {;}
      association(const association &ass);
      ~association() ;

            association& operator=(const association&);

           int get_max_con() ;
           put_con_max(int );
           string get_destination() ;
           put_destination(string ) ;

           string destination;
           int con_max;
};
//************
//Classe OBJET
//************
class objet {


public:
      objet() : nom(""),super_type(""),sub_type(false) {;}
      objet(const objet &);
      ~objet() ;

      objet& operator=(const objet&);

      string get_nom() ;
      put_nom(string n) ;
      boolean is_a_sub_type() ;
      put_is_a_sub_type(boolean b) ;
      put_super_type(string str) ;
      string get_super_type() ;

      string nom;
      liste_pt<attribut> attributs;
      liste_pt<association> associations;
      boolean sub_type;
      string super_type;
      listestr cluster;
};

//****************
//Classe METABASE
//****************

class metabase {
public:
      metabase() : nom("") {;}
      metabase(const metabase&);
      ~metabase() ;

      metabase& operator=(const metabase&);

      string get_nom() ;
      put_nom(string  n);

      string nom;
      liste_pt<objet> objets;
};
```

Interface 6.4 : Interface du méta-schéma.

6.5 La forme du code généré.

Nous allons à présent présenter la forme que possèdera le code C++ généré. Dans le cadre du développement du générateur C++, nous avons essentiellement travaillé par prototypage et par essais de différentes solutions possible. Dans ce qui suit nous allons présenter pas à pas le raisonnement qui nous amené à la solution la plus stable en fonction de la représentation physique que nous avions choisie dès le départ.

6.5.2 La forme canonique orthodoxe de classe.

Dans le cadre d'une génération de code et toujours dans un soucis de stabilité et cohérence de ce code, nous devons nous fixer une forme d'implémentation fixe des classes d'objets qui devront être générées. Nous allons utiliser la forme canonique orthodoxe de classe.

La forme canonique orthodoxe de classe est l'un des idiomes les plus important de C++. Si nous respectons cette recette dans la formulation de nos classes, alors les variables créées à partir de nos classes peuvent être affectées et passées en arguments exactement comme toute autre variable d'un type de base[42].

Supposons que nous voulons implémenter la classe `customer`.

```
class customer {

public:
      void x();
      string y();
private:
        string nom;
        string prenom;
        liste_pt<order> orders;
};
```

Pour qu'elle puisse se comporter comme un vrai type de données, l'interface à besoin de bien plus qu'uniquement les opérations spécifiques de manipulation de ses intances.

[42] c'est dire les types int, char, float, ...

La classe `customer` devrait posséder les fonctions membres suivantes :

customer()

Ce constructeur initialise les objets `customer` avec des valeurs par défaut quand aucun contexte d'initialisation n'est fourni. Il est appelé *constructeur par défaut* ou parfois *constructeur de type void*. Pour la classe `customer`, l'action pourrait être de donner des valeurs nulles ou équivalentes à ses membres privés (attributs).

```
customer() : nom("") , prenom("") {;}
```

C'est le constructeur que le compilateur appelle automatiquement. Si le programmeur ne définit pas de constructeur, alors le compilateur fournira le constructeur par défaut à la place du programmeur. Ce constructeur par défaut s'arrangera pour invoquer tous les constructeurs par défaut de tous les membres de cette classe qui sont eux-mêmes des objets de classes[43] . Cependant, si le compilateur fournit le code du constructeur par défaut, alors les membres sans constructeur par défaut ne seront pas initialisés.

customer(const customer&) :

Etant donné un objet customer déjà construit (et sans doute déjà modifié), ce constructeur par défaut devrait construire une copie exacte de cet objet. Le paramètre est une référence à un objet `customer` ; ici, ceci signifie que cette fonction va travailler avec l'objet passé en paramètre (au lieu d'avoir juste un pointeur ou une copie de l'objet).

```
customer(const customer &cus)
                {
                 nom=cus.nom;
                 prenom=cus.prenom;
                 orders=cus.orders;
                }
```

En général, ce constructeur doit copier chaque champ de donnée de l'objet existant à la place correspondante dans le nouvel objet? Nous l'appellerons le *constructeur de copie*, signifiant qu'il initialise un objet par copie d'un objet similaire. Le terme est assez trompeur, car souvent, il réalise plutôt une copie logique ou ce que l'on appelle *copie superficielle* d'un objet[44]. Plus précisément,

---

[43] pour la classe `customer`, il s'agit des objets de classe `string` et `liste_pt`.

[44] Copie superficielle signifie copier un objet sans copier physiquement les objets qu'il référence, alors que copie profonde signifie copier récursivement les objets référencés par l'objet qui est en train d'être copié.

ce constructeur crée des images logiques des objets existants ; toutefois, le terme constructeur de copie est communément utilisé.

customer& operator=(const customer&) :

L'*opérateur d'affectation* est très semblable au constructeur de copie, avec deux différences. D'abord, il a une valeur de retour - le constructeur n'en avait pas. Si une fonction membre n'est pas un constructeur, elle doit fournir une valeur de retour si l'appelant en attend une. Il s'en suit que, pour éviter le surcoût qu'impliquerait la création d'un objet intermédiaire et temporaire, le type qui est le plus souhaitable qu'un opérateur d'affectation retourne, est une référence au type courant.

Ensuite, parce que les contenus de cet objet sont remplacés par de nouveaux contenus (ce qui est la définition même de l'affectation), cette fonction à la responsabilité de libérer les anciens contenus. L'opérateur d'affectation doit explicitement nettoyer toute zone mémoire ont il détruit une référence la pointant, lors de la mise en place d'une nouvelle valeur. Du fait qu'on utilise la forme canonique, on a la garantie qu'un constructeur à été appelé avant l'opérateur `operator=`.

```
customer& customer::operator=(const customer &cus)
{
        if (this==&cus) return *this;
        nom=cus.nom;
        prenom=cus.prenom;
        orders=cus.orders;
        return *this;
}
```

Notons que redéfinir la sémantique de l'opérateur d'affectation est une forme de surcharge : nous donnons à l'opérateur d'affectation une signification différente pour les `customer` que pour les autres types. Par défaut, l'affectation est l'affectation membre par membre des parties de l'objet du côté droit, à leurs homologues de l'objet situé à gauche de l'opérateur d'affectation. Ce que nous faisons ici, c'est "attraper" l'opérateur d'affectation et retirer le contrôle au compilateur pour redéfinir la signification de l'affectation. Nous pouvons donner à l'affectation des significations différentes dans des contextes différents.

~customer() :

Le destructeur doit libérer toutes ressources que l'objet a acquises durant l'exécution de ses constructeurs, ou lors de l'exécution des fonctions membres. les autres fonctions membres de `customer` peuvent avoir conserver la trace d'autres ressources allouées dynamiquement dans les données membres de

customer de telle sorte que le destructeur puisse les trouver et les libérer convenablement.

```
~customer() {;}
```

Dans notre cas, le destructeur ne contient aucune instruction. Ce sont les destructeurs des données membres qui serons déclenchés.

Notons que les destructeurs ne prennent jamais d'arguments. Toute classe a au plus un destructeur.

> Cette forme canonique orthodoxe, pour une classe X est caractérisée par la présence de :
>
> - Un constructeur par défaut (X::X())
> - Un constructeur de copie (X::X(const X&))
> - Un opérateur d'affectation (X& operator=(const X&))
> - Un destructeur (X::~X())

```
class customer {

public:
        customer() : nom("") , prenom("") {;}
        customer(const customer &cus)
                {
                 nom=cus.nom;
                 prenom=cus.prenom;
                 orders=cus.orders;
                }
        ~customer() {;}
        customer& operator=(const customer&);

private:
        string nom;
        string prenom;
        liste_pt<order> orders;
};

customer& customer::operator=(const customer &cus)
{
        if (this==&cus) return *this;
        nom=cus.nom;
        prenom=cus.prenom;
        orders=cus.orders;
        return *this;
}
```

Fig. 6.7 : Exemple complet de la forme canonique orthodoxe pour la classe `customer`

6.5.3 Les méthodes supplémentaires offertes.

Nous savons à présent quelle forme vont prendre les classes d'objets (canonique orthodoxe) et d'où nous allons tirer nos informations (du méta-schéma). Nous savons également la forme physique que prendra notre base de données, et nous avons créé toutes les librairies nécessaires à son implémentation.

En plus de la forme canonique orthodoxe, nous allons offrir à nos classes d'objets :

- des accesseurs à leurs données membres en entrée, et en sortie.
- des opérateurs << et >> surchargés pour l'affichage ou la saisie à l'écran d'une instance de cette classe d'objets.

```
class customer {

//opérateurs << et >> surchargés
friend ostream& operator<<(ostream&,customer&);
friend istream& operator>>(istream&,customer&);

public:
     ...
     //Méthodes d'accès à l'attribut nom
     put_nom(string n) {nom=n;}
     string get_nom() {return nom;}

     //Méthodes d'accès à l'attribut prenom
     put_prenom(string p) {prenom=p;}
     string get_prenom() {return prenom;}

    //Méthodes d'accès à la relation vers la classe d'objets order
     order *get_first_order() {return orders.getfirst();}
     order *get_next_order(order*o) {return orders.getnext(o);}
     order *get_next_order() {return orders.getnext();}
      order *get_previous_order(order*o)
              {return orders.getprevious(o);}
     order *get_previous_order() {return orders.getprevious();}
     order *get_current_order() {return orders.getcurrent();}
     add_order(order *o) {orders.append(*o);}
     delete_order(order *o) {orders.del(o);}

private:
        string nom;
        string prenom;
        liste_pt<order> orders;
};
```

Fig. 6.8 : Exemple des méthodes supplémentaires offertes pour la classe `customer`

6.5.4 Une donnée membre et une méthode supplémentaire pour la gestion de l'héritage.

Nous avons vu lors de la description physique de la base de données que la liste des pointeurs sur les instances d'une classe d'objets générique contient également des pointeurs sur les instances de ses classes spécifiques. Un pointeur déclaré comme étant un pointeur sur une instance d'une classe générique pourrait éventuellement pointer, en réalité, sur une instance d'une de ses classes spécifiques. Il se pourrait que l'on ai besoin de savoir quel est le type réel de l'instance pointée par un tel pointeur. Nous allons pour cela ajouter une donnée membre supplémentaire à la classe d'objets générique de type string[45] qui contient tout simplement le nom de la classe d'objet. Cette donnée sera mise à jour par les constructeurs des classes concernée. Nous avons également ajouté une méthode à la classe générique retournant le type de la classe d'objets concernée (*)[46]. Cet aspect est décrit dans la figure ci-dessous par les caractères en gras.

Nous attirons l'attention du lecteur sur les conséquences de l'héritage sur l'implémentation du générateur de code. En effet vous pouvez remarquer :

- que les données membre de la classe d'objets générique ne sont plus privées (private) mais protégées (protected), car les classes d'objets spécifiques devront avoir accès aux données de leur type générique.
- que les attributs d'une classe spécifiques ne sont pas uniquement ceux déclarés dans la classe, mais également ceux déclarés dans la classe d'objets générique, et cela, de manière récursive. Il faudra en tenir compte dans la génération du constructeur de copie et de l'opérateur d'affectation.
- qu'il en est de même pour les associations.

```
//*****************
//Classe customer
//*****************

class customer {

public:
        customer() : nom(""),prenom(""),type("customer")  {;}
        customer(const customer &cus)
                {
                 nom=cus.nom;
                 prenom=cus.prenom;
                 orders=cus.orders;
                 type=cus.type;
                }
```

[45] Par héritage, elle fera donc également partie des données membres des classes d'objets spécifiques.

[46] Une illustration de l'utilisation de cette méthode peut être trouvée dans l'annexe contenant la totalité du code généré de notre exemple, dans l'implémentation des opérateurs << et >>.

```
        ~customer() {;}
        customer& operator=(const customer&);

        string typeof() {return type;} (*)

protected:
        string nom;
        string prenom;
        string type;
        liste_pt<order> orders;
};


//*****************
//Classe priv_cus
//*****************

class priv_cus : public customer {

public:
        priv_cus() : ntva(0)  {type ="priv_cus";}
        priv_cus(const priv_cus &pri)
                {
                 ntva=pri.ntva;
                 type=pri.type;
                 nom=pri.nom;
                 prenom=pri.prenom;
                 orders=pri.orders;
                }
        ~priv_cus() {;}
        priv_cus& operator=(const priv_cus&);


private:
        int ntva;
};


//*****************
//Classe pub_cus
//*****************

class pub_cus : public customer {

public:
        pub_cus() : commentaire("")  {type ="pub_cus";}
        pub_cus(const pub_cus &pub)
                {
                 commentaire=pub.commentaire;
                 type=pub.type;
                 nom=pub.nom;
                 prenom=pub.prenom;
                 orders=pub.orders;
                }
        ~pub_cus() {;}
        pub_cus& operator=(const pub_cus&);

private:
        string commentaire;
};

//--------------------------- classe customer

customer& customer::operator=(const customer &cus)
{
        if (this==&cus) return *this;
        nom=cus.nom;
        prenom=cus.prenom;
        orders=cus.orders;
```

```
        type=cus.type;
        return *this;
}

//---------------------------- classe priv_cus

priv_cus& priv_cus::operator=(const priv_cus &pri)
{
        if (this==&pri) return *this;
        ntva=pri.ntva;
        type=pri.type;
        nom=pri.nom;
        prenom=pri.prenom;
        orders=pri.orders;
        return *this;
}

//---------------------------- classe pub_cus

pub_cus& pub_cus::operator=(const pub_cus &pub)
{
        if (this==&pub) return *this;
        commentaire=pub.commentaire;
        type=pub.type;
        nom=pub.nom;
        prenom=pub.prenom;
        orders=pub.orders;
        return *this;
}
```

Fig. 6.8 : Illustration de la gestion de l'héritage.

6.5.5 conventions de notation dans le code généré.

- Les listes de pointeurs vers les instances d'une classe d'objets sont appelées l_<nom de la classe d'objets>. Exemple : la liste de pointeurs vers les instances de la classe d'objet `customer` s'appellera **`l_customer`**.
- Les listes de pointeurs matérialisant les relations que possèdent les classes d'objets sont appelées <nom de la classe d'objet>s. Exemple : la liste de pointeurs de la classe d'objet `customer` matérialisant l'association vers des instances de la classe d'objets `order` s'appellera **`orders`**.
- Le nom des variables contenant des instances d'une classe d'objets est constitué des trois premières lettres du nom de la classe d'objets en question. Exemple : une variable contenant une instance de la classe d'objets `customer` s'appellera **`cus`**.

6.6 Le générateur de code C++ .

Résumons ce que nous connaissons et ce dont nous disposons pour développer le générateur de code C++ :

- Nous connaissons la représentation physique de notre base de données orientée objets et de ses types de données. Nous possédons les librairies

nécessaires à leur construction : les classes d'objets `liste_pt, string` et le type `boolean`.

- Nous connaissons également la forme syntaxique des classes qui seront générées puisque nous allons suivre la forme canonique orthodoxe de classe.
- Nous savons que nous devrons greffer, à la forme canonique orthodoxe de classe, premièrement, des méthodes d'accès aux données et aux objets associés, et deuxièmement, des procédures qui surchargerons les opérateurs d'entrée et de sortie.
- Nous devrons également greffer une donnée et une méthode supplémentaire afin de gérer l'héritage.
- Nous possédons également une source d'information facile d'emploi via la librairie `metabase.h`.

Grâce à nos acquis, le générateur ne sera maintenant "plus qu' "un processeur de texte.

Le générateur génére le code lignes par lignes. Un ligne est matérialisée par un string. Au fur et à mesure de la génération, les lignes sont insérées dans une liste de pointeurs vers des strings. Une fois la totalité de code généré, il est écrit dans un fichier texte qui pourra plus tard être compilé.

L'algorithme de génération du code C++ est décrit brièvement dans la figure ci-dessous :

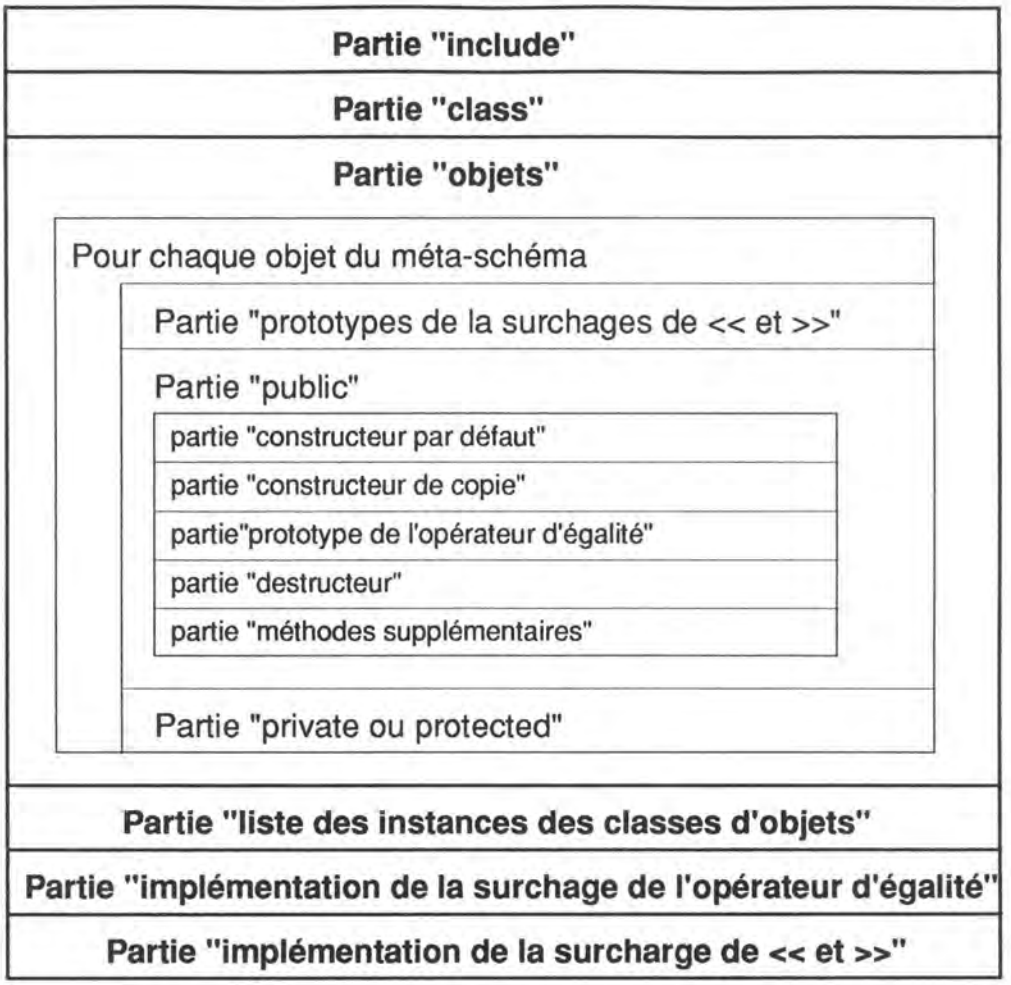

Fig. 6.9 : Algorithme général du générateur.

Dans ce qui suit, nous allons utiliser un méta-langage simple. Il utilise la fonction générer, possède les concepts d'itération, de condition et de variable. Tout texte écrit en gras sera généré tel quel. Si une variable est utilisée, c'est sa valeur qui sera générée. Un autre type de valeur est utilisé : si du texte est entouré par < et >, la valeur représentée sera celle de la sémantique de ce texte. Exemple :

pour chaque classe X
    y = <nom de la classe X>
    générer **class** y

génèrera le texte

```
class line;
class product;
class order;
class customer;
class priv_cus;
class pub_cus;
class brol;
```

Pour chaque partie décrite dans l'algorithme, nous allons exprimer le code généré grâce au méta-langage et donner la valeur du code généré réel sur base de notre exemple classique du client et de ses commandes.

6.6.1 Partie "Include".

C'est la partie du code généré qui inclu les librairies nécessaires à la compilation du code généré. Il s'agit des librairies `iostream.h` du système, `l_pt.cpp`, `string.h` et `boolean.h`.

générer **#include <iostream.h>**
générer **#include "string.h"**
générer **#include "l_pt.cpp"**

Alg. 6.1 : Algorithme de la partie "Include".

```
#include <iostream.h>
#include "string.h"
#include "l_pt.cpp"
```

Fig. 6.10 : Résultat de la génération de la partie "Include".

6.6.2 Partie "Class".

Cette partie de code consiste à déclarer prématurément toutes les classes d'objets sans encore les définir afin de pouvoir utiliser des références à ces classes d'objets sans les avoir déjà définies complètement.

pour chaque classe d'objets X
y=<nom de la classe X>
générer **class** y **;**

Alg. 6.2 : Algorithme de la partie "Class".

```
class line;
class product;
class order;
class customer;
class priv_cus;
class pub_cus;
class brol;
```

Fig. 6.11 : Résultat de la génération de la partie "Class".

6.6.3 Partie "Objets".

Dans la partie "Objet", tous les traitements ci-dessous sont affectués pour chaque classe d'objet du schéma. Dans nos exemples de code généré nous ne placerons que le code concernant les classes d'objets `customer`, `priv_cus` et `pub_cus`.

PARTIE "PROTOTYPES DE LA SURCHARGE DES OPÉRATEURS D'ENTREE << ET DE SORTIE >>" :

générer **friend ostream& operator<<(ostream&,**<nom de la classe>**&);**
générer **friend istream& operator>>(istream&,**<nom de la classe>**&);**

Alg. 6.3 : Algorithme de la partie "Prototypes de la surcharge des opérateurs d'entrée << et de sortie >>.

```
friend ostream& operator<<(ostream&,customer&);
friend istream& operator>>(istream&,customer&);
friend ostream& operator<<(ostream&,pub_cus&);
friend istream& operator>>(istream&,pub_cus&);
friend ostream& operator<<(ostream&,priv_cus&);
friend istream& operator>>(istream&,priv_cus&);
```

Fig. 6.12 : Résultat de la génération de la partie "Prototypes de la surcharge des opérateursd'entrée << et de sortie >>.

PARTIE "PUBLIC" :

- Partie "constructeur par défaut" : Il consiste à initialiser les attributs et les associations avec des valeur "nulles". Pour les attributs, si le type est un string, la valeur "nulle" sera `""` ; si le type est un entier, la valeur "nulle" sera `0` ; enfin, si le type est un booléen, la valeur "nulle" sera `FALSE`. Concernant les associations, les pointeurs seront mis à `0`. En ce qui oncerne la gestion de l'héritage, l'attribut `type` sera mis à jour avec le nom de la classe qui le contient.

```
customer() : nom("") , prenom("") , type("customer")  {;}
priv_cus() : ntva(0)  {type ="priv_cus";}
pub_cus() : commentaire("")  {type ="pub_cus";}
```

Fig. 6.13 : Résultat de la génération de la partie "constructeur par défaut".

- Partie "constructeur de copie" :

X= <nom de la classe>
xxx = <3 premiers caractères de X>
générer X (**const** X &xxx)
générer {

pour chaque attribut de X
    générer <nom de l'attribut>=xxx.<nom de l'attribut>;

pour chaque association de X
        Y = <nom de la classe d'objets destination>
        yyy = <3 premiers caractères de Y>
        si connectivité maximale = 1
                générer yyy=xxx.yyy ;
        sinon
                générer Ys=xxx.Ys ;

générer }

Alg. 6.4 : Algorithme de la partie "constructeur de copie".

```
customer(const customer &cus)
                {
                 nom=cus.nom;
                 prenom=cus.prenom;
                 orders=cus.orders;
                 type=cus.type;
                }

priv_cus(const priv_cus &pri)
                {
                 ntva=pri.ntva;
                 type=pri.type;
                 nom=pri.nom;
                 prenom=pri.prenom;
                 orders=pri.orders;
                }

pub_cus(const pub_cus &pub)
                {
                 commentaire=pub.commentaire;
                 type=pub.type;
                 type=pub.type;
                 nom=pub.nom;
                 prenom=pub.prenom;
                 orders=pub.orders;
                }
```

Fig. 6.14 : Résultat de la génération de la partie "constructeur de copie".

- Partie "prototype de l'operateur d'égalité" :

générer <nom de la classe>**& operator=(const** <nom de la classe>**&);**

Alg. 6.5 : Algorithme de la partie "prototype de l'opérateur d'égalité".

```
customer& operator=(const customer&);
priv_cus& operator=(const priv_cus&);
pub_cus& operator=(const pub_cus&);
```

Fig. 6.15 : Résultat de la génération de la partie "prototype de l'opérateur d'égalité".

- Partie "destructeur"

générer **~**<nom de la classe>() **{;}**

Alg. 6.6 : Algorithme de la partie "destructeur".

```
~customer() {;}
~priv_cus() {;}
~pub_cus() {;}
```

Fig. 6.15 : Résultat de la génération de la partie "destructeur".

- Partie "méthodes supplémentaires" :

si X est un sur_type
générér **string typeof() {return type;}**

pour chaque attribut de la classe d'objets X
T = <nom de l'attribut>
N = <type de l'attribut>
n = <premier caractère de N>
générer T **get_**N() **{return** N;**}**
générer **put_**N(T n){N=n;}

pour chaque classe d'objets associée à X
Y = <nom de la classe d'objets associée>
y = <premier caractère de Y>
si connectivité maximale > 1
générer Y ***get_first_**Y() **{return** Ys.**getfirst**();**}**
générer Y ***get_next_**Y() **{return** Ys.**getnext**();**}**
générer Y ***get_next_**Y(Y *y) **{return** Ys.**getnext**(y);**}**
générer Y ***get_previous_**Y() **{return** Ys.**getprevious**();**}**

générer Y ***get_previous**_Y(Y *y) **{return** Ys.**getprevious**(y);**}**
générer Y ***get_current**_Y() **{return** Ys.**getcurrent**();**}**

générer **add**_Y(Y* y) {Ys.**insert**(y);**}**
générer **delete**_Y(Y* y) {Ys.**del**(y);**}**
sinon
yyy = <3 premiers caractères de Y>
générer **put**_Y(Y *y){yyy=y;**}**
générer Y* **get**_Y() **{return** yyy;**}**

Alg 6.7 : Algorithme de la partie "méthodes supplémentaires".

```
//Méthodes d'accès à l'attribut nom
put_nom(string n) {nom=n;}
string get_nom() {return nom;}

//Méthodes d'accès à l'attribut prenom
put_prenom(string p) {prenom=p;}
string get_prenom() {return prenom;}

//Méthodes d'accès à la relation vers l'objet order
order *get_first_order() {return orders.getfirst();}
order *get_next_order(order*o) {return orders.getnext(o);}
order *get_next_order() {return orders.getnext();}
order *get_previous_order(order*o) {return orders.getprevious(o);}
order *get_previous_order() {return orders.getprevious();}
order *get_current_order() {return orders.getcurrent();}
add_order(order *o) {orders.append(*o);}
delete_order(order *o) {orders.del(o);}

//Méthodes d'accès à l'attribut ntva
put_ntva(int n) {ntva=n;}
int get_ntva() {return ntva;}

//Méthodes d'accès à l'attribut commentaire
put_commentaire(string c) {commentaire=c;}
string get_commentaire() {return commentaire;}
```

Fig. 6.16 : Résultat de la génération de la partie "méthodes supplémentaires".

PARTIE "PRIVATE OU PROTECTED" :

elle consiste en l'énumération des données membres de la classe qui seront constituées des attributs et des associations de l'objets. Cette partie sera "private" ou public selon que la classe soit un sur-type ou non.

6.6.4 Partie "Liste des instances des classes d'objets".

Dans cette partie, on déclare les listes de pointeurs cers les instances des classes d'objets qui constituent la base de données.

Pour chaque objet X
générer **liste_pt**<nom de la classe> **l_**<nom de la classe>;

Alg. 6.8 : Algorithme de la partie "Listes des instances des classes d'objets".

```
liste_pt<line> l_line;
liste_pt<product> l_product;
liste_pt<order> l_order;
liste_pt<customer> l_customer;
liste_pt<priv_cus> l_priv_cus;
liste_pt<pub_cus> l_pub_cus;
liste_pt<brol> l_brol;
```

Fig. 6.17 : Résultat de la génération e de la partie "Listes des instances des classes d'objets".

6.6.5 PARTIE " Implémentation de la surcharge de l'opérateur d'égalité"

Pour chaque classe d'objets X
xxx= <3 premiers caractères du nom de classe X>
générer <nom de la classe>**& operator=**(<nom de la classe> **&**xxx**);**
générer {
générer **if (this==&**xxx**) return *this;**
pour chaque attribut de X
générer <nom de l'attribut>=xxx.<nom de l'attribut>;
pour chaque association de X
Y = <nom de la classe d'objets associée>
si connectivité maximale = 1
générer Y=xxx.Y;
sinon
générer Ys=xxx.Ys;
générer **return *this;**
générer }

Alg. 6.9 : Algorithme de la partie "Implémentation de la surcharge de l'opérateur d'égalité"

```
//---------------------------- classe customer

customer& customer::operator=(const customer &cus)
{
        if (this==&cus) return *this;
        nom=cus.nom;
        prenom=cus.prenom;
                orders=cus.orders;
                type=cus.type;
        return *this;
}
```

```
//---------------------------- classe priv_cus

priv_cus& priv_cus::operator=(const priv_cus &pri)
{
        if (this==&pri) return *this;
        ntva=pri.ntva;
        type=pri.type;
        nom=pri.nom;
        prenom=pri.prenom;
        orders=pri.orders;
        return *this;
}

//---------------------------- classe pub_cus

pub_cus& pub_cus::operator=(const pub_cus &pub)
{
        if (this==&pub) return *this;
        commentaire=pub.commentaire;
        type=pub.type;
        nom=pub.nom;
        prenom=pub.prenom;
        orders=pub.orders;
        return *this;
}
```

Fig 6.18 : Résultat de la génération de la partie "Implémentation de la surcharge de l'opérateur d'égalité".

6.6.6 PARTIE "Implémentation de la surcharge de << et >>".

Il s'agit uniquement d'appeler les opérateurs << ou >> surchargés des attributs membres de l'objet. Une remarque concernant la gestion de l'héritage : puisque la liste des instances d'une classe d'objet générique contient à la fois les instances de sa classe d'objets et de ses classes spécifiques, il faudra vérifier le type réel de l'instance via la méthode `typeof()` et appeler l'opérateur << ou >> surchargé adéquat.

```
//******************** classe customer

ostream& operator<< (ostream &os,customer &cus)
{
 if (cus.typeof() == "priv_cus") os << (priv_cus&)cus;
 else  if (cus.typeof() == "pub_cus") os << (pub_cus&)cus;
 else
        {
         os << cus.nom;
         os << cus.prenom;
        }
 return os;
}

istream& operator>> (istream &is,customer &cus)
{
 if (cus.typeof() == "priv_cus") is >> cus;
 else  if (cus.typeof() == "pub_cus") is >> cus;
 else
        {
```

```
        cout<<"nom customer?"; is >> cus.nom;
        cout<<"prenom customer?"; is >> cus.prenom;
        }
 return is;
}

//******************** classe priv_cus

ostream& operator<< (ostream &os,priv_cus &pri)
{
         os << pri.ntva;
         os << pri.nom;
         os << pri.prenom;
 return os;
}

istream& operator>> (istream &is,priv_cus &pri)
{
        cout<<"ntva priv_cus?"; is >> pri.ntva;
        cout<<"nom priv_cus?"; is >> pri.nom;
        cout<<"prenom priv_cus?"; is >> pri.prenom;
 return is;
}

//******************** classe pub_cus

ostream& operator<< (ostream &os,pub_cus &pub)
{
         os << pub.commentaire;
         os << pub.nom;
         os << pub.prenom;
 return os;
}

istream& operator>> (istream &is,pub_cus &pub)
{
        cout<<"commentaire pub_cus?"; is >> pub.commentaire;
        cout<<"nom pub_cus?"; is >> pub.nom;
        cout<<"prenom pub_cus?"; is >> pub.prenom;
 return is;
}
```

Fig. 6.19 : Résultat de la génération de la partie "Implémentation de la surcharge de << et >>".

6.7 Utilisation du code généré :

6.7.1 La compilation.

Une fois le fichier généré, il suffit de créer un projet sous Borland C++ et d'y insérer les fichiers suivants : l_pt.cpp, listestr.cpp, string.cpp et le fichier résultat de la génération de code C++.

6.7.2 Le développement d'une application.

L'utilisateur de la base de données dispose de toutes les méthodes nécessaire au développement d'une application complète utilisant cette base de données.

Il peut, si il le désire, construire des méthodes spécifiques concernant une classe d'objets. Exemple : on peut construire pour les instances de la classe d'objets `customer` la procédure `customer_with_name_begin_with_B()` qui retourne une liste d'instances de la classe `customer` dont le nom commence par un B :

```
customer_with_name_begin_with_B(liste_pt<customer> &li)
{
      customer *cus = l_customer.getfirst();
      while (cus !=0)
            {
                  string nom=cu->get_name();
                  if nom(0,1)="B" li.append(cus);
                  cus=l_customer.getnext();
            }
}
```

Si on veut que cette procédure ne retourne que les client dont le nom commence par un B mais ayant passé au moins 10 commandes :

```
customer_with_name_begin_with_B(liste_pt<customer> &li)
{
      customer *cus = l_customer.getfirst();
      while (cus !=0)
            {
                  if (cus->orders.length() >= 10)
                        {
                              string nom=cus->get_name();
                              if nom(0,1)="B" li.append(cus);
                        }
                  cus=l_customer.getnext();
            }
}
```

Notons que cette procédure est valable quelle que soit le type de client (il peut être un client public ou un client privé). L'utilsateur peut éventuellement prévoir des traitement spécifiques par la méthode get_name(). Il faudra alors définir cette méthode comme étant virtuelle dans la classe générique et la redéfinir dans les classes spécifiques.

CONCLUSION

Dans un premier temps, nous avons analysé et essayé de comprendre les objectifs d'une approche "orientée objet" de la programmation. Nous avons parcouru les possibilités que devraient offrir les langages de programmation en général et les langages de programmation orientés objets en particulier. Pour chaque objectif atteint par une programmation orientée objets, nous avons montré brièvement lesquels le sont réellement par le langage C++, ce pourquoi il a été choisi dans le développement de ce mémoire.

Une fois les objectifs posés, nous avons parcouru les concepts fondamentaux de la programmation objet. Chaque concept a été expliqué et illustré, si c'était possible, par un exemple C++.

Nous avons tenté ensuite de dresser une typologie des méthodologies orientées objets : les unitaires et les ternaires. Pour chaque catégorie, nous avons exposé brièvement une méthodologie représentative : respectivement celles de Coad & Yourdon et de Rumbaugh.

Nous avons ensuite défini les caractéristiques que devraient recouvrir une base de données orientée objets. Il est clair que notre base de donnée ne répond pas à toutes les caractéristiques obligatoires citées au chapitre 3. Elle répond cependant à 8 de ces caractéristiques obligatoires :

1. les objets complexes : le fait d'avoir choisi la forme canonique orthodoxe de classe pour former nos classes nous permet de construire n'importe quel objet complexe. Ces objets complexes pourront être manipulés de la même manière que les objets prédéfinis du système.
2. l'identité de l'objet
3. l'encapsulation : l'accès aux données ne se fait pas directement, on doit passer par une méthode. Cependant, l'encapsulation n'est pas la plus forte possible puisque les données restent visibles.
4. les types et classes
5. la hiérarchie de classes ou de types
6. la surcharge et liaison dynamique
7. l'extensibilité
8. les facilités de requête adéquate

Après avoir pris connaissance de l'approche orientée objets, des méthodologies existantes et des caractéristiques des systèmes de gestion de bases de données orientées objets, nous avons proposé notre propre modèle de représentation des données en fonction des concepts dont nous aurons besoin dans la conception de notre SGBDOO. Si nous avons choisi notre propre représentation des données, c'est que les méthodologies présentées précédemment sont trop complètes : elles soulignent tous les aspects de l'analyse et du design d'un problème alors que seules les données nous intéressent ici. De plus, elles ont tendance à multiplier inutilement le nombre de concepts utilisés, alors qu'un nombre limité de ces concepts suffisent largement à représenter toutes les caractéristiques des données que nous avons à manipuler.

Dans le dernier chapitre, après avoir présenté l'atelier logiciel TRAMIS, nous nous sommes attardés sur l'implémentation finale de notre base de données. Tout d'abord, nous avons fait une description physique de notre base de données. Le choix de la représentation physique n'a pas été facile, surtout en ce qui concerne les clusters : le fait que la liste des intances d'une classe d'objets possède non seulement ses propres intances, mais aussi celles de ses classes d'objets spécifiques n'est pas forcément évident. A partir de cette description physique, nous avons présenté les différentes librairies qui ont été implémentées pour la supporter.

Ensuite nous nous sommes intéressés à la forme que prendra le code C++ généré : nous avons vu la nécessité d'utiliser la forme de classe canonique orthodoxe à laquelle nous avons greffé certaines méthodes et certains attributs afin de répondre aux spécifications de notre base de données orientée objets. Nous avons terminé en décrivant succinctement la structure générale de l'algorithme du générateur C++.

Le code généré sera suffisant pour construire une application complète utilisant une base de données orientée objets. Une faiblesse de notre système est qu'en l'absence d'un browser d'objet, ou d'une interface, l'intégrité de la base de données doit être gérée par l'utilisateur : il ne pourra, par exemple, oublier que s'il crée une nouvelle instance d'une classe d'objets, il devra la faire pointer par la liste des instances de sa classe d'objets. De plus si cette classe d'objets est une classe d'objets spécifique, il devra aussi la faire pointer par la liste des instances de la classe d'objet générique. Cependant, une interface de ce type pourra être implémentée par l'utilisateur si il le désire. Il en est de même pour la gestion des identifiant : elle n'est pas implémentée dans le code généré, mais l'utilisateur dispose de toutes les méthodes nécessaires pour l'implémenter, sans ce que cette gestion soit visible.

BIBLIOGRAPHIE

- [OODB1], "The Object-Oriented Database System Manifesto", M. Atkinson (University of Glasgow), D. DeWitt (University of Wisconsin), D. Maier (Oregon Graduate Center), F. Bancilhon (Altaïr), K. Dittrich (University of Zurich), S. Zdonik (Brow University), 19 August 1989.

- [OODB2], SCOOP Europe 1991, "Building Applications with an Object DBMS", T. Atwood, Session T3, Oktober 1991.

- [OODB3], "Bases de données : des systèmes relationnels aux systèmes à objets", C. Delobel, C. Lécluse, Ph. Richard, InterEditions, Paris, 1991.

- [MOO1], A tutorial for OOPSLA '92, "A comparison of Object-Oriented Analysis and Design Methods", M. Fowler, July 1992.

- [MOO2], SCOOP Europe 1991, "Coherent Model for Object-Oriented Analysis", F. Hayes, D. Colmeman, Session TH7, 1991.

- [MOO3], SCOOP Europe 1991, "Analysis of Current Object-Oriented Methods", E. Colbert, Session F4, October 1991.

- [MOO4], "Object-Oriented Modeling & Design", J. Rumbaugh, Englewood Cliffs, NJ, Prentice-Hall, Inc., 1991.

- [MOO5], "Object-Oriented Analysis", P. Coad & E. Yourdon, Prentice-Hall, Englewood Cliffs, NJ, 1990.

- [MOO6], "Object-Oriented Analysis (second edition)", P. Coad & E. Yourdon, Prentice-Hall, Englewood Cliffs, NJ, 1991.

- [MOO7], "Object-Oriented Design", P. Coad & E. Yourdon, Prentice-Hall, Englewood Cliffs, NJ, 1991.

- [MOO8], "Interoperation of Object-Oriented Applications with conventionnal IT", Object EXPO Europe, session W4, July 1992.

- [OO1], "Synthesis : Object-Oriented Systems Development", v2.3 1989-1990-1991 Wayland Sytems Inc.

- [C++1], "Programmation avancée en C++ : Styles et Idiomes", James O. Coplien, Editions Addison Wesley, France, 1992.

- [C++2], SCOOP Europe 1991, "C++ Strategy & Tactics", R. Murray (AT&T Bell Laboratories), Session T4, 17 October 1991.

- [C++3], SCOOP Europe 1991, "Advanced C++ Programming Techniques", D. Bern, 1 November 1991.

- [C++4], "The C++ programming language", B. Soustrup, Addison Wesley, 1986.

- [ER1], "Conception assistée des systèmes d'information (deuxième édition)", F. Bodart, Y. Pigneur, Masson, 1989.

- [TRAMIS1], "TRAMIS : a transformation-based database CASE tool", Proceeding of th 5th international conference on Software engeneering and it's application, Toulouse (FRANCE), December 1992.

ANNEXE : Le code généré pour l'exemple classique

```
#include <iostream.h>
#include "string.h"
#include "1_pt.cpp"
#include "metabase.h"


class line;
class product;
class order;
class customer;
class priv_cus;
class pub_cus;

//*****************
//Classe line
//*****************

class line {

friend ostream& operator<<(ostream&,line&);
friend istream& operator>>(istream&,line&);

public:
        line() : quantite(0) , libelle("") , ord(0) ,pro(0) {;}
        line(const line &lin)
                {
                 quantite=lin.quantite;
                 libelle=lin.libelle;
                 ord=lin.ord;
                 pro=lin.pro;
                }
        ~line() {;}
        line& operator=(const line&);

        //Méthodes d'accès à l'attribut quantite
        put_quantite(int q) {quantite=q;}
        int get_quantite() {return quantite;}

        //Méthodes d'accès à l'attribut libelle
        put_libelle(string l) {libelle=l;}
        string get_libelle() {return libelle;}

        //Méthodes d'accès à la relation vers l'objet order
        order* get_order() {return ord;}
        put_order(order *o) {ord=o;}

        //Méthodes d'accès à la relation vers l'objet product
        product* get_product() {return pro;}
        put_product(product *p) {pro=p;}

private:
        int quantite;
        string libelle;
        order *ord;
        product *pro;
};
```

```
//*****************
//Classe product
//*****************

class product {

friend ostream& operator<<(ostream&,product&);
friend istream& operator>>(istream&,product&);

public:
        product() : nom("")  {;}
        product(const product &pro)
                {
                 nom=pro.nom;
                 lines=pro.lines;
                }
        ~product() {;}
        product& operator=(const product&);

        //Méthodes d'accès à l'attribut nom
        put_nom(string n) {nom=n;}
        string get_nom() {return nom;}

        //Méthodes d'accès à la relation vers l'objet line
        line *get_first_line() {return lines.getfirst();}
        line *get_next_line(line*l) {return lines.getnext(l);}
        line *get_next_line() {return lines.getnext();}
        line *get_previous_line(line*l) {return lines.getprevious(l);}
        line *get_previous_line() {return lines.getprevious();}
        line *get_current_line() {return lines.getcurrent();}
        add_line(line *l) {lines.append(*l);}
        delete_line(line *l) {lines.del(l);}

private:
        string nom;
        liste_pt<line> lines;
);



//*****************
//Classe order
//*****************

class order {

friend ostream& operator<<(ostream&,order&);
friend istream& operator>>(istream&,order&);

public:
        order() : numero(0) , libelle("") , cus(0) {;}
        order(const order &ord)
                {
                 numero=ord.numero;
                 libelle=ord.libelle;
                 cus=ord.cus;
                 lines=ord.lines;
                }
        ~order() {;}
        order& operator=(const order&);

        //Méthodes d'accès à l'attribut numero
        put_numero(int n) {numero=n;}
```

```
        int get_numero() {return numero;}

        //Méthodes d'accès à l'attribut libelle
        put_libelle(string l) {libelle=l;}
        string get_libelle() {return libelle;}

        //Méthodes d'accès à la relation vers l'objet customer
        customer* get_customer() {return cus;}
        put_customer(customer *c) {cus=c;}

        //Méthodes d'accès à la relation vers l'objet line
        line *get_first_line() {return lines.getfirst();}
        line *get_next_line(line*l) {return lines.getnext(l);}
        line *get_next_line() {return lines.getnext();}
        line *get_previous_line(line*l) {return lines.getprevious(l);}
        line *get_previous_line() {return lines.getprevious();}
        line *get_current_line() {return lines.getcurrent();}
        add_line(line *l) {lines.append(*l);}
        delete_line(line *l) {lines.del(l);}

private:
        int numero;
        string libelle;
        customer *cus;
        liste_pt<line> lines;
};


//*****************
//Classe customer
//*****************

class customer {

friend ostream& operator<<(ostream&,customer&);
friend istream& operator>>(istream&,customer&);

public:
        customer() : nom("") , prenom("") , type("customer")  {;}
        customer(const customer &cus)
                {
                 nom=cus.nom;
                 prenom=cus.prenom;
                 orders=cus.orders;
                 type=cus.type;
                }
        ~customer() {;}
        customer& operator=(const customer&);
        string typeof() {return type;}

        //Méthodes d'accès à l'attribut nom
        put_nom(string n) {nom=n;}
        string get_nom() {return nom;}

        //Méthodes d'accès à l'attribut prenom
        put_prenom(string p) {prenom=p;}
        string get_prenom() {return prenom;}

        //Méthodes d'accès à la relation vers l'objet order
        order *get_first_order() {return orders.getfirst();}
        order *get_next_order(order*o) {return orders.getnext(o);}
        order *get_next_order() {return orders.getnext();}
        order *get_previous_order(order*o) {return orders.getprevious(o);}
```

```
        order *get_previous_order() {return orders.getprevious();}
        order *get_current_order() {return orders.getcurrent();}
        add_order(order *o) {orders.append(*o);}
        delete_order(order *o) {orders.del(o);}

protected:
        string nom;
        string prenom;
        string type;
        liste_pt<order> orders;
};


//*****************
//Classe priv_cus
//*****************

class priv_cus : public customer {

friend ostream& operator<<(ostream&,priv_cus&);
friend istream& operator>>(istream&,priv_cus&);

public:
        priv_cus() : ntva(0)  {type ="priv_cus";}
        priv_cus(const priv_cus &pri)
                {
                 ntva=pri.ntva;
                 type=pri.type;
                 nom=pri.nom;
                 prenom=pri.prenom;
                 orders=pri.orders;
                }
        ~priv_cus() {;}
        priv_cus& operator=(const priv_cus&);

        //Méthodes d'accès à l'attribut ntva
        put_ntva(int n) {ntva=n;}
        int get_ntva() {return ntva;}

private:
        int ntva;
};


//*****************
//Classe pub_cus
//*****************

class pub_cus : public customer {

friend ostream& operator<<(ostream&,pub_cus&);
friend istream& operator>>(istream&,pub_cus&);

public:
        pub_cus() : commentaire("")  {type ="pub_cus";}
        pub_cus(const pub_cus &pub)
                {
                 commentaire=pub.commentaire;
                 type=pub.type;
                 type=pub.type;
                 nom=pub.nom;
                 prenom=pub.prenom;
```

```
                orders=pub.orders;
               }
        ~pub_cus() {;}
        pub_cus& operator=(const pub_cus&);

        //Méthodes d'accès à l'attribut commentaire
        put_commentaire(string c) {commentaire=c;}
        string get_commentaire() {return commentaire;}

protected:
        string commentaire;
};

liste_pt<line> l_line;
liste_pt<product> l_product;
liste_pt<order> l_order;
liste_pt<customer> l_customer;
liste_pt<priv_cus> l_priv_cus;
liste_pt<pub_cus> l_pub_cus;


//********************
//Méthodes des classes
//********************


//---------------------------- classe line

line& line::operator=(const line &lin)
{
        if (this==&lin) return *this;
        quantite=lin.quantite;
        libelle=lin.libelle;
        ord=lin.ord;
        pro=lin.pro;
        return *this;
}

//---------------------------- classe product

product& product::operator=(const product &pro)
{
        if (this==&pro) return *this;
        nom=pro.nom;
        lines=pro.lines;
        return *this;
}

//---------------------------- classe order

order& order::operator=(const order &ord)
{
        if (this==&ord) return *this;
        numero=ord.numero;
        libelle=ord.libelle;
        cus=ord.cus;
        lines=ord.lines;
        return *this;
}

//---------------------------- classe customer

customer& customer::operator=(const customer &cus)
{
```

```
        if (this==&cus) return *this;
        nom=cus.nom;
        prenom=cus.prenom;
        orders=cus.orders;
        type=cus.type;
        return *this;
}

//---------------------------- classe priv_cus

priv_cus& priv_cus::operator=(const priv_cus &pri)
{
        if (this==&pri) return *this;
        ntva=pri.ntva;
        type=pri.type;
        nom=pri.nom;
        prenom=pri.prenom;
        orders=pri.orders;
        return *this;
}

//---------------------------- classe pub_cus

pub_cus& pub_cus::operator=(const pub_cus &pub)
{
        if (this==&pub) return *this;
        commentaire=pub.commentaire;
        type=pub.type;
        nom=pub.nom;
        prenom=pub.prenom;
        orders=pub.orders;
        return *this;
}

//------------------  PROCEDURES GENERALES ---------------------------

//******************** classe line

ostream& operator<< (ostream &os,line &lin)
{
         os << lin.quantite;
         os << lin.libelle;
         return os;
}

istream& operator>> (istream &is,line &lin)
{
        cout<<"quantite line?"; is >> lin.quantite;
        cout<<"libelle line?"; is >> lin.libelle;
        return is;
}

//******************** classe product

ostream& operator<< (ostream &os,product &pro)
{
         os << pro.nom;
         return os;
}

istream& operator>> (istream &is,product &pro)
{
        cout<<"nom product?"; is >> pro.nom;
        return is;
```

```
}

//******************** classe order

ostream& operator<< (ostream &os,order &ord)
{
        os << ord.numero;
        os << ord.libelle;
 return os;
}

istream& operator>> (istream &is,order &ord)
{
       cout<<"numero order?"; is >> ord.numero;
       cout<<"libelle order?"; is >> ord.libelle;
 return is;
}

//******************** classe customer

ostream& operator<< (ostream &os,customer &cus)
{
 if (cus.typeof() == "priv_cus") os << (priv_cus&)cus;
 else  if (cus.typeof() == "pub_cus") os << (pub_cus&)cus;
 else
       {
        os << cus.nom;
        os << cus.prenom;
       }
 return os;
}

istream& operator>> (istream &is,customer &cus)
{
 if (cus.typeof() == "priv_cus") is >> cus;
 else  if (cus.typeof() == "pub_cus") is >> cus;
 else
       {
       cout<<"nom customer?"; is >> cus.nom;
       cout<<"prenom customer?"; is >> cus.prenom;
       }
 return is;
}

//******************** classe priv_cus

ostream& operator<< (ostream &os,priv_cus &pri)
{
        os << pri.ntva;
        os << pri.nom;
        os << pri.prenom;
        return os;
}

istream& operator>> (istream &is,priv_cus &pri)
{
       cout<<"ntva priv_cus?"; is >> pri.ntva;
       cout<<"nom priv_cus?"; is >> pri.nom;
       cout<<"prenom priv_cus?"; is >> pri.prenom;
       return is;
}

//******************** classe pub_cus
```

```
ostream& operator<< (ostream &os,pub_cus &pub)
{
         os << pub.commentaire;
         os << pub.nom;
         os << pub.prenom;
          return os;
}

istream& operator>> (istream &is,pub_cus &pub)
{
        cout<<"commentaire pub_cus?"; is >> pub.commentaire;
        cout<<"nom pub_cus?"; is >> pub.nom;
        cout<<"prenom pub_cus?"; is >> pub.prenom;
        return is;
}
```