Université de Montréal

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

« Principes et implantation de vues dans les langages Orientés-objet »

Par

Joumana Dargham

Thèse présentée à la faculté des études supérieures
en vue de l'obtention du grade de
Philosophiae Doctor (Ph.D.)
en informatique

Septembre 2001

Université de Montréal
Faculté des études supérieures

Cette thèse, intitulée

Principes et implantation de vues dans les langages Orientés-objet

et
présentée par

Joumana Dargham

a été évaluée par un jury composé des personnes suivante :

Rachida Dssouli
(directrice de recherche)

Hafedh Mili
(co-directeur de recherche)

François Pachet
(examinateur externe)

Houari Sharaoui
(membre du jury)

Jean Vaucher
(président du jury)

(représentant du doyen)

Thèse acceptée le : 2 novembre 2001

Sommaire

Dans un système d'information d'entreprise, les même entités jouent divers rôles fonctionnels. Ces rôles sont destinés à différents usages et à différents usagers. Du fait que leur implantation requière souvent des compétences différentes, ils sont aussi souvent développés par diverses organisations sous forme d'applications indépendantes et difficilement intégrables. Dans cette recherche, je me suis intéressée au problème du développement, de la réutilisation et de l'entretien, et du déploiement séparé de ces rôles fonctionnels.

Traditionnellement, ce problème est traité en bases de données avec la notion de *vues* par lesquelles on définit des *projections* sur le modèle de données, mais les vraies données restent centralisées dans un seul modèle conceptuel, même si différentes applications n'en voient qu'une partie. En représentation de connaissances, cette question a souvent été traitée par une *spécialisation* de la relation d'héritage [Haut 86], [Carr 89]. Dans les langages *orientés-objet*, on a d'abord utilisé les *interfaces*, comme des *filtres*, sur toutes les fonctionnalités offertes par une classe donnée, mais toutes les données et fonctions d'une classe existent en même temps dans la classe, et leur intégration doit être préplanifiée [Shil 89], [Marc 94]. Harrison et Ossher ont proposé la notion de programmation par composition de *sujets*, où un *sujet* représente une tranche compilable, et possiblement exécutable, d'une application *orientée-objet* [Harr 93]. La composition de *sujets* exige que les applications satisfassent certaines règles de programmation [Ossh 95], et on ne peut ajouter ou retirer des *sujets* dynamiquement. Kiczales et al., de Xerox PARC [Kicz 97], ont introduit la programmation par *aspects*, où un *aspect* est un fragment de logiciel, *orthogonal* aux notions de *classe* et de *méthode*. Tout comme la composition de *sujets*, l'introduction ou l'« injection » d'*aspects* se fait au moment de la compilation, et donc, à l'exécution, tous les *aspects* sont présents dans les classes.

Je propose une nouvelle approche, appelée *programmation par vues*, qui se distingue des approches existantes de deux façons :

- Les aspects fonctionnels des objets, appelés *vues,* peuvent être ajoutés et retirés à un objet durant l'exécution, permettant à l'objet de changer de comportement durant sa durée de vie.
- Les aspects fonctionnels des objets peuvent être conçus comme l'instantiation d'une fonctionnalité générique, que nous appelons *point de vue*, pour une classe du domaine d'application (c.-à-d. objet métier). Le *point de vue* incarne un *rôle* pouvant être joué par des objets de types différents; la *vue* correspondante est l'interprétation de ce rôle par un objet métier donné.

J'ai défini un modèle formel de définition, d'extension et d'instantiation de *point de vue*, et développé des outils logiciels pour les implanter pour C++.

Le chapitre 1 présente la problématique, les principes de cette approche et un survol de la thèse. Le chapitre 2 contient une revue de la littérature. Le chapitre 3 présente les principes de l'approche et en décrit l'utilisation. Les modèles formels sont présentés dans le chapitre 4. Les chapitres 5 et 6 sont consacrés à l'analyse et à la présentation des outils de support de vues développés. Le chapitre 7 présente une étude de cas. La conclusion est présentée dans le chapitre 8.

Table des matières

Liste des figures

à Élie

à Karim et à Joe

à mes Parents, mes frères et soeurs

à la mémoire de ma soeur Najat

Remerciements

Je tiens à exprimer mes plus vifs remerciements à Hafedh Mili, professeur à l'Université du Québec à Montréal, pour avoir accepté de diriger mes travaux de recherche. Tout au long de ce travail, il a su me guider et m'a fait bénéficier de ses pertinents conseils.

Je remercie Rachida Dssouli, ma directrice de recherche, pour ses commentaires et les discussions que nous avons tenues ensemble et qui m'ont aidée à continuer et à mener à bien ce travail.

Les travaux présentés dans cette thèse s'étalent sur plusieurs années au cours desquelles j'ai vécu des hauts et des bas, et grâce à la présence d'amis j'ai pu tenir le coup et terminer.

Je remercie d'abord mes amis de l'Université de Montréal avec qui j'ai commencé mes études de doctorat et avec lesquels j'ai passé les examens pré-doctoraux. C'est grâce à eux que je garde les meilleurs souvenirs de mes premières années d'études. Merci aussi aux professeurs du département d'informatique qui m'ont permis de discuter avec eux de mes craintes et de mes soucis à l'égard de mes travaux.

Je remercie également mes amis du groupe LARC qui ont rendu le travail plus agréable et les longues journées plus belles et moins fatigantes. Merci Sonia pour les beaux moments que nous avons passés ensemble, merci Saleh pour les discussions et les réflexions, merci à Hamid et à tous les autres membres du groupe. Merci à tous ceux qui étaient là quand j'ai eu besoin d'eux.

Je voudrais remercier profondément ma famille, mes parents, mes frères et mes soeurs pour la confiance qu'ils m'ont donnée et pour leur soutient constant. Je crois qu'ils ont tous participé à la réalisation de ce travail.

Enfin, un grand merci à Karim qui est venu ajouter plus de vie à ma vie, qui a supporté mon absence et dont les sourires m'ont parfois permis d'oublier le stress et de bien continuer. Merci à Joe qui a supporté tout le stress de la finalisation du travail. À Élie,

merci pour tout. Merci pour sa présence, son soutient, son amour et tant d'autres choses.

Chapitre I

Introduction

La crise du logiciel, annoncée par Doug McIlroy lors de la conférence de l'OTAN de 1968, constitue maintenant un état permanent de l'industrie. Même si le développement des logiciels demeure aujourd'hui une activité relativement artisanale, des progrès importants au chapitre des méthodes et des outils de développement nous rapprochent davantage de la vision de McIlroy, c'est-à-dire d'une industrie basée plutôt sur l'assemblage de composants fonctionnels commerciaux que sur l'usinage individuel des milliers ou des millions de pièces dont est formé un programme. Plusieurs obstacles ont empêché ou retardé l'émergence de cette industrie, tant d'ordre technique que d'ordre organisationnel, voire même économique. Notre travail s'inscrit dans le cadre de la problématique technique de réutilisation ayant comme objectif la réduction de complexité pour le développement et le déploiement de gros systèmes logiciels.

Pour qu'un développement par assemblage de composants puisse se faire, il nous faut avoir accès à de bons composants et à de bons mécanismes d'assemblage. Un bon composant en est un qui est à la fois «utile», c'est-à-dire qui répond à un besoin fonctionnel fréquent et «utilisable», c'est-à-dire que l'on peut facilement intégrer dans une application. Un bon mécanisme d'assemblage permet un assemblage i) qui demande peu d'efforts, ii) qui n'exige pas une connaissance approfondie des composants assemblés et iii) qui puisse se faire aussi tard que possible dans le développement. La réutilisabilité des composants ainsi que la qualité du mécanisme d'assemblage dépendent d'une bonne modularisation des composants réutilisables. Une telle modularisation correspondrait à des unités «naturelles» de réutilisation et permettrait de présenter des composants relativement indépendants en vue d'en faciliter la composition. Dans une certaine mesure, la programmation objet n'est en soi qu'un mécanisme de modularisation. Cependant, la frontière d'objet s'est avérée trop petite pour certains besoins, donnant lieu à des composants de

plus grande taille, tels que les cadres d'application, et trop grande pour d'autres, dont la réutilisation d'aspects fonctionnels, qui est l'objet de ce travail.

Depuis une dizaine d'années, plusieurs chercheurs ont développé des mécanismes de modularisation avancés qui permettent de rendre compte de plusieurs problèmes que la modularisation par objet ne permet pas de résoudre.

Le premier de ces problèmes est d'ordre conceptuel et a trait à la gestion de la complexité des systèmes d'information d'aujourd'hui. Dans de tels systèmes, les mêmes objets peuvent jouer plusieurs rôles qui correspondent chacun aux diverses fonctionnalités qu'ils supportent. Le fait de forcer ces fonctionnalités dans le même modèle affecte tant la compréhension des modèles que la maintenance et la réutilisation sélective de ces fonctionnalités. Notons, à titre d'exemple, les recherches sur la modélisation par modèle de rôles [Ande 92] et les modules conceptuels [Bani 98].

Le deuxième est d'ordre méthodologique et vise la séparation conceptuelle et la séparation logicielle des considérations à caractère architectural. Les avantages d'une telle séparation sont une configurabilité, une réutilisabilité et une portabilité plus grandes des fonctionnalités développées.

Le troisième est d'ordre plus pragmatique et vise l'intégration de composants développés de façon décentralisée (par divers constructeurs) en un tout fonctionnel et cohérent.

Les deux derniers problèmes ont donné lieu à plusieurs recherches portant sur divers aspects de la problématique, dont la composition par *filtres* [Aksi 92], la programmation *orientée-sujet* [Harr 93], la notion de *points de vues* [Nuse 94], la programmation *adaptative* [Lieb 96], la programmation *orientée-aspect* [Kicz 97] et la programmation *générique* [Bato 00a], [Bato 00b], [Czar 00].

On s'intéresse, depuis quelques années, à l'application des solutions de l'un des problèmes aux deux autres. Les techniques de modularisation telles que la *programmation par aspects* et la *programmation par sujets* ont notamment été proposées comme moyens d'implanter les *facettes* ou les *rôles fonctionnels* ([VanH 96], [Clar 99], [Ken 99]). L'approche que nous présentons a tout d'abord été développée comme une approche visant la représentation et l'implantation d'aspects fonctionnels [Mili 99].

Nous allons ici présenter plus précisément la problématique du développement de logiciels d'aujourd'hui, particulièrement celle reliée à l'intégration de composants réutilisables. Dans la section 2, nous discutons d'un nombre de techniques d'intégration et de composition de composants rendues possibles par la programmation *orientée-objet.* Ces techniques sont toutefois *orientées-objet* dans le sens qu'elles permettent d'intégrer des *objets* et non des «modules» qui correspondent à des *facettes fonctionnelles*. La section 3 présente notre approche de façon sommaire. La section 4 offre un survol du reste de la thèse.

I.1. Problématique du développement de logiciels

La complexité des systèmes d'information d'aujourd'hui provient principalement de la complexité des organisations qui les développent et de la complexité de leurs composants. Dans cette section, nous discutons de la problématique du développement en s'attardant sur les principales caractéristiques des logiciels d'aujourd'hui. Chacune de ces caractéristiques requiert la mise en oeuvre d'un certain nombre de techniques de modularisation et de composition.

I.1.1. Le développement avec réutilisation

Pour répondre à la demande sans cesse croissante d'applications de plus en plus complexes et de meilleure qualité, deux solutions s'offrent à nous, soit la génération automatique des programmes, soit la réutilisation, la composition et l'intégration de logiciels existants. La génération automatique de programmes sera possible le jour où nous concevrons des processus de développement qui sont précis et systématiques et qui garantissent un produit de qualité. Malgré des réussites enregistrées dans certains domaines d'applications limités, il n'est toujours pas possible de développer des processus qui soient génériques et applicables à tous les domaines. Il nous faut donc réutiliser des logiciels existants, ce qui est perçu comme un moyen efficace pour améliorer la productivité et la qualité dans le développement des logiciels [Coul 96].

La réutilisation peut toucher différentes phases de développement comme, par exemple, la conception, l'analyse et l'implantation [Booc 86]. La construction de nouveaux systèmes à partir de composants logiciels préconçus et prétestés devrait permettre de réduire l'effort de développement et de maintenance et d'améliorer la fiabilité des produits résultants.

Une organisation qui développe du matériel logiciel peut puiser dans deux sources pour obtenir du matériel logiciel réutilisable :

i) Elle peut utiliser des composants commerciaux (*Commercial Off The Shelf, ou COTS*) développés par des tierces parties. Bien que ces composants soient en général de très bonne qualité, le développeur a peu de marge de maoeuvre pour les réutiliser, et le manque d'accès au code source peut s'avérer être un obstacle important.

ii) Elle peut aussi réutiliser du matériel logiciel développé à l'interne. Il est possible, par exemple, de prévoir le développement d'une ligne de produits desservant un marché particulier. Les applications auront beaucoup en commun, et il vaudra par conséquent la peine d'investir dans le développement et la réutilisation des aspects communs aux diverses applications. Les composants seront ici faits sur mesure, car l'organisation en question exerce un contrôle absolu sur leur code et leur évolution.

Dans l'industrie d'aujourd'hui, peu d'organisations peuvent se permettre de développer des composants réutilisables pré-planifiés. La plupart achètent des composants sur le marché et doivent faire face aux difficultés que causent l'intégration des composants développés par divers fabricants et dont l'intégration ou l'interaction n'a pas été prévue. La composition devient alors difficile, surtout quand les *objets* (entités) manipulés par les différents composants sont les mêmes ou doivent être fusionnés pour former un seul objet [Harr 93].

I.1.2. Des systèmes d'information à la fois intégrés et distribués

Le problème d'intégration des composants se pose sur une plus grande échelle. Après avoir connu les systèmes d'information dédiés ou départementaux, nous nous retrouvons aujourd'hui avec des systèmes d'information d'entreprise qui consistent en

l'intégration de divers systèmes départementaux. Cette intégration repose sur deux éléments :

- une intégration des données : chaque application départementale aura ses propres données, mais différentes applications devront partager certaines données. Par exemple, dans un système d'information d'une entreprise manufacturière, une donnée «employé» sera manipulée non seulement par l'application de «Paie», mais aussi par l'application de «Gestion de la production» et
- une intégration au niveau des traitements : les applications doivent pouvoir s'invoquer mutuellement pour supporter les chaînes de traitement au complet.

Les problèmes reliés au partage, à l'intégration et à la composition des données sont traité dans les bases de données par l'introduction des *vues* qui nous permettent de regrouper des données en fonction de divers critères, d'y accéder au moyen d'un certain contrôle d'accès et de générer de nouvelles données à partir de celles qui existent [Ullm 82], [Delo 92] [Date 90] [Tsic 78]. Les *vues* sont considérées comme étant une abstraction de données qui facilitent l'accès et la manipulation des sous-ensembles d'information [Heil 90] tout en comportant les informations centralisées. On fait appel au concept des *données abstraites* ou de *vues* dans les systèmes de base de données pour faciliter la définition des interfaces usagers à travers les langages de base de données et pour décrire l'architecture des systèmes de base de données [Clay 85].

Avec le passage à l'*orienté-objet* vers la fin des années 80, on a dû abandonner les techniques qui proviennent des bases de données (*vues*) au profit d'une réutilisabilité et d'une évolutivité plus grandes des applications *orientées-objet,* surtout en raison du fait que les premières applications *orientées-objet* étaient des applications «individuelles» (*desktop applications*), ou applications à petite échelle. Depuis que l'*orienté-objet* est utilisé pour le développement et le déploiement d'applications à l'échelle de l'entreprise, le concept de fragmentation de la fonctionnalité des objets, à l'image des *vues* dans les bases de données relationnelles, a refait surface, et ce, sous diverses formes. Comme nous le verrons dans la prochaine section, les mécanismes de base, tels que l'héritage et l'agrégation, qui sont supportés par les langages de programmation *orientés-objet* ne sont, pour diverses

raisons, pas adéquats. Ainsi, d'autres concepts ont été proposés dans le cadre de *l'orientation-objet* comme, par exemple, les *interfaces multiples*, la modélisation par *rôles*, la programmation *orientée-sujet* et la programmation *orientée-aspect*, qui introduisent de nouveaux mécanismes de langages ou de modularisation permettant de mieux mettre en relief la problématique d'intégration et de composition. Notre travail s'inscrit dans cette lignée.

I.1.3. Déploiement incrémental et configurabilité dynamique

À cause des risques reliés au développement de grands systèmes informatiques et des fortes pressions pour mettre en service les applications, une bonne partie des systèmes d'information d'aujourd'hui est mise en service de façon incrémentale. Ainsi, nous devons prévoir la possibilité de pouvoir ajouter des fonctionnalités qui viennent s'ajouter aux systèmes existants. Ces fonctionnalités doivent souvent interagir avec celles qui existent déjà. Selon la nature de l'interaction, l'intégration peut être plus ou moins compliquée, et doit, dans la majorité des cas, être pré-planifiée. Par exemple, la programmation par *sujet* et la programmation par *aspect* ont été conçues dans le but de supporter le développement par fonctionnalités ou "*features*" (*feature based development*), où des fonctionnalités additionnelles viennent s'ajouter à un ensemble de fonctionnalités de base sans en modifier la structure d'appel de l'application [Harr 93], [Kicz 97].

La plupart des techniques opèrent l'intégration au niveau du code source. Elles requièrent donc une nouvelle compilation et un redéploiement des applications, y compris les approches *orientée-sujet* et *orientée-aspect* [Harr 93], [Kicz 97]). Or, de plus en plus les applications fonctionnent en continu, et l'on ne peut se permettre de les arrêter, ne serait-ce que partiellement, pour effectuer les mises à jour (remplacement de fonctionnalités existantes, ajout de nouvelles fonctionnalités). Nous devons par conséquent faire l'intégration au moment de l'exécution. Cette intégration (la *composition dynamique)* pose des problèmes beaucoup plus grands et exige que l'on fasse appel à un certain nombre de techniques pour la supporter, dont la liaison dynamique et la direction d'appels (*dispatching*) dynamique. La liaison dynamique, tardive, dans les langages *orientés-objet*

nous permet de reporter des décisions relatives au code jusqu'au moment de l'exécution. L'avantage d'une telle liaison est le fait que les objets puissent changer de type au moment de l'exécution, mais les types sont limités à une hiérarchie prédéfinie. Les DLL (*Dynamic Link Libraries*) constituent un autre moyen d'effectuer une liaison dynamique qui permet l'édition de liens avec une librairie externe au moment précis où une fonction de cette librairie est utilisée. Ceci permet d'interchanger des versions de librairies pendant que le programme s'exécute, sans que l'on ait à redémarrer ce dernier. L'éventail de fonctionnalités est limité toutefois par l'API de la librairie en question.

En ce qui a trait au *dispatch* dynamique, notons l'exemple de la programmation *événementielle*, où plusieurs programmes s'enregistrent pour réagir suite à certains événements. Pour mettre en place un nouveau service, il suffit de l'enregistrer en fonction d'un événement donné. Cette méthode permet l'ajout et le retrait dynamique de fonctionnalités. Elle est utilisée par les systèmes d'exploitation (p. ex., *Windows*), mais permet une interaction qui est limitée et prédéfinie entre les applications.

Le travail que nous présentons se veut être un compromis entre ces deux approches. Il permet un *dispatch* dynamique riche, non-limité à des envois ou des réactions à des messages, et un changement dynamique de comportement. L'éventail de comportements *potentiellement disponibles* est plus ou moins figé.

Dans la section qui suit, nous allons présenter les divers mécanismes de composition offerts par les langages de programmation. Comme nous le verrons, ces mécanismes ne répondent pas aux besoins décrits dans cette section, et il nous faudra explorer de nouvelles voies.

I.2. Mécanismes de composition et d'intégration

Dans cette section, nous présentons les mécanismes de base pour la composition dans les langages de programmation *orientés-objet* et traiterons leurs limitations. Nous parlerons tour à tour de l'héritage, de l'agrégation, des classes génériques et des interfaces. L'héritage et l'agrégation sont deux mécanismes fondamentaux de structuration et de hiérarchie. L'héritage permet de traduire une hiérarchie conceptuelle de classes bâtie sur le

lien «*is_a*» interprété comme étant une relation d'abstraction/concrétisation et de généralisation/spécialisation. L'agrégation d'objets permet de traduire une hiérarchie structurelle bâtie sur le lien «*is-part*», perçu comme relation de décomposition d'un tout (objet composite, agrégat) en parties (sous-objets, composants). Ces deux concepts permettent de définir intuitivement des objets plus ou moins semblables [Stef 86] à quelques caractéristiques près. Leur principe est de généraliser/spécialiser et de composer/décomposer un regroupement d'objets (une classe). Les classes génériques sont paramétrées par d'autres classes et permettent d'en composer assez étroitement deux classes. Nous traiterons aussi brièvement des *interfaces*, telles que supportées par quelques langages de programmation.

I.2.1. La relation d'héritage

La relation d'héritage permet de spécifier une nouvelle classe en précisant ce qui la différencie des classes existantes. Elle stipule que si une classe «A» hérite d'une classe «B», toutes les propriétés qui s'appliquent à «B» s'appliquent aussi à «A». Nous dirons ainsi que les instances de «A» héritent des propriétés introduites dans la classe «B». La sous-classe peut introduire de nouvelles propriétés (attributs ou fonctions) ou spécialiser les propriétés héritées de(s) la classe(s) mère(s). La Figure 1.1 montre un exemple de la relation d'héritage qui existe entre des classes. Cette figure montre deux spécialisations de la même classe mère «Iterator». Les deux spécialisations correspondent ici à deux implantations particulières du comportement défini dans la classe mère.

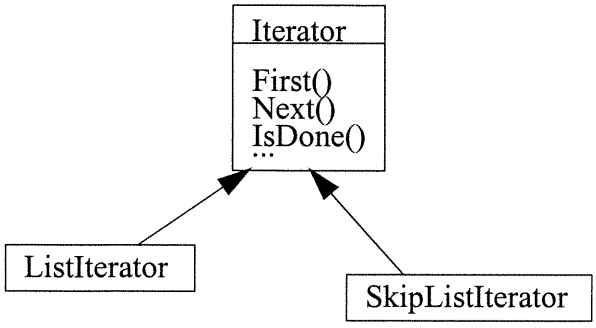


Figure 1.1 : Relation d'héritage simple

L'héritage a longtemps été considéré comme un mécanisme central à la réutilisation, car il permet de réutiliser du code de façon naturelle et intuitive : en spécifiant qu'une classe est une sous-classe d'une autre, nous pouvons réutiliser tout le code de la super-classe et ajouter (ou redéfinir) les aspects qui nous conviennent. En quelque sorte, si «A» est une sous-classe de «B», «A» peut être perçue comme la composition des fonctionnalités de «B» et des fonctionnalités propres à «A» (définies dans «A»). Cette composition est «profonde» dans ce sens que les méthodes de «A» et de «B» vont être invoquées, peu importe leur origine. Par contre, cette façon d'étendre la fonctionnalité d'une classe impose une séquence dans le développement : on ne peut pas définir la fonctionnalité propre à «A» avant de définir «B». De plus, on ne peut pas effectuer cette extension pendant l'exécution.

Dans certains langages, il est possible de spécifier une classe comme étant l'extension de deux classes différentes et indépendantes : c'est l'*héritage multiple*. La figure 1.2 montre un exemple d'héritage multiple. La classe «EmpFormation» (pour employé(e) en formation) représente une personne qui est à la fois «employé(e)» et «étudiant(e)». Cette classe hérite des caractéristiques des deux super-classes.

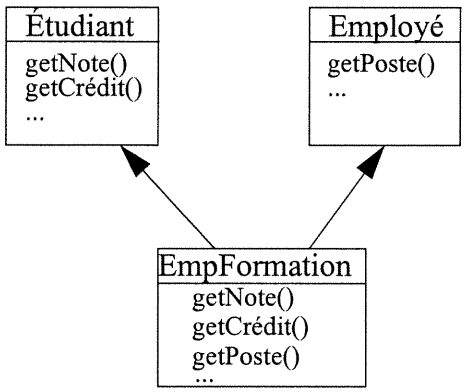


Figure 1.2 : Relation d'héritage multiple (spécialisation)

Dans cet exemple, un objet de la classe «EmpFormation» joue deux rôles dans son application (tout au moins dans la vie) : le rôle d'étudiant et le rôle d'employé. La composition de ces deux rôles s'est opérée au niveau de la classe «EmpFormation».

En tant que mécanisme de composition de domaines fonctionnels, l'héritage multiple est un mécanisme relativement simple et efficace. Il n'est cependant pas disponible dans tous les langages et ne permet pas la composition dynamique : la classe «EmpFormation» doit être définie de façon statique dans le code. De plus, un objet, une fois créé, appartiendra à une classe ou à une autre et ne pourra pas changer de comportement. Supposons finalement que nous voulons ajouter à la classe «Employé» le nombre d'actions que l'employé détient dans l'entreprise. Au lieu de modifier la classe «Employé» directement, ce qui n'est pas toujours possible, surtout lorsque la classe est fournie par une tierce partie, il nous faudra :

- définir une sous-classe d'«Employé» avec cette caractéristique additionnelle et
- redéfinir de nouvelles sous-classes des classes existantes pour prendre en compte la nouvelle addition.

Ce dernier point constitue un handicap majeur pour la composition incrémentale des classes [Harr 93].

I.2.2. L'agrégation

La relation d'agrégation, traduite par une hiérarchie structurelle, consiste en une décomposition des structures en sous-structures jusqu'à l'obtention des structures élémentaires non décomposables. Cette hiérarchie est définie par la relation «*is-part*» que l'on retrouve dans les réseaux sémantiques. Le lien qui existe entre la hiérarchie structurelle et l'agrégation résulte du fait qu'à chaque niveau de cette hiérarchie, une entité est un agrégat ou un assemblage des entités représentant ses parties.

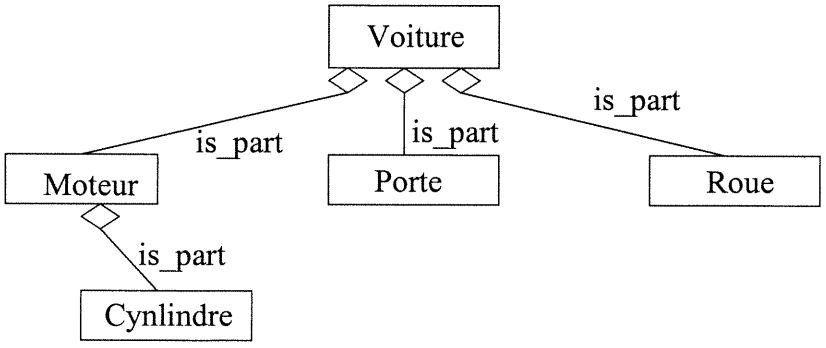


Figure 1.3 : Relation d'agrégation

L'agrégation traduit une implantation structurée d'un objet en termes de sous-objets (les agrégats) correspondant à des aspects particuliers et offrant chacun un comportement propre à lui. Voir, à cet effet, le schéma suivant.

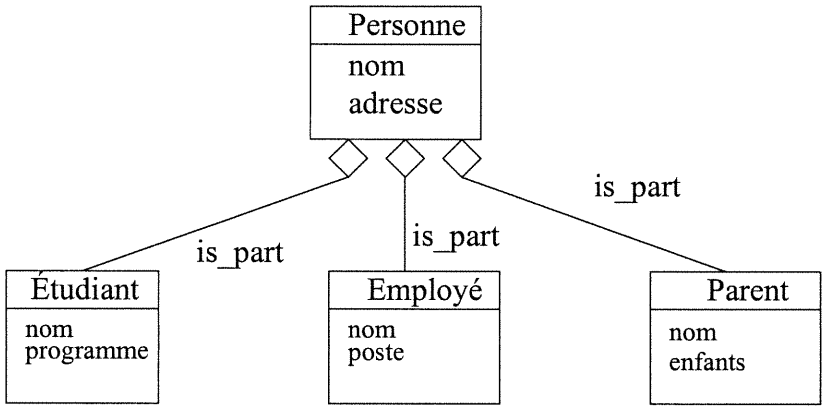


Figure 1.4 : Implantation de plusieurs comportements

Ici, une personne qui est employée à plein temps, qui est un parent et qui étudie sera représentée par l'agrégat «Personne», lequel regroupe trois autres objets des types «Étudiant», «Employé» et «Parent». Chaque objet du type «Personne» comportera les trois objets ou composants. En outre, en absence d'une intégration plus serrée entre l'agrégat et ses composants, on remarque que le même attribut, soit «nom», est présent dans les trois composants. Que faire si une méthode sur une personne «p» appelle le "*setter*" p.setNom("Jean")? Lequel des noms sera touché? Il faudra probablement les affecter tous pour que toutes les méthodes de tous les composants voient la même donnée. Qu'en est-il d'un appel interne à l'une des méthodes de «Parent» qui lui changerait le nom? Ne faudrait-il pas effectuer le même changement pour les autres composants?

Il est clair que la simple utilisation de l'agrégation ne permet pas la composition adéquate des comportements inhérents aux trois classes d'origine. L'approche que nous proposons a été *implantée* en utilisant l'agrégation comme *mécanisme de base*, mais nous avons veillé à offrir toute l'infrastructure qui permet de répondre aux questions présentées ci-dessus [Mili 99].

I.2.3. Les classes génériques

Une classe générique est une classe paramétrée qui peut être utilisée dans des contextes différents les uns des autres. Les classes génériques permettent de faire abstraction des diverses utilisations d'une ou de plusieurs structures de données et de capturer ces différences dans un paramètre d'où le nom *paramétrisation*. Les paramètres d'une classe générique peuvent être les types de certains des ses membres. La paramétrisation est supportée par plusieurs langages, dont les langages de spécifications formelles, tels que Clear [Burs 77], Lileanna [Trac 93] et Obj [Futa 87], et les langages de programmation tels que Ada[Barn 89], Eiffel [Biel 93], C++ [Stro 97] et Java [Barn 00].

Les paramètres génériques sont spécifiés au moment de la déclaration des variables des classes. Prenons l'exemple d'une classe générique «List» munie de son instanciation pour deux types différents, soit le type «int» pour déclarer une liste d'entiers et le type «string» pour déclarer une liste de chaînes de caractères (Figure 1.5).

```
template <class Item>
class List {
public :
   List(long size = DEFAULT_LIST_CAPACITY);
   List(List&);
   ~List();
   List& operator=(const List&);
   long Count() const;
   Item& Get(long index);
   // ...
};
List <string> liste1;
List <int> liste2;
```

Figure 1.5 : Classe générique en C++

Dans cet exemple, la classe «List» est générique, et son paramètre générique est un *paramètre de type* appelé «Item». Cela signifie que cette classe est définie pour tout type capable de se substituer au type «Item» dans le code de la classe, que l'on ne voit pas ici. Le type doit supporter les opérations appliquées sur les variables de type «Item» dans le code de la classe. Il est probable que l'on n'exige aucune opération particulière dans le cas

d'une liste simple : on ne fait que placer et retirer les valeurs. Dans le cas d'une liste triée, on pourrait exiger que le type «Item» supporte une opération de comparaison. Pour le cas de C++, c'est le compilateur qui ramasse toutes les opérations que nous avons appelées sur les éléments de type «Item» dans le code de «List» et qui s'assurera, au moment de la déclaration des variables («liste1» et «liste2»), que le type d'instanciation («string» et «int», respectivement) supporte ces opérations. Dans le langage Ada, le développeur spécifie lui-même les exigences du type paramètre dans l'entête du *package*.

Les classes génériques peuvent être perçues comme étant un mécanisme de composition de classes. Dans ce cas, la composition est spécifiée dans l'une des «composantes», notamment, la classe «List» elle-même. L'autre composante, «Item», est définie d'une manière «abstraite» et représente une «exigence» sur le genre de classes que nous pourrons composer avec les listes. Prenons l'exemple d'une application de gestion de personnel qui inclut une liste triée du personnel. Une classe générique «ListeTriée» pourrait exiger que son paramètre de type «Item» supporte l'opérateur «<» comme : `boolean operator < (Item unAutre);`

Les classes génériques ont été utilisées par plusieurs chercheurs à titre de mécanisme de composition de classes qui a l'avantage de permettre de spécifier l'un des paramètres de façon abstraite [Guog 86]. Batory [Bato 92] et VanHilst [VanH 96] ont utilisé ce type de paramétrisation pour implanter la composition d'*aspects fonctionnels* et de *rôles*. Cependant, cette composition est statique et restrictive. Elle doit respecter une séquence d'instanciation particulière et exige une vision centralisée du développement et de l'intégration.

Notre approche fait appel à un mécanisme semblable à la généricité sur le plan définitionnel, notamment pour représenter les exigences fonctionnelles d'un *point de vue* et pour générer une *vue* à titre d'instanciation de ce *point de vue* pour une classe donnée [Darg 96]. Par contre, l'ajout et le retrait de *vues* se font de façon dynamique, comme nous le verrons plus tard.

I.2.4. Interfaces multiples

À part la réutilisation des composants et la flexibilité de leur intégration dans un environnement multi-usagers distribué, nous aimerions offrir divers droits d'accès à divers usagers. Un groupe d'usagers aura alors accès à un ensemble spécifique de données, ce qui représente une forme de sécurité à l'égard des données qui sont inaccessibles. Cela représente une approche utile surtout lorsque plusieurs usagers ont accès à ces données et que l'on veuille fournir et assurer une cohérence et des privilèges d'accès. Les interfaces représentent, pour leur part, un moyen de contrôle d'accès et de regroupement de caractéristiques. Elles nous permettent de filtrer les fonctionnalités et les données pour une classe spécifique [Marc 93].

Les interfaces sont déjà implantées dans quelques langages de programmation *orientés-objet.* Dans C++, par exemple, le nombre d'interfaces est limité à trois pour l'accès privé, protégé et publique. Mais à toute fin pratique, une seule interface est disponible dans le cas d'un programme client, c'est-à-dire l'interface publique. Dans Java, plusieurs interfaces peuvent être définies pour une classe donnée qui offre les services de ces interfaces.

Les interfaces sont définies au moment de l'implantation pour un ensemble précis d'objets et leur usage est perçu comme un bon moyen de contrôler l'accès aux données et de regrouper des composants.

Dans cette section, nous avons traité des divers mécanismes de composition et d'intégration existants. Nous avons étudié, pour chacun de ces mécanismes, la mesure dans laquelle il peut être utilisé pour intégrer de façon dynamique des facettes fonctionnelles d'un même objet. Comme nous l'avons vu, deux problèmes majeurs sont reliés à ces mécanismes :

- ils exigent un modèle de développement centralisé où l'on doit être propriétaire de la fonctionnalité de base pour pouvoir y intégrer d'autres fonctionnalités, et
- ils ne permettent pas l'évolution dynamique de cet ensemble.

Plusieurs approches ont été proposées dans la littérature pour répondre à cette problématique, notamment la séparation des aspects fonctionnels que l'on veut intégrer, dont la pro-

grammation par *sujets* [Harr 93], et la programmation par *aspects* [Kicz 97]. Cependant, comme nous le verrons dans le chapitre deux, ces approches ne permettent pas l'évolution dynamique des fonctionnalités d'un objet.

L'approche que nous proposons, appelée *programmation par vues*, permet l'évolution dynamique de l'ensemble des fonctionnalités des objets.

I.3. Notre approche

Dans les systèmes d'information qui sont présentement développés, les objets jouent plusieurs rôles et supportent diverses fonctionnalités. Très souvent, ces fonctionnalités sont développées d'une façon indépendante (par des tierces parties). De plus, un objet doit pouvoir acquérir et perdre des fonctionnalités durant sa durée de vie. La «composabilité» des fonctionnalités exige une bonne modularisation de ces fonctionnalités qui devrait en permettre le développement et la réutilisation séparée. L'évolution dynamique de l'ensemble de fonctionnalités supportées par un objet à un moment donné exige que ces dernières ne soient pas rattachées à la définition des objets, mais plutôt qu'elles pouvaient y être intégrées d'une façon dynamique.

Dans notre travail, nous avons tenté de répondre à un éventail d'attentes en introduisant la notion de *vues* qui représentent les aspects fonctionnels des classes d'objets. Ces aspects sont décrits d'une façon «modulaire», facilitant ainsi la composition et la réutilisation.

Pour nous et à tout moment, un objet d'une application donnée peut être caractérisé par une fonctionnalité de base et un ensemble variable de *fonctionnalités* ou de *vues* [Mili 99]. Dans notre modèle, chaque *vue* comporte ses propres fonctionnalités et ses propres *don*nées. Par contre, les *vues* peuvent s'appuyer sur des fonctionnalités et des données de base de l'objet pour offrir leurs propres fonctionnalités. Au moment de sa création, un objet a accès à la fonctionnalité de base. Durant sa durée de vie, des *vues* peuvent lui être rajoutées et retirées, permettant d'en changer le comportement dynamiquement. Le comportement de l'objet est caractérisé en tout temps par l'ensemble des *vues* qui lui sont rattachées. Si l'on représente la fonctionnalité de base et les fonctionnalités individuelles par

des interfaces, l'interface globale, «*I*», de l'objet peut être définie comme suit :

$$I = I_0 \oplus Iv,_1 \oplus Iv,_2 \oplus ... \oplus Iv,_n$$

Où I_0 est une interface de *base* de l'objet, et $I_{v,i}$ l'interface de la $i^{ème}$ *vue*. Le comportement de base (I_0), ou comportement intrinsèque, regroupe les objets d'une même classe en fonction des caractéristiques minimales; les autres comportements ($I_{v,i}$) sont dynamiques et peuvent changer, et cela, pour une même classe et d'un objet à un autre. Le signe $\oplus$ représente ici l'union des *interfaces* et symbolise la composition des comportements sous-jacents aux diverses *vues*. Il arrivera en effet que le même comportement soit offert par des *vues* différentes ou par la fonctionnalité de base et les *vues*. Dans ce cas, l'ensemble des implantations disponibles sera appelé selon un protocole par défaut, que les développeurs pourront redéfinir [Ossh 95].

Notre approche traduit, dans une certaine mesure, la notion de *classification multiple et dynamique* par laquelle un objet peut appartenir à plusieurs types simultanément, même changer de types durant sa vie [Mart 92].

En plus de la possibilité d'ajouter et de retrancher des fonctionnalités (*vues*) de façon dynamique, notre approche se distingue aussi des approches existantes par la façon dont ces fonctionnalités sont développées [Darg 96]. Nous croyons en effet que ces fonctionnalités décrivent fréquemment des processus d'affaire génériques ou des manipulations d'infrastructure, lesquelles ne dépendent pas spécifiquement de l'*objet de base*. Par exemple, dans une entreprise de location de camions, un objet «camion» devrait être traité comme un *capital amortissable* par une application «Comptable», au même titre que la «chaise» ou l'«ordinateur» de la secrétaire. De plus, l'objet devra supporter des fonctionnalités de persistance (p. ex., utilisation de variables cachées, fonctions de sauvegarde) qui seront applicables à toute entité que l'on voudrait emmagasiner. Dans les deux cas, il est possible de codifier la fonctionnalité en question sous forme *générique* de sorte qu'elle puisse être *instanciée* pour différents objets (camion, chaise, ordinateur). Ainsi, nous avons introduit la notion de *point de vue,* **viewpoint**, qui est aux vues ce que sont les classes génériques aux instanciations de ces classes. Notre mécanisme de programmation par vues se résume ainsi :

Une *vue,* comme une classe, est définie par un ensemble de variables d'état et d'opérations. Elle spécifie un aspect fonctionnel donné et représente un rôle particulier des objets qui lui sont reliés. Notre approche permet la modification de l'ensemble des *rôles* des objets en rendant dynamique l'ajout et le retrait des *vues*. Pour amener le concept de *vue* à être plus générique et non lié à une seule classe, les attributs de la *vue* sont décrits par une autre construction, le **viewpoint**, qui est une forme de «patrons» de *vues*. Le **viewpoint** peut être appliqué à un ensemble de classes, que nous appelons *classes de base*, qui répondent à un ensemble d'exigences spécifiées dans sa partie **requires**. Il ajoute à chaque *classe de base*, à laquelle il est appliqué, un ensemble de caractéristiques définies dans sa partie **provides**. Une *vue* est le résultat de l'application du **viewpoint** à une classe, ses membres sont générés automatiquement à partir des membres de la partie **provides** du **viewpoint** et peuvent référer des membres de sa partie **requires**. Pour utiliser un **viewpoint**, l'usager doit d'abord le particulariser pour une classe donnée. Ceci est spécifié par un énoncé **viewdef** qui donne un nom à la *vue* obtenue en combinant **viewpoint** et la *classe de base*. Au besoin, l'énoncé pourra faire la liaison entre des attributs de la partie **requires** et des attributs de la *classe de base*.

Pour la réalisation pratique du travail, nous avons développé un support aux fins d'une programmation par *vues* pour le langage C++. Les outils que nous avons développés permettent la définition des **viewpoint(s)**, la génération des *vues* et leur manipulation dynamique. Tout peut être réalisé grâce à un ensemble de règles de grammaire ajoutées à celles du langage C++ et analysées par notre pré-processeur développé. Le rôle de ce dernier est de générer un code C++ qui permette la manipulation transparente des *vues*. Notre approche à l'implantation repose sur deux principes :

- introduire un minimum de constructions possibles aux langages de programmation existants et
- s'assurer que la programmation par *vues* serait une extension naturelle de la programmation objet traditionnelle.

Par ces deux principes, nous avons pu adopter une approche par transformation de code, étant donné que la description des *points de vues* et l'utilisation des *vues* sont transformées

par des outils que nous avons développés pour générer du code standard dans le langage hôte, dans notre cas le langage C++.

I.4. Contenu de la thèse

Le chapitre deux contient une revue de la littérature. Le chapitre trois présente les principes de notre approche et en décrit l'utilisation. Les modèles formels de *points de vues*, et des *vues* sont présentés dans le chapitre quatre. Les chapitres cinq et six sont consacrés à l'analyse et à la présentation des outils de support de *vues* qui ont été développés. Le chapitre sept présente un exemple d'implantation d'une application avec *vues* et une comparaison entre notre approche et d'autres approches similaires. Nous concluons dans le chapitre huit. Dans ce qui suit, nous présentons de manière plus détaillée le contenu des chapitres.

Chapitre II : Revue de littérature

Dans ce chapitre nous faisons un survol des diverses approches offrant la notion de *vues*. Nous présentons les *vues* en représentation de connaissances en discutant le problème de la représentation des catégories conceptuelles. Ensuite, nous décrivons le mécanisme de *vues* dans les bases de données. Dans les systèmes *orientés-objet*, la notion de *vues* est étudiée à différents niveaux de développement d'une application, notamment aux niveaux de l'analyse, de la conception, de la programmation et de l'exécution. L'introduction des *vues* au niveau de la programmation nécessite l'usage de certaines particularités des langages utilisés, lesquelles feront l'objet de discussions.

Chapitre III : Programmation par vue: principes et exemples

Dans ce chapitre nous présentons le cahier de charge de la notion de *vues* telle que nous la voyons, ainsi que les principes et les concepts de notre approche. Des exemples de *vues* sont donnés pour en montrer l'utilité. Nous démontrons ensuite pourquoi les approches existantes ne répondent pas à nos exigences sur les *vues*. L'implantation des *vues* nécessite de nouveaux concepts car les relations d'héritage et d'agrégation ne permettent pas de dynamiser les *vues*.

Chapitre IV : Modèle formel

Dans le chapitre quatre, nous parlons des formalismes et notations existantes pour la définition de nouveaux concepts. Les notions de types, de types génériques et de théories sont présentées et utilisées pour définir les *points de vues*. Le renommage, l'extension et la spécialisation de *vues* et de *points de vue* sont discutés. Les notions de sous-typage, de type statique, de type actuel, de type maximal et de leur relation avec les *vues* sont aussi présentés.

Chapitre V : Support pour la programmation par vues en C++

Dans le chapitre cinq, nous décrivons les principes de l'implantation de la programmation par *vues* en C++. Nous commençons par un survol de notre outillage. Nous décrivons ensuite la syntaxe des *vues* et *points de vues* ainsi que le principe d'implantation des diverses fonctionnalités de gestion de *vues*. Nous décrirons aussi les transformations appliquées au code usager qui utilise des *vues* pour obtenir du code C++ standard.

Chapitre VI : Implantation des outils de support de vues

Dans ce chapitre, nous décrivons notre implantation en détail tout en s'attardant aux structures de données utilisées par les divers outils d'analyse et de génération ainsi qu'aux détails de l'implantation des diverses fonctionnalités de gestion de *vues*.

Chapitre VII : Exemple et discussion

Dans ce chapitre nous présentons l'implantation d'un simple exemple qui s'appuie sur la programmation par *vue*s. Nous faisons ensuite une comparaison qualitative de notre approche pour les *vues* avec la programmation *orientée-sujet* et la programmation *orientée-aspect*.

Chapitre VIII : Conclusion

Dans ce chapitre, nous rappelons la problématique, résumons notre approche et le travail accompli et décrivons l'ensemble de travaux que nous poursuivrons, tant sur le plan théorique que sur le plan pratique.

Chapitre II

Revue de la littérature

Les notions de *vue*, de *point de vue* ou de *perspective* reliées aux systèmes informatiques ne sont pas nouvelles. Elles sont déjà définies dans les bases de données, dans la représentation des connaissances, dans l'orientation objet et au niveau de la modélisation.

Le concept de *vue* a été introduit dans les bases de données pour accroître l'indépendance et la sécurité des données et pour faciliter et minimiser leur accès. Cependant, le modèle de données reste centralisé et statique.

En représentation de connaissances, la notion de *vue* permet de concevoir une représentation flexible des données qui répond aux besoins variés des divers utilisateurs. Cependant, les langages de représentation de connaissances sont plus adaptés à la description d'entités qu'à la programmation.

Au niveau de la modélisation, les *vues* représentent les différents *rôles* qu'un ensemble d'objets peut jouer. Cependant, la séparation en *rôles* indépendants disparaît aux phases de la conception et de la programmation.

Dans les systèmes *orientés-objet*, les *vues* sont introduites en premier au niveau de la programmation où les mécanismes existants tels que les relations d'héritage et d'agrégation sont utilisées pour les supporter. Plus tard, de nouvelles recherches sur les *interfaces multiples* des classes, sur la programmation *orientée-sujet* qui compose des *sujets* développés indépendamment et sur la programmation *orientée-aspect* qui consiste à intégrer des composants ou *aspects* dans une application, ont été élaborées. Ces recherches ont traité le comportement multiple que peut avoir un ensemble d'objets. Cependant, elles souffrent du fait qu'elles sont statiques et l'éventail de comportements disponibles à un objet donné est fixé au moment de la compilation, voire l'édition de lien, et ne peut pas changer durant l'exécution.

Dans ce qui suit, nous allons présenter une vue générale de chacune de ces

approches.

II.1. Vues en base de données

Pour les bases de données, le mécanisme de *vue* a été introduit à l'origine par le groupe CODASYL au moyen du concept de sous-schéma pour assurer l'indépendance et la sécurité des données entre les divers niveaux de représentation d'une base de données. Des sous-schémas, ou schémas externes, qui représentent les *vues* sont spécifiés à partir du schéma conceptuel qui est le schéma de base des données [Date 90], [Tsic 78], [Bell 98]. Chaque schéma externe décrit un point de vue spécifique pour une application particulière et représente une *vue* des données qui est accessible par un ensemble d'usagers. Cette façon de faire permet une sécurité et une flexibilité d'accès aux données spécifiques à une application et à des usagers particuliers. Il peut être considéré comme une manière commode de manipuler et de regrouper des données, qui ne montre à l'usager que ce qui l'intéresse, ou comme un moyen de garantir une certaine sécurité en cachant certaines données jugées confidentielles [Adib 81], [Date 81], [Conn 98], [Rica 90].

II.1.1. Principes des vues dans les bases de données

Le modèle relationnel représente les entités d'une application et les associations qui les relient sous forme de tables ou de relations [McFa 88], [Rica 90]. Une *vue* est le résultat dynamique d'une ou de plusieurs opérations portées sur les tables d'une base de données [Conn 98]. Elle est une relation virtuelle dérivée par une requête appliquée sur des relations de la base de données [Bert 92]. Elle définit des *projections* plus ou moins compliquées sur le modèle des données [Ullm 82], [Delo 92]. Mais les vraies données restent centralisées dans un seul modèle, même si diverses applications n'en voient qu'une partie. Les *vues* sont considérées comme une abstraction de données qui facilite l'accès et la manipulation des sous-ensembles d'information [Heil 90] tout en comportant les informations centralisées [Clay 85]. Une information dans une *vue* correspond à une (ou plusieurs) donnée réelle dans les tables de base. Elle peut aussi être dérivée à partir

d'une ou de plusieurs données réelles.

II.1.2. Caractéristiques

Les *vues* dans les bases de données qui sont représentées sous forme de tables virtuelles ne sont pas maintenues d'une façon permanente comme le sont les tables de base; elles sont plutôt construites automatiquement par le SGBD (Système de gestion de base de données) [Stor 88], [Elma 00]. Dans ce qui suit, nous en décrivons les caractéristiques.

Indépendance des données : Elle est concrétisée par le traitement indépendant de sous-ensembles de données représentés par les *vues*.

La sécurité : Elle est reliée au fait que les *vues* permettent un contrôle d'accès aux données particulières qui intéressent un ensemble d'usagers. Les données ne devant pas être accessibles sont sécuritaires et seuls les usagers qui en ont le droit peuvent y accéder.

La réduction de la complexité : Elle est offerte grâce aux *vues* dérivées à partir de plusieurs tables de base. Ainsi, au lieu d'avoir à accéder à un grand nombre de tables de base pour chercher l'information, cette dernière, regroupée, nous est offerte dans une même table virtuelle.

La composition fonctionnelle et le contrôle d'accès sur les informations : Ils sont le résultat du regroupement des données.

À part ces caractéristiques, il faut que les *vues* comportent d'autres caractéristiques, telles que l'intégrité et la cohérence des données. Il faut notamment que chaque mise à jour effectuée dans une table de base le soit aussi dans toutes les *vues* qui lui sont reliées pour assurer une intégrité. De la même façon, si une *vue* est mise à jour, les tables de base sous-jacentes doivent l'être elles aussi. Cela n'est pas toujours facile à faire, surtout pour les mises à jour des *vues* impliquant plusieurs tables de base et résultantes d'une agrégation d'opérations. Par ailleurs, le problème de la cohérence relative aux *vues* dans les bases de données relationnelles est traité dans [Gott 88]. Pour Gottlob et al., une vue cohérente est celle qui, une fois modifiée, affecte correctement la base de données correspondante. Cependant, cette modification n'est pas toujours évidente, surtout lor-

sque plusieurs utilisateurs ont accès simultanément aux données de base. Le problème de mise à jour des *vues* est davantage difficile lorsqu'elle affecte indirectement les données. Par exemple, il peut exister, dans la *vue*, un enregistrement «père» sans que ses descendants n'y existent. Dans le cas d'une suppression de cet enregistrement, les enregistrements qui lui sont reliés ne sont plus accessibles, bien qu'ils existent toujours. Pour régler ce problème, il suffit de manipuler seulement les données invariantes [Bros 88], [Ullm 82].

II.1.3. Limitations

On ne peut négliger certains des inconvénients reliés à la génération des *vues* et à leur manipulation. Parmi ces inconvénients, notons i) la mise à jour restrictive des *vues*, ii) la performance réduite et iii) la centralisation et l'aspect statique du modèle conceptuel.

- La mise à jour est restrictive à travers les *vues* : La mise à jour peut se faire plus ou moins facilement selon que la table virtuelle (la *vue*) est dérivée directement ou indirectement de la table de base, ou qu'elle le soit de plusieurs tables de base.
- La performance : La création, la mise à jour et la façon dont on accède aux tables virtuelles sont coûteuses pour le système. Une *vue* n'existe pas à l'avance, elle est dynamiquement créée. Ainsi, si elle est dérivée à partir de plusieurs tables de base, sa création est encore plus coûteuse. À chaque fois qu'une requête est effectuée, un accès aux tables de base permet d'y extraire les informations demandées, mais une telle requête a toutefois l'avantage de n'occuper que peu de place.
- Le modèle conceptuel est statique : Les requêtes qui nous permettent de construire les *vues* sont prédéfinies et ne sont pas modifiables dynamiquement. Ainsi, le modèle conceptuel est fixe pour toute la durée de vie de l'application.

II.2. Vues en représentation des connaissances

Une représentation des connaissances flexible et commode et qui répond aux besoins des divers types d'utilisateurs est nécessaire pour assurer une bonne qualité de

l'application manipulant ces connaissances. C'est la diversité des représentations, la façon d'extraire les données et les services reliés aux données manipulées offerts simultanément qui ont entraîné une représentation générique qui change selon les besoins des utilisateurs.

Dans le domaine de la représentation des connaissances, les *vues* sont introduites pour offrir la représentation multiple et possiblement dynamique des connaissances. Dans la pratique, les *vues* traitent la représentation de «catégories conceptuelles»; le défi de la représentation se présente lorsque ces catégories offrent diverses facettes de caractéristiques. Une telle variation des catégories est reliée aux perspectives variées des concepteurs des objets. Ainsi, la représentation par *vues* est conçue pour simplifier et supporter ce changement dynamique des objets.

Nous présentons ci-après la problématique inhérente à la représentation multiple des connaissances, les différents mécanismes de représentation ainsi que les limitations de ces mécanismes.

II.2.1. Plusieurs interprétations pour une représentation

Une représentation traduit une connaissance qui peut, selon le contexte dans lequel elle est traitée ou selon son interprétation, représenter différentes significations. Il ne suffit donc pas de la faire, il faut pouvoir l'interpréter.

Prenons l'exemple d'un objet «Personne». Cet objet comporte les caractéristiques nom, adresse, salaire, poste, éducation, etc. Pour un objet «Personne» particulier, on peut s'intéresser à un sous-ensemble de ses caractéristiques. Par exemple, son salaire et son poste peuvent nous intéresser si nous l'interprétons du *point de vue* «Emploi». Du *point de vue* «Éducation», ce sont ses cours et ses diplômes qui nous intéressent.

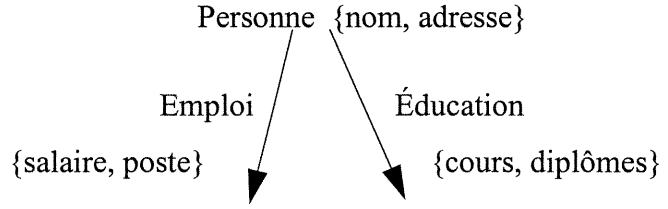


Figure 2.1 : représentation sous forme d'arbre

Schématiquement, nous pouvons représenter un objet avec ses différentes interprétations ou *points de vue* sous forme d'un arbre [Mart 84], chaque sous-arbre étant un concept (interprétation) différent. Par exemple, pour l'objet «Personne», nous avons l'arbre de la figure 2.1. La relation d'héritage nous permet aussi de faire un autre type de représentation sous forme d'arbre, dans laquelle chaque *vue* et représentée par une sous-classe [Way 87].

En général, le problème de représentation de connaissances consiste à trouver des formules informatiques capables de regrouper les propriétés dans des contextes différents les uns des autres. Tout semble être possible quand les connaissances, munies de leurs propriétés, sont bien définies statiquement et quand on connaît les contextes dans lesquels elles vont être étudiées.

II.2.2. Les mécanismes de représentation

En représentation de connaissances, plusieurs mécanismes sont utilisés pour faire la représentation des *vues*. Ces dernières regroupent un ensemble défini de caractéristiques reliées à un ensemble d'objets. Dans ce qui suit, nous présentons les techniques KRL (***K**nowledge **R**epresentation **L**anguage*) [Bobr 77], LOOPS [Stef 85], ROME [Carr 89] et VIEWS [Davi 87], qui illustrent différentes façons de représenter des *vues* dans les représentations par objets.

II.2.2.1. KRL

KRL [Bobr 77] est un modèle développé comme outil de construction de systèmes pour la compréhension du langage naturel. Il est le premier système à reconnaître qu'un objet pourra être vu de plusieurs façons, selon la perspective de l'observateur.

Il existe, dans KRL, divers modes qui permettent de décrire un objet : i) appartenance à une classe, ii) relation dont l'objet est un membre, iii) identification unique, ou iv) rôle dans un événement. Chaque description est une liste de descripteurs ayant des facettes associées. Chaque descripteur est une caractérisation indépendante de l'objet associé. Par exemple, on peut décrire une «Personne» dont le nom est «Jean» de la

manière suivante : «Jean, employé de l'UDM», «inscrit au cours INF1800», etc. Chaque ensemble de description forme une *unité* KRL qui sert de référence mentale aux entités du monde réel.

Dans KRL, il y existe sept types d'*unités* : *Basic*, *Specialization, Individual, Abstract, Manifestation, Relation* et *Proposition*. Les trois premières unités permettent le traitement des *vues*. En particulier, les classes *Basic* établissent une première partition de l'univers du discours; les classes *Specialization* sont des sous-classes d'une classe *Basic* ou d'une autre classe *Specialization*, et les classes *Individual* décrivent des entités uniques du monde des individus. Pour l'exemple de la «Personne» cité ci-haut, nous pouvons dire que l'*individu* «Jean», demeurant à l'adresse X, est membre de la classe «Employé» qui est une *Spécialization* de la *classe de base* «Personne».

II.2.2.2. LOOPS

Contrairement à KRL où tous les attributs sont stockés dans le même objet qui décrit l'individu, LOOPS [Stef 85] divise l'instance même (l'objet) en plusieurs composantes (ses différentes *perspectives ou vues*).

Dans LOOPS, un objet et ses *perspectives* est une sorte d'objet composite, les composants étant les différentes *perspectives*. Chaque *perspective* est un objet indépendant auquel on peut s'adresser directement en lui envoyant des messages. Cette indépendance permet de définir des attributs de même nom ayant des divers sens dans plusieurs *perspectives*.

Les *perspectives* sont toutes liées à l'objet. Le lien qui existe entre les *perspectives* se veut être un lien conceptuel en raison du fait que toutes nomment le même objet. À part le lien qui existe entre une instance et ses *perspectives*, chaque *perspective*, en tant qu'instance indépendante, appartient à une classe à laquelle elle est liée par le lien «est-un». Pour traiter les *perspectives*, LOOPS utilise deux classes abstraites (appelées *mix-ins*) : *Node* et *Perspective*. Une classe décrivant des objets qui sont des *perspectives* pour d'autres objets doit être définie comme sous-classe de la classe *Perspective*, et une classe qui décrit des objets ayant des *perspectives* est sous-classe de la classe *Node*.

Dans LOOPS, la modélisation des *perspectives* présente de nouvelles idées sur le sujet de composition et d'intégration. D'une part, le traitement indépendant des *perspectives* permet de regarder un objet en fonction d'un *point de vue* sans être saturé par l'information des autres *points de vue*, fait qui reflète la façon de travailler d'un groupe interdisciplinaire : il regarde le modèle en fonction d'un *point de vue* et en tire des conclusions, puis il change de *perspective* pour compléter ou vérifier l'information acquise. D'autre part, la dynamique permise par cette indépendance au niveau de la création et de la modification des *perspectives* permet de stocker une instance dans la *base* et de la manipuler sans avoir accès à l'information complète de toutes les *perspectives*. Malgré ses apports, la représentation des *perspectives* dans LOOPS exige l'ajout d'éléments artificiels, comme les liens spéciaux et les classes abstraites. Le fait de mélanger les *mixins Perspectives* et *Node* avec le graphe des classes obscurcit le modèle [Stef 85].

II.2.2.3. ROME

ROME [Carr 89] est un langage hybride qui combine les principes des représentations par objets et ceux des langages à objets.

Dans ce langage, les instances sont créées par des méthodes de leur classe, appelée *classe d'instanciation*. Elles lui sont reliées pour toute la durée de leur vie. Vu au niveau représentation, un objet (instance) peut être incomplet. Il peut être vu selon divers *aspects* et peut évoluer pendant la durée de sa vie. Ainsi, pour traiter le côté représentation des instances, ROME introduit la notion de *classe de représentation.*

Une *classe de représentation*, par opposition à une *classe d'instanciation*, n'a pas la fonctionnalité d'instanciation. Les *classes de représentation* sont des spécialisations d'une *classe d'instanciation;* elles décrivent des sous-ensembles d'individus de la *classe d'instanciation* ayant certaines propriétés spécifiques. Une instance appartient à une seule *classe d'instanciation* (mono-instanciation) mais elle peut avoir plusieurs *classes de représentation*. Les *classes de représentation* filtrent, en un certain sens, les éléments de la *classe d'instanciation* et définissent ainsi l'ensemble des *vues* des instances. Une instance qui évolue peut changer de *classes de représentation*, mais elle reste toujours

attachée à sa *classe d'instanciation*. Dans l'exemple de «Personne», «Jean» est, avant tout, une *instance* de «Personne»; il est aussi *représentant* des classes «Employé» et «Étudiant». Cette idée de faire appel à une *classe d'instanciation* pour l'information de base de l'objet et à plusieurs *classes de représentation* pour ses *perspectives* est aussi proposée par le système PINOL [Ngu 91], en vertu duquel on établit une distinction entre le type d'une instance contenant son information structurelle de base et ses classes qui représentent les différentes perspectives.

Le modèle ROME offre plusieurs avantages. D'une part, il établit la différence conceptuelle qui existe entre la classe qui définit un concept et les classes qui ajoutent des contraintes pour décrire des sous-catégories de ce concept. D'autre part, il permet la liaison d'une instance à plusieurs classes grâce à la représentation multiple.

D'autres systèmes ayant des caractéristiques semblables à celles de ROME ont été développés. Citons les exemples du langage FROME, du langage CROME et du système TROPES. FROME est utilisé pour la représentation dynamique et multiple des cadres (*frames*) des données [Dekk 92]. Il est basé sur la hiérarchie de classes représentée par la relation d'héritage. Dans CROME, la représentation multiple consiste à déclarer qu'un objet est une instance de plusieurs classes. Chacune des classes est interprétée à titre de description d'un *point de vue* particulier [Debr 97]. Dans le système TROPES [Rech 90], la représentation multiple est dédiée à la classification des objets selon le *point de vue*. Chaque objet est décrit par un concept qui, à son tour, est décrit par un certain nombre de *points de vue*. Chaque *point de vue* est structuré sous forme d'une arborescence de classes dont la racine est le concept.

II.2.2.4. VIEWS

VIEWS [Davi 87] est un langage de représentation de connaissances qui combine des idées, des schémas et des réseaux sémantiques. L'unité de description est *"view"* qui comporte trois types d'objets dans une organisation de réseau : les *parties*, les *relations* et les *contraintes*.

- Une *partie* peut être vue comme un noeud d'un réseau pouvant être une valeur simple ou une autre *vue*. La *partie* ne comporte a priori pas de sémantique, celle-ci étant donnée par des relations entre les *parties*.

- Les *relations* sont comme les attributs des schémas : elles peuvent comporter diverses sémantiques.

- Les *contraintes* limitent la portée de la structure d'une *vue*.

Dans VIEWS, une *perspective* est représentée par une *vue*, c'est-à-dire un ensemble d'éléments liés entre eux. La «Personne», qui fait l'objet de l'exemple précédent, comporte trois *vues* : une *vue* pour chacune des perspectives «Employé» et «Étudiant» et une *vue* globale qui regroupe ces perspectives.

II.2.3. Limitations

En représentation de connaissances, les travaux sont faits pour les langages de description et non pas pour les langages de programmation. Or, ce sont ces derniers qui sont utilisés pour le développement des applications. En plus, le comportement dynamique est resté au stade de l'expérimentation.

Ces travaux ont cependant suscité des approches *orientées-objet*. Citons l'exemple des langages de programmation CROME et FROME, qui ont donné lieu à quelques nouvelles idées pour les approches *orientées-objet* et pour certains langages de programmation. Ces travaux constituent également la base de nouvelles recherches.

II.3. Vues en orientation-objet

Lors du développement d'un système logiciel, il faut définir diverses entités, structures et fonctionnalités. Dans le cas d'un grand système, la division en sous-systèmes est une solution qui facilite le travail de développement. Cela peut être fait sur une base de services ou de fonctions que le système doit offrir, ou sur une base d'objets, avec leurs services, que le système manipule.

Dans le cas d'une modélisation *orientée-objet*, un cycle de développement doit être suivi pour bien mettre sur pied l'application. Le développement ne se ter-

mine pas au moment de la mise au point, il continue pour englober l'étape d'entretien. Cette dernière est la plus coûteuse en terme de temps et d'argent. Ainsi, les travaux effectués pendant les phases de développement sont liés entre eux et influencent la qualité du système tout au long de sa durée de vie. Les phases de développement sont divisées en i) une phase d'étude des besoins, dans laquelle les besoins des utilisateurs de l'application sont définis, ii) une phase d'analyse, qui consiste à identifier les diverses entités de l'application, les différents objets et leurs fonctions, iii) une phase de conception qui fait appel aux résultats de la phase d'analyse pour préparer la phase d'implantation, iv) une phase d'implantation et v) une phase de test et d'entretien.

Pour un développement *orienté-objet*, nous retrouvons les notions d'objets et de classes ainsi que leurs relations, et ce, à toutes les étapes du développement. Cette continuité dans l'utilisation des mêmes notions facilite les ajouts et les modifications à apporter à des étapes préliminaires. C'est à l'étape d'analyse que les objets sont définis; dans le cas des autres étapes, une telle définition est maintenue.

D'une manière générale, l'interface de l'objet avec l'extérieur est définie statiquement selon les besoins des divers usagers. Dans un cas pratique, cependant, les besoins peuvent changer d'un usager à un autre. Ainsi, il est préférable d'utiliser des sous-ensembles d'opérations pour chaque type d'usagers au lieu de rendre toutes les opérations accessibles à tous les usagers. Chaque sous-ensemble reflète une *vue* particulière d'un groupe d'utilisateurs.

Une *vue* est définie comme une collection d'états et de comportements qui reflètent une perception particulière du monde vu par des usagers particuliers [Gold 80], [Harr 93]. Elle permet une concentration sur une seule partie des informations de l'objet. Les objets (entités) pouvant jouer plusieurs *rôles* peuvent avoir des perceptions diverses et simultanées [Stef 86], [Pern 90]. Les divers comportements des objets peuvent changer durant la vie des objets; des données et des fonctionnalités peuvent être acquises ou perdues.

La nécessité d'introduire des *vues* en *orientation-objet* se manifeste aussi quand des applications partageant les mêmes objets (ayant plusieurs fonctionnalités) sont développées séparément et ont besoin d'être fusionnées, et quand on veut offrir divers droits d'accès à plus d'un usager. On doit introduire cette nouvelle notion de *vues* tôt dans le cycle de vie du développement pour assurer concordance et continuité entre les étapes de développement.

L'étude de la notion de *vues* dans les systèmes informatiques a touché toutes les phases de développement d'un logiciel. Certaines recherches portent sur l'étude des *vues* au niveau de la phase d'implantation. Dans ce cas, les propriétés existantes des langages de programmation et de nouvelles propriétés sont étudiées pour supporter la notion de *vues*.

Les *vues* peuvent être introduites à des phases préliminaires à l'implantation, celles de l'analyse et de la modélisation. En ce qui a trait à l'exécution, l'introduction et l'utilisation des *vues* permettent d'offrir un comportement dynamique des objets.

Dans ce qui suit, nous présentons la notion de *vues* introduite au niveau de la modélisation, de l'analyse, de la conception et de la programmation.

II.3.1. Vues en modélisation

La modélisation par modèle de *rôle* (ou *vue*) a été introduite pour offrir une réutilisabilité de la phase d'analyse de diverses applications. Elle consiste à définir les objets ayant différents *rôles* à jouer face aux autres objets avec lesquels ils entrent en contact. Chaque *rôle* peut être défini indépendamment des autres et peut, par la suite, être intégré dans une ou plusieurs classes qui représentent les objets. Cette intégration peut être faite par les relations de spécialisation-généralisation.

La restriction qui nous est imposée lorsque nous lions un objet à une même classe et à un seul comportement limite le contexte d'interprétation et la flexibilité de modélisation du comportement des objets qui jouent des *rôles* différents pendant leur exécution. Pour sa part, le concept de *rôle* est utilisé pour modéliser séparément divers comportements d'un même objet [Pern 90], [Alba 93], [Hoyd 93]. Il permet d'offrir des comporte-

ments variés liés à une même classe. Un *rôle* est défini comme étant un ensemble de propriétés importantes d'un objet qui lui permettent de se comporter d'une façon particulière avec un certain ensemble d'objets [Kris 96]. Ainsi, un *rôle* identifie un objet, dont il est partie intégrante, par l'ensemble des propriétés qu'il définit.

Velho et Carapuca [Velh 93] ont présenté le modèle sémantique d'objet (SOM) qui est un modèle d'analyse et de conception développé en tenant compte des *rôles*. Dans ce modèle SOM, un *rôle* est un ensemble de caractéristiques partagées par divers objets. Ces *rôles* sont définis comme «Phases». Une «Phase» est constituée d'attributs d'événements, d'un cycle de vie et de propriétés. Elle peut être reliée à d'autres «Phases» en précisant certaines caractéristiques au moyen de la clause *"Becomes with"*. Une classe fait appel aux «Phases» qui lui sont utiles. Les figures 2.2 et 2.3 représentent un exemple de «Phase» et de «classe». La classe «Person» fait appel à la phase «Taxpayer» .

```
PHASE Taxpayer
  LOCAL DESTINATION PHASES
    NON_EXEMPTED_TAXPAYER
      Becomes with
        Registration
    EXEMPTED Becomes with
        NON_EXEMPTED_TAXPAYER  : : Exemption
  ATTRIBUTES
  EVENTS
  LIFE CYCLE
  INVARIANT
  ...
END - - Phase Taxpayer
```

Figure 2.2 : Phase Taxpayer

Dans la figure 2.2, le *rôle* (*Phase*) «*Taxpayer*» est défini et est utilisé dans la figure 2.3 pour la classe «Person». Ce même *rôle* peut être réutilisé pour d'autres classes.

Le travail de conception des *rôles* et de leur intégration dans les classes doit être fait au niveau de l'analyse. Une fois les *rôles* définis et composés avec les classes, on ne peut plus en ajouter. Dans ce cas, les modèles de *rôles* sont des entités réutilisables au niveau de l'analyse, mais ne le sont pas au niveau de la conception ou de l'implantation.

```
CLASS Person
  SHARED DESTINATION PHASES
    TAXPAYER
      Becomes with tax_registration
  LOCAL DESTINATION PHASES
    MAN Becomes with [sex="M"]
  ATTRIBUTES
    Sex   : STRING
  ...
  EVENTS
    Tax_Registration (dist   : STRING)
      Do TAXPAYER : : Registration End;
END - - Class Person
```

Figure 2.3 : Classe «Person»

Au niveau de l'implantation, la modélisation par *rôles* consiste à concevoir une relation d'héritage ou de paramétrisation entre les *rôles* qui sont représentés sous forme de super-classes de la classe à laquelle ils sont reliés [Reen 96], [VanH 96]. Les *rôles* peuvent aussi être directement intégrés et identifiés dans la classe [Alba 93].

Dans l'exemple de la figure 2.4, une voiture peut avoir les *rôles* i) «production», qui modélise toutes les étapes de construction d'une voiture et ii) «accident», qui modélise le comportement d'une voiture après un accident. En plus, le rôle de la classe «Voiture» en est un *de base* qui modélise ses *caractéristiques de base* représentées par le «numéro de matricule», le «propriétaire», la «couleur», etc. Pour chaque *rôle* identifié, il y existe un ensemble de règles à suivre et de messages à échanger. Lors de la création d'une instance de la classe «Voiture», le *rôle de base* lui est accordé automatiquement, alors que d'autres *rôles* peuvent y être spécifiés explicitement. Ainsi, même si les *rôles* sont déjà intégrés dans la classe au moment de sa définition et qu'ils lui sont propres, on peut en utiliser un sous-ensemble. Par ailleurs, ces *rôles* ne peuvent pas être réutilisés pour d'autres classes.

```
Class =
 (Voiture,
   R0= <Rôle_de_base,
     properties={(nu_matricule,int)
                 (propriétaire,string)
                 (couleur, string) ... etc.)},
     states={active,non_active},
     messages={->met-marche,add-rôle,resume-role,
                suspend-rôle ...},
     rules = {
     rule0,1 :constraint(in-rôle(production)=>
             not in_role(insurance))
     rule0,2 :msg (<- terminate-rôle(production))=>
             msg(->add-rôle(insurance)),...}>,
   R1 = <production,
     properties = {...},
     states = {...},
     messages = {...},
     rules = {...} >,
   R2 = <accident, ...>,
   )
```

Figure 2.4 : Définition de la classe «voiture»

Dans [Reen 96], les auteurs parlent de la méthode d'analyse par modèle de *rôles* OORAM (*Object Oriented Role Analysis Method*) comme étant un cadre d'application pour plusieurs méthodologies *orientées-objet*. Cette méthode permet de modéliser l'interaction des objets en utilisant le modèle de *rôle* pour la séparation des domaines d'intérêts (*"concerns"*). Elle consiste à faire correspondre un modèle de *rôle* pour chaque domaine d'intérêt défini par un ensemble d'opérations. La différence qui existe entre une classe et un *rôle*, c'est que la première décrit les objets ayant des caractéristiques communes et le deuxième fait état des objets ayant les mêmes fonctions dans un cadre donné. Un modèle de *rôle* peut être créé en suivant les étapes suivantes :

1- définition du domaine d'intérêt,

2- identification de la nature des objets,

3- identification des *rôles*,

4- identification des séquences de messages possibles entre les objets,

5- définition de la structure générale,

6- définition des interfaces, et

7- définition des comportements des *rôles*.

Dans le modèle OORAM, les fonctions sont isolées et étudiées selon les *rôles* qui leur sont attribués.

Prenons l'exemple d'une classe «Personne» qui est définie dans une application pour une agence de voyage. Les objets de cette classe peuvent se voir attribuer les *rôles* de «Comptable», de «Voyageur» et de «Gestionnaire». Selon les *rôles* qu'ils jouent, les objets peuvent interagir et s'échanger des messages. Par exemple, un «gestionnaire» peut échanger des messages avec un «comptable». Un «voyageur» peut aussi avoir des interactions avec un «comptable». Ainsi, l'interaction avec le «comptable» peut présenter diverses interfaces (voir figure 2.5).

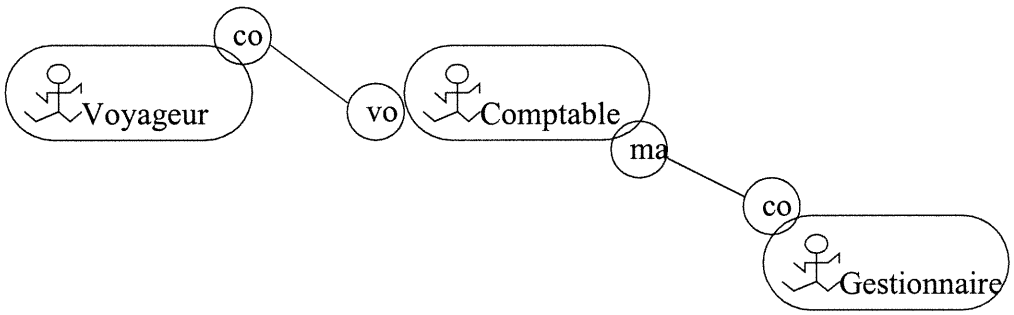


Figure 2.5 : Interaction des trois rôles

Synthèse

Les recherches menées sur les modèles de *rôles* ont permis d'identifier plusieurs ensembles de propriétés relatifs à divers domaines d'intérêts [Kris 96], [Reen 96]. Cela permet à l'objet de se comporter de diverses manières, selon les objets avec lesquels il entre en contact. Ainsi, un *rôle* représente une perspective d'un ensemble d'objets [Ferb 89], [Rich 91].

La notion de *rôle* est introduite à deux niveaux dans le cycle de développement d'un logiciel : i) au niveau de l'analyse, où l'ensemble des *rôles* est défini et intégré dans les propriétés des objets, et ii) au niveau de la programmation, où l'on utilise la relation

d'héritage pour relier les *rôles* aux classes ou la paramétrisation des classes avec leurs *rôles*. Les *rôles* définis au niveau d'analyse peuvent être réutilisés pour plusieurs applications. Cependant, une fois les *rôles* intégrés dans les classes, ils ne peuvent plus en être séparés [VanH 96].

Le *rôle* ajoute également des propriétés extrinsèques aux propriétés intrinsèques des objets [Kris 96]. Ces dernières sont les propriétés de base (de la super-classe dans le cas de la relation d'héritage) et ne sont pas relatives à un domaine spécifique. Les propriétés extrinsèques sont reliées à un *rôle* donné et utilisées dans des situations précises.

II.3.2. Vues en programmation

II.3.2.1. Mécanismes supportant une variante des vues

La spécialisation multiple [Carr 90], la multi-instanciation [Rieu 92] et l'agrégation sont des mécanismes supportés par les langages de programmation et pouvant être utilisés pour supporter la notion de *vues*. De plus, l'encapsulation (ou regroupement) des sous-ensembles de méthodes avec leurs règles correspondantes [Pern 90] et le regroupement des caractéristiques des classes en des sous-ensembles de caractéristiques (ou filtres) [Velh 93], [Heil 90] sont d'autres moyens qui facilitent l'implantation des *vues*.

Les diverses approches de modélisation de *vue*s font appel aux approches basées sur la hiérarchie des classes, en particulier les relations d'héritage et d'agrégation qui permettent un regroupement de caractéristiques dans des agrégats ou des sous-classes. Nous représentons ci-après ces deux relations en précisant la façon dont elles peuvent être utilisées comme moyen de représentation des *vues*. Nous décrirons aussi les interfaces qui sont un autre moyen de représentation des vues.

II.3.2.1.1. Relation d'héritage

L'héritage est le procédé de partage ou de factorisation de l'information. Il permet d'étendre, d'entretenir et de réutiliser des logiciels en spécialisant des classes préexistants par de nouvelles classes [Duco 89]. La spécialisation opère de haut en bas par

addition de nouvelles propriétés ou substitution de valeurs dans des propriétés déjà définies. L'intérêt de l'héritage pour la spécialisation est de permettre une conception incrémentale d'une application [Carr 89].

La relation d'héritage est représentée sous forme d'un arbre hiérarchique dont la racine est la super-classe de toutes les autres sous-classes (figure 2.6). La sous-classe est définie pour spécialiser les objets représentant la même super-classe par rapport à une utilisation dans un contexte plus spécifique. Lorsque l'héritage est simple, chaque objet, sauf la racine, admet un unique père. Or, il arrive qu'une sous-classe hérite de plusieurs super-classes, comme dans l'exemple de la sous-classe «ÉtudiantÉtrangerGradué», qui hérite à la fois d'«ÉtudiantGradué» et d'«ÉtudiantÉtranger» (figure 2.6) et qui représente le cas de l'héritage multiple.

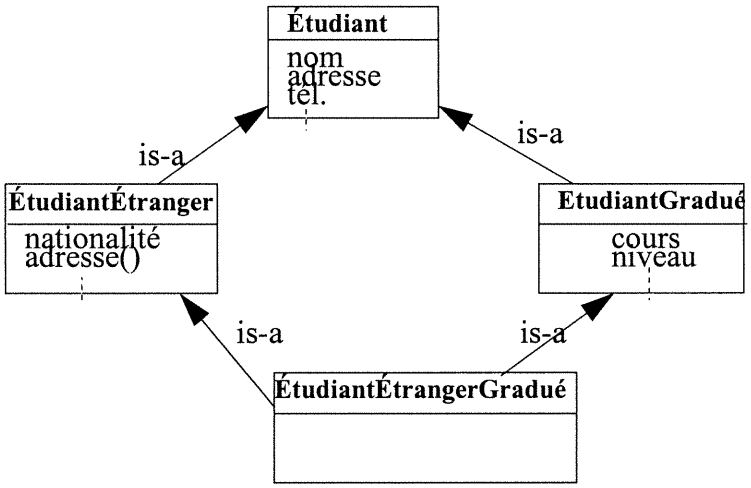


Figure 2.6 : Hiérarchie d'héritage

Dans la figure 2.6, les sous-classes «ÉtudiantÉtranger» et «ÉtudiantGradué» héritent des attributs et des méthodes de la classe «Étudiant», et définissent d'autres attributs qui leurs sont spécifiques. Un objet «étudiantgradué» est une spécialisation d'un objet «étudiant». Les caractéristiques de la super-classe «Étudiant», telles que «nom», «adresse» et «tél», sont réutilisées dans les sous-classes. Une redéfinition de la méthode «adresse()» est présentée dans la sous-classe «ÉtudiantÉtranger».

Des travaux sur la hiérarchie de l'héritage multiple ont été faits pour faciliter la représentation des relations entre les classes et leurs super-classes. Le problème de représentation est surtout traité en intelligence artificielle lorsque l'on parle d'une multiplicité de concepts. Ce problème de représentation des hiérarchies est aussi important dans les langages *orientés-objet*, dans lesquelles un assemblage d'objets est nécessaire pour simplifier l'interaction des usagers. Dans [Case 93], la hiérarchie peut être représentée sous forme de treillis dont la racine est la super-classe et les noeuds sont les sous-classes reliées par la relation d'ordre. Les feuilles du treillis constituent les classes instanciables les plus élémentaires.

L'héritage comme moyen de représentation des vues

La relation d'héritage, étant une façon de traduire les classifications, permet une approche du type «multi-expertise», chaque expert ayant sa propre classification selon le *point de vue* qu'il a face aux objets. Chaque *point de vue* concerne un aspect particulier de l'objet ou une *perspective* (ou *vue*) de l'objet [Card 88].

La relation d'héritage est utilisée pour la modélisation par *rôles*, où les divers *rôles* sont considérés comme différentes super-classes d'une *classe de base*. Dans la figure 2.7, nous définissons les *rôles* «Gestionnaire» et «Comptable» comme étant les super-classes de la classe «Agent». Cette dernière comporte deux *vues* (rôles) différentes qui co-existent.

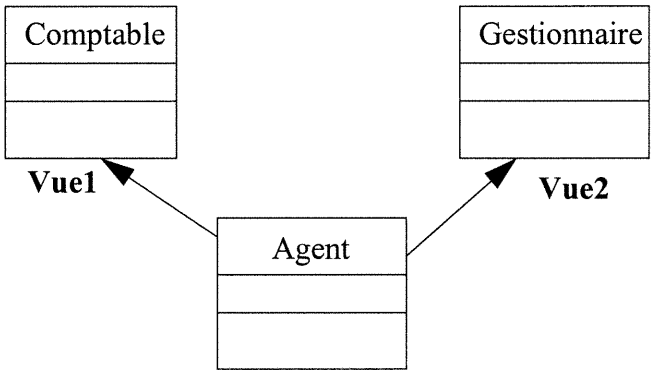


Figure 2.7 : Représentation des vues par héritage multiple

L'héritage multiple consiste à combiner différentes spécialisations en faisant leur jointure dans une même classe, ce qui correspond à la jointure de plusieurs *points de vue* ou *vues* multiples d'une même classe.

Une fois qu'une relation d'héritage a défini une hiérarchie de classes, ces dernières ne peuvent plus être séparées. Ainsi, le comportement dynamique des objets et le contrôle d'accès à leurs attributs ne sont pas possibles, ce qui ne répond pas aux exigences que nous avons établies en vertu des *vues*.

II.3.2.1.2. Relation d'agrégation

L'agrégation peut être définie comme un regroupement d'entités. Elle consiste en une hiérarchie structurelle obtenue par la décomposition récursive d'un tout en sous-éléments jusqu'à l'obtention des éléments non décomposables. Cette hiérarchie s'est révélée à travers le lien *"is-part"* des réseaux sémantiques.

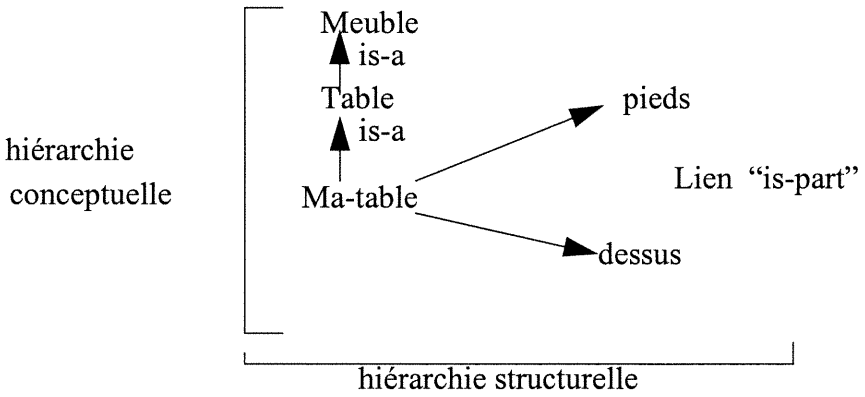


Figure 2.8 : Hiérarchies conceptuelle et structurelle

La figure 2.8 représente un exemple des relations d'héritage et d'agrégation sous forme de hiérarchies conceptuelles et structurelles. Dans cet exemple, «Ma-table», qui est un «Meuble», est composée de pieds et d'un dessus qui représentent des ensembles d'attributs pour un objet (agrégat) de «Ma-table».

L'agrégation traduit une implantation structurée d'un objet en termes de sous-objets correspondant à des aspects particuliers. Les mêmes problèmes relatifs à la hiérarchie conceptuelle (relation d'héritage) existent aussi au chapitre de la hiérarchie struc-

turelle. Citons les exemples des conflits des valeurs des attributs, communs à plusieurs composants, de la gestion des messages envoyés à un même composant et de la cohérence des données.

L'agrégation comme moyen de représentation des vues

La relation d'agrégation est utilisée dans les langages LOOPS [Stef 86] et JANUS [Garl 87] comme moyen d'implantation des *vues*. Par exemple, dans le langage JANUS, chaque *point de vue* est défini par une classe avec ses divers attributs et opérations. La classe qui présente différents *points de vue* sera une agrégation de toutes les *vues* (les composantes) possibles. La figure 2.9 schématise la représentation des *vues* «Comptable» et «Gestionnaire» pour la classe «Agent» selon la relation d'agrégation. C'est là une représentation simple, bien que les données relatives aux divers *points de vue* soient toujours accessibles. La relation d'agrégation, comme la relation d'héritage, offre certaines facilités de regroupement des données relatives à une *vue* spécifique. Cependant, cette relation n'offre pas la flexibilité de manipulation et l'indépendance des *vues*.

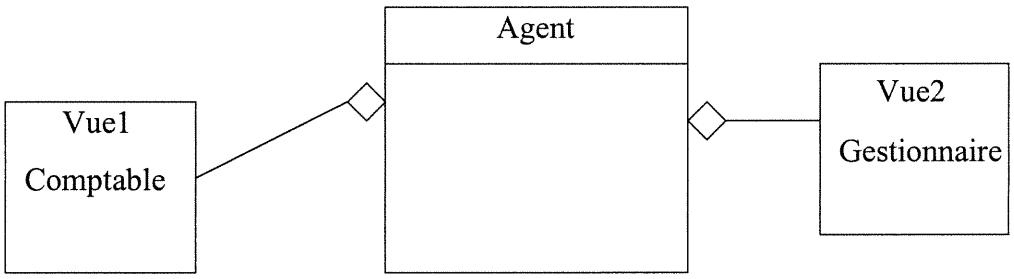


Figure 2.9 : Représentation des vues par agrégation

II.3.2.1.3. Les interfaces

Une interface représente une activité d'une instance de classe et détermine le comportement de cette instance par un ensemble de méthodes [Rieh 98]. Les interfaces sont les clés d'accès à une classe et peuvent définir des privilèges d'accès ou des fonctionnalités particulières à accéder. La variété des interfaces qui modélisent divers *rôles* et comportements correspond à la variété des *vues* que les instances de la classe peuvent avoir.

Ces *vues* sont considérées comme des *filtres* capables d'accommoder les terminologies spécifiques à chaque type d'usage.

Cette notion de *vue* généralise les interfaces de visibilité dans les langages de programmation, tels que dans Smalltalk («protégées» versus «publiques»), C++ («privées», «publiques», et «protégées»), Java («interface») et Eiffel («feature»).

Dans Smalltalk, C++ et Eiffel les interfaces se limitent au contrôle d'accès aux divers types de propriétés. Un ensemble particulier d'usagers y a accès, et cet accès se fait à partir des différentes *interfaces*. Cependant, pour le langage Java, la notion d'*interfaces* est conçue pour permettre l'acheminement dynamique des appels de méthodes à l'exécution [Naug 98]. Dans Java, les interfaces ne sont pas des filtres mais des constructions à part. Elles sont reliées explicitement aux classes qui peuvent définir leurs méthodes et les manipuler. Elles ont surtout l'avantage d'être utilisées par des classes différentes qui n'ont pas de lien hiérarchique.

Les interfaces, telles qu'elles sont utilisées dans les langages de programmation, offrent certains avantages que l'on souhaiterait avoir avec l'introduction de la notion de *vues*. Ces avantages se résument par i) le contrôle d'accès qui assure la sécurité des données, ii) le regroupement des propriétés qui est relié aux *aspects fonctionnels* des différents composants et iii) la réutilisation.

Les interfaces peuvent ne pas se limiter seulement au contrôle d'accès et représenter des caractéristiques particulières pour divers usagers [Shil 89]. L'environnement Pecan [Reis 84], le langage Trellis/Owl [Scha 86], le système PIE de Xerox [Gold 81], les travaux de Shilling et Sweeney [Shil 89], Hailpern et Ossher [Hail 90] et de Bielak et Mckim [Biel 93] sont des exemples de modélisation de *vues* sous forme d'interfaces multiples.

PIE utilise un mécanisme de *vue* appelé *perspective*. Une *perspective* est un objet particulier qui est attaché à un autre objet. Elle sert à décrire l'objet d'un *point de vue* particulier, et plusieurs *perspectives* peuvent être attachées à un seul objet. L'objet est manipulé par rapport à la *perspective* choisie par le client, et les messages sont traités par rapport à la *perspective* plutôt qu'à celui de l'objet lui-même. Ainsi, les variables

d'instance d'un objet sont divisées par *perspectives* et chaque *perspective* fournit sa propre *interface*.

Pour Shilling et Sweeney, trois étapes sont ajoutées au paradigme *orienté-objet* pour supporter la notion de *vues* [Shil 89] :

1-définition des interfaces multiples,

2-contrôle de la visibilité des variables d'instance et

3-plusieurs copies de la variable d'instance pour la même instance d'objet.

1) Interfaces multiples

La figure 2.10 nous donne l'exemple d'une classe avec deux interfaces. I_1 et I_2 sont deux interfaces comportant les méthodes {A, B et C} et {A, B et D} respectivement. Une instance de la classe peut être accédée par l'une ou l'autre des interfaces.

Class VOOP :

I_1						I_2
	A	=> M1	Variables d'instance	M1 <=	A	
	B	=> M2		M4 <=	B	
	C	=> M3		M3 <=	D	

Figure 2.10 : Classe avec deux interfaces (I_1 and I_2)

2) Visibilité des instances

La multiplicité des interfaces permet plusieurs comportements d'un même objet, ce qui implique une diversité de visions. Il est nécessaire de relier les variables d'instance à leurs classes correspondantes et de spécifier leur interface correspondante. Une des façons de faire est d'étiqueter chaque instance au moyen de l'identificateur de l'interface par laquelle elle est accessible (figure 2.11).

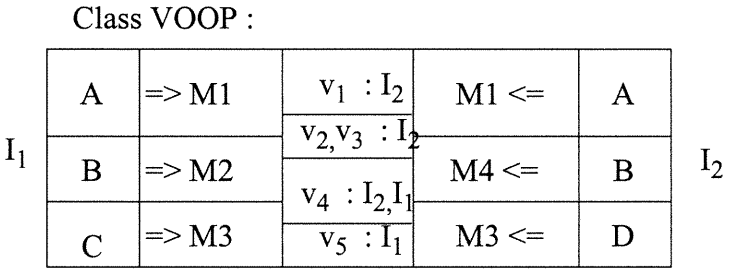


Figure 2.11 : Variables d'instance étiquetées

3) Plusieurs copies des variables d'instance

Les interfaces peuvent être activées plusieurs fois, ce qui implique la nécessité d'avoir plusieurs copies des variables d'instance. Chaque copie correspond à une activation différente de la même interface. Elle est référée par l'identificateur de l'interface et par celui de l'activation. Dans la figure 2.12, les variables d'instances sont représentées et reliées à une interface et à une activation spécifiques. La même variable d'instance peut apparaître deux fois et pour deux activations différentes. Cette diversité de copies implique des problèmes au niveau de la cohérence des données. Une copie d'une instance doit conserver les mêmes valeurs que celles des autres copies des autres activations. Dans [Shil 89], les copies multiples sont définies simplement pour référer les interfaces et les activations; elles ne changent pas de valeurs.

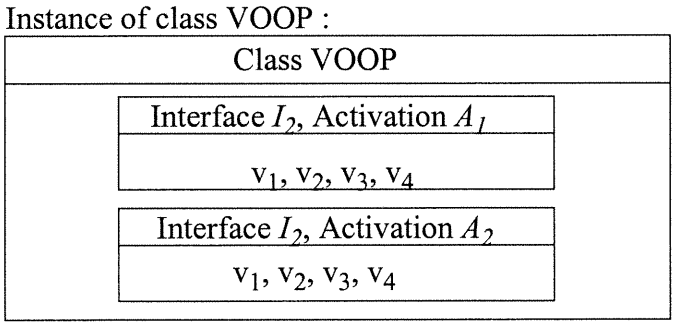


Figure 2.12 : Plusieurs copies des variables d'instance

Somme toute, la modélisation des *vues* par interfaces multiples offre une flexibilité au niveau de l'accès à des méthodes spécifiques. Cependant, on ne traite ni des problèmes de cohérence des multiples copies des instances, ni des conflits au niveau de la hiérarchie des classes et son implication dans l'héritage des interfaces. En outre, les interfaces doivent co-exister au même moment, ce qui implique que l'aspect dynamique des *vues* ne peut pas être offert.

II.3.2.2. Langages supportant une variante des vues

Plusieurs langages de programmation implantent la notion de *vue* sous diverses variantes comme, par exemple, *point de vue*, *perspective*, *interface*, *rôle* ou *filtre*. Nous présentons ci-après certains de ces langages et leurs constructions spécifiques aux *vues*.

II.3.2.2.1. Le langage Eiffel

Eiffel est un langage orienté-objet qui, selon Bielak et McKim [Biel 93], peut supporter la notion des *vues*, bien qu'avec certaines restrictions.

En Eiffel, une classe définit un ensemble d'attributs, de méthodes et d'invariants [Meye 91]. L'instruction *"feature"* permet de regrouper un ensemble de méthodes relatives à un contexte particulier. Cet ensemble de méthodes est accessible par le truchement des clients de *"feature"*, tel que déclaré dans sa signature (figure 2.13). Les *vues* étant des cas particuliers de comportements, elles sont déclarées comme cas particuliers de *"feature"*. Pour expliquer l'implantation des *vues* en Eiffel, prenons l'exemple de la classe «Cat», avec ses méthodes regroupées en méthodes pour le «propriétaire», pour le «vétérinaire» ou pour tout autre usager (ANY).

```
Class CAT
creation give_birth_to
ensure birth_year=current_date.year
       cardiovascular_status=5
       digestive_status=5; not ready_for_medicine
       not is_purring; not is_content
       hungry; dirty
feature {ANY}
  age : INTEGER
    ensure Result=current_date.year-birth_year
  name :STRING
feature {OWNER}
  hungry, dirty, is_content :BOOLEAN;
  feed(f :FOOD)
    require hungry
    ensure not hungry
  change_name(n :STRING)
    ensure name=n
feature{VETERINARIAN}
  ready_for_medecine :BOOLEAN
  give_heart_medecine
    require cardivascular_status<5
      ready_for_medecine
    ensure cardivascular >5
end - class CAT
```

Figure 2.13 : Exemple du langage Eiffel

II.3.2.2.2. Le langage VBOOL

Marcailloux et al. ont travaillé sur une extension du langage Eiffel pour mieux supporter la notion de *vue* tout en assurant leur réutilisation [Marc 94]. Ils ont ajouté au langage Eiffel une nouvelle relation *"seen_as"*, semblable à la relation d'héritage mais conçue pour déclarer des *points de vue* (ou des *vues*). Bien que la relation d'héritage offre des possibilités de représentation de comportements variés d'une classe par ses sous-classes, elle s'avère insuffisante pour traiter les problèmes de sécurité des données entre les sous-classes et, par conséquent, pour implanter les *vues* et *les points de vue* [Marc 93]. De là l'importance de pouvoir faire appel à la relation *"seen_as"*. Un exemple expliquant la relation *"seen-as"* offrant la notion de *vues*, nous est donné dans la fig-

ure 2.14.

```
Class ordinateur

seen_as machine_électr
seen_as outil_apprent
seen_as produit_vente

end; class ordinateur
```

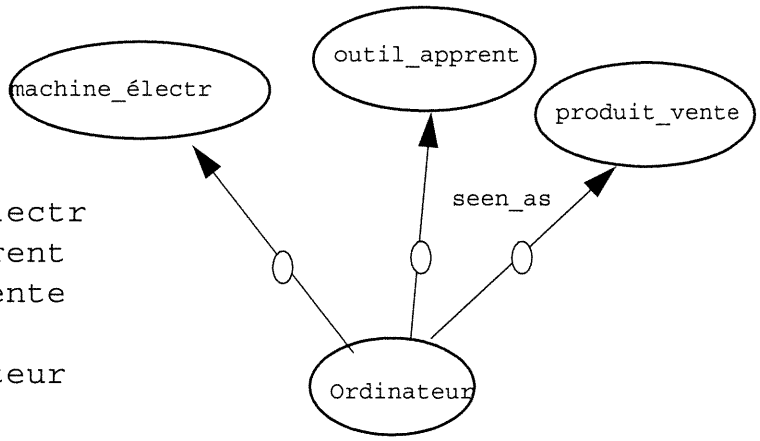


Figure 2.14 : Déclaration de trois vues de la classe «ordinateur»

Dans la figure 2.14, la classe «ordinateur» présente trois *vues* différentes : «machine_électr», «outil_apprent» et «produit_vente» . La relation *"seen-as"* fait correspondre les *vues* à la classe.

```
flexible class Station
feature ANY
  nom   : STRING
  -- partie concernant les vues
seen-as Ordinateur_Configuré
seen-as Noeud_Réseau
seen-as ...

view_feature Ordinateur_Configuré, Noeud_Réseau
  -- visibilité autorisée pour toute classe ayant
  -- accès à la station vue selon ces deux vues
  afficher is do ... end;
```

Figure 2.15 : Exemple d'utilisation du langage VBOOL

Dans VBOOL, chaque *vue* est définie séparément en utilisant les instructions du langage Eiffel. Une classe «flexible» peut faire appel à plusieurs *points de vue* s'il y a correspondance. Par exemple, la classe «Station» comporte les vues «ordinateur_configuré» ou «ordinateur_configurable» et la vue «noeud_reseau» (figure

2.15). Dans cet exemple, l'instruction *"view_feature"* définit un ensemble de méthodes accessibles par des objets «Ordinateur_Configuré» et «Noeud_Réseau».

Bien que le langage VBOOL ait apporté des améliorations au chapitre de l'implantation des *vues*, il n'a pas traité les problèmes relatifs aux conflits dans les méthodes des *vues* interdépendantes et dans le traitement des cas où un objet peut comporter plusieurs *vues* simultanément.

II.3.2.2.3. Le langage JAVA

Le langage Java mérite d'être cité comme un langage qui supporte une variante des vues grâce à ses interfaces. Dans ce langage, une interface est définie comme une description d'un comportement. À la différence d'une classe, la définition d'une interface ne contient ni attributs ni corps des méthodes. Elle consiste à faire les déclarations des méthodes qui définissent le comportement qu'elle représente. La figure 2.16 montre un exemple de définition de deux interfaces, «FCamion» et «OCamion».

```
interface FCamion {
   double gettauxAmortissement();
   double getvaleurRésiduelle();
}
interface OCamion {
   void scheduleEntre(int,int);
}
```

Figure 2.16 : Exemple d'interfaces Java

Une interface peut être implantée par plusieurs classes ce qui leur permet d'offrir le même comportement. De plus, une classe peut implanter plusieurs interfaces. Dans ce cas, elle doit fournir une implantation des méthodes de ces interfaces offrant ainsi leurs comportements. Ces derniers correspondent aux *vues* multiples que la classe peut avoir. Une application peut manipuler un objet avec un sous-ensemble des interfaces que sa classe implante. La figure 2.17 présente un exemple d'une classe qui implante les deux interfaces et une application qui manipule un objet de cette classe avec seulement une interface.

```
class Camion implements FCamion , OCamion {
 ...
 double gettauxAmortissement(){...}
 double getvaleurRésiduelle(){...}
 void scheduleEntre(int,int){...}
 ...
}

class ApplicationFinance {
 private double totalvaleurRésiduelle;
 ....
 public void ajoutInventaire(FCamion fc) {
   totalvaleurResiduelle = fc.getvaleurRésiduelle();
   ....
   ....
 }
}

class Monsystème {
 static void main () {
  ApplicationFinance af = new ApplicationFinance(0;
  FCamion fc = new Camion();
  af.ajouteInventaire(fc);
  OCamion oc = new Camion();
  af.ajouteInventaire (oc);
 }
 ...
}
```

Figure 2.17 : Une classe avec deux interfaces

Dans l'exemple de la figure 2.17, la fonction «ajouteInventaire» de la classe «ApplicationFinance» est invoquée en premier pour un objet de la classe «Camion» avec seulement l'interface «FCamion» et elle l'est une deuxième fois pour un objet de la même classe avec l'interface «OCamion». Ainsi, grâce aux interfaces que la classe «Camion» implante, nous avons pu manipuler ses objets avec des comportements différents. Cependant, cela exige que toutes les interfaces doivent être implantées dans la classe à l'avance.

Bien que les approches présentées ci-haut offrent une variante des *vues*, elles comportent diverses problématiques, et certains de nos objectifs pour les *vues* n'y sont pas

supportés. Par exemple, l'ajout et la suppression dynamique des *vues* ne sont offerts dans aucune des approches présentées. Les *vues* doivent être définies à l'avance et attachées aux classes avant la compilation. Dans certaines approches, les *vues* sont des parties intégrées de la classe; il est par conséquent difficile de les ajouter ou de les supprimer sans modifier toutes les classes qui leur sont reliées.

II.3.2.3. Modularisation pour les séparations des "concerns"

La séparation des *"concerns"* est à la base de la programmation par *vues*. Elle consiste à identifier plusieurs domaines d'intérêts, pour un même ensemble d'objets, relatifs aux différentes *vues* que ces objets peuvent avoir. Cette séparation peut se faire à différentes phases de développement d'une application. L'objectif est de garder une concordance entre les étapes et de retarder le plus possible, dans le cycle de développement, la fusion des *"concerns"* pour un même ensemble d'objets. Dans ce qui suit, nous présentons les nouvelles approches qui traitent la séparation des *"concerns"*.

II.3.2.3.1. Les « sujets »

La subjectivité est une notion proposée par Harrison et Ossher pour faciliter le développement des applications [Harr 93] en tenant compte des états des objets qui interagissent. Elle est l'implication naturelle du monde physique et du langage humain. Par exemple, un objet dans le monde physique n'a pas qu'une apparence, et un même mot possède différentes connotations. En *orienté-objet*, un objet répond au même message de la même façon sans tenir compte de son émetteur. La plupart des langages *orientés-objet* sont objectifs. Un objet a le même comportement, peu importe le contexte dans lequel il interagit. Cependant, un objet subjectif est celui qui offre plusieurs comportements dépendamment de ses clients. Contrairement aux approches de modélisation par modèles de *rôles* et par *interfaces*, les *sujets* offrent une flexibilité et une indépendance au niveau de leur définition et de leur liaison avec les classes. Ils ne sont pas liés à l'avance à une classe, mais définis séparément et appelés quand c'est nécessaire. De plus, ils offrent l'aspect dynamique de changement, d'ajout ou de suppression de *sujet*.

La programmation par *sujet* diffère des autres approches par le fait que les *sujets* sont développés séparément et forment une tranche compilable et possiblement exécutable. Ces *sujets* sont composés plus tard dans le cycle de développement d'une application. Plusieurs *sujets* peuvent définir différentes parties des mêmes classes, voire même offrir diverses possibilités d'implantation pour un même comportement. La composition de *sujets* s'opère en fusionnant les définitions partielles des classes, résolvant les conflits avec des règles de composition explicites [Ossh 95]. En principe, la composition *a posteriori* de *sujets,* grâce à des règles explicites, permet d'intégrer des applications dont la «collaboration» n'a pas été planifiée. Cette composition permet ainsi de décentraliser la définition des données et l'implantation des applications [Harr 93]. En pratique, il faut observer certains édicats de programmation [Ossh 95] et même pré-planifier la composition [Mili 96].

L'approche proposée par Harrison et Ossher répond assez bien au besoin d'intégration d'applications développées par des tierces parties, mais comporte deux problèmes majeurs : i) les conditions de «composabilité» sont assez restrictives, et ii) la composition des *vues* se fait en pré-compilation, ou pré-chargement, ce qui impose le même nombre de *sujets* sur tous les objets de la même classe.

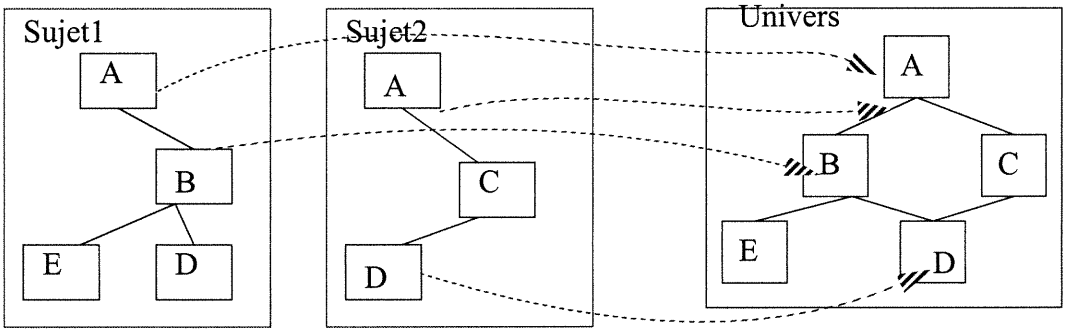


Figure 2.18 : Composition de sujets

Un *sujet* correspond à une collection d'états et de comportements qui reflètent une perception du monde vu pour une application spécifique. Il peut être composé de plusieurs classes ayant en commun une même perception. Plusieurs *sujets* sont donc déve-

loppés séparément et peuvent avoir les mêmes objets qui interagissent. Ces objets offrent différents comportements, selon le *sujet* auquel ils appartiennent. Les différents *sujets* peuvent être fusionnés pour former un *univers* (voir figure 2.18) constitué de l'ensemble de tous les *sujets* et des différents objets assortis de leurs comportements multiples. Des règles de composition de *sujets* doivent être respectées lors de la formation d'un univers. Ces règles peuvent être basées sur l'union des interfaces et l'accumulation ou la composition des implantations. Cette fusion peut être faite au niveau source ou objet, ce qui avantage l'approche par *sujets* face à d'autres approches.

L'interaction d'un objet dans un univers est contrôlée par un identificateur qui lui est propre, quel que soit son *sujet* actif. Cet identificateur est utilisé pour référer un objet globalement dans une application. Un ensemble d'identificateurs est prédéfini pour éviter des conflits entre certains *sujets*. Lors de la création d'un objet, il faut en spécifier le *sujet* et l'activer; cette *activation* fait correspondre l'objet à un *sujet* spécifique.

Un des avantages de la programmation *orientée-sujet* réside dans le fait que plusieurs *sujets* opèrent séparément sur les mêmes objets sans qu'un sujet ne prenne en considération les détails de la relation entre ses objets et d'autres *sujets*. Dans un *sujet*, un objet est lié à une classe qui définit son comportement et ses opérations. Une *opération* est représentée par un tuple (a,op,p), où «a» est l'activation qui demande l'opération, «op» est l'opération demandée et «p» est la liste des paramètres de l'opération. Avec l'approche par *sujet*, toute application C++ peut être «subjectifiée». Il suffit de faire correspondre les classes à un même *sujet* et à des règles de composition valables pour l'application. Une fois les *sujets* et les *univers* composés et formés, on ne peut plus ajouter de *sujets* dynamiquement. De plus, selon les règles de composition, un objet appartenant à plusieurs classes comporte toutes les fonctionnalités de ses classes. Ainsi, les *vues* (*sujets*) sont disponibles simultanément pour tous les usagers.

II.3.2.3.2. Les « aspects »

La *programmation par aspect* permet, elle aussi, un développement décentralisé des *aspects* mais n'offre pas de changement de comportement au moment de l'exécution.

Les *aspects*, comme les *sujets*, sont développés séparément et peuvent être utilisés pour plusieurs classes. Ils sont intégrés dans les classes pour changer leur comportement.

Un *aspect* est un module utilisé pour faciliter la gestion des propriétés des objets qui peuvent se croiser et s'intégrer pour une application donnée. Ce module définit un ensemble de caractéristiques et un comportement qui peuvent être appliqués à un ensemble d'objets. Cela permet une définition indépendante de comportements et facilite leur réutilisation. Les *aspects* sont utilisés au niveau de la conception et au niveau de l'implantation. En ce qui a trait à la conception, ils facilitent la définition et la liaison d'un ensemble de propriétés et offrent un faible couplage entre les modules définis. Au niveau de l'implantation, un *aspect* peut être appliqué à une composante de programme pour lui ajouter un comportement particulier.

Un *aspect* est défini comme un module indépendant spécifiant des propriétés et des contraintes sur son application à une classe donnée. L'ajout d'un *aspect* à une classe lui accorde ses propriétés. Cet ajout se fait automatiquement grâce à un support pour la *programmation par aspect* [Kicz 99].

En résumé, la modélisation par *aspect* et la modélisation par *sujet* offrent une définition isolée et indépendante des applications et des modules et facilite leur intégration. L'utilisation, la définition et l'intégration des *aspects* et des *sujets* sont réalisables lors de la programmation. Il existe des outils de support pour la *programmation par sujets* et la *programmation par aspect*. Dans ce qui suit, nous citons des caractéristiques et quelques insuffisances des deux approches :

- Les *aspects* sont définis et peuvent être ajoutés à un objet donné au niveau de l'analyse et du codage; cela, ne peut pas se faire dynamiquement au moment de l'exécution.
- Les *aspects* sont intégrés facilement à un module sans une préoccupation des conflits des caractéristiques; c'est un bloc de code définissant un comportement qui vient s'ajouter à la classe. Pour les *sujets*, certaines règles de composition doivent être respectées pour faire l'intégration.

- L'intégration de plusieurs *aspects* pour une même classe peut générer des problèmes au niveau de l'identité des opérations répétées. Pour résoudre ce problème, dans le cas de la programmation *orientée-sujets*, certaines liaisons entre les objets et leurs sujets sont définies. Par exemple, la relation d'activation dans la programmation par sujet détermine l'ensemble des opérations que peuvent offrir les objets.
- La liaison entre les *aspects*, les *sujets* et les classes est statique durant le cycle de vie des objets.

```
Class Point {
   int _x = 0;
   int _y = 0;
   void set(int x, int y) {_x=x; _y = y;}
   void setX(int x) {_x=x;}
   void setY(int y) {_y=y;}
   int getX() {return _x;}
   int getY{return _y;}
};

aspect Inspection {
   introduce void Point.print() {
          System.out.println("_x =" + _x +
                                "_y =" + _y);
   advise void Point.set(int,int) {
   static before{
          System.out.println("Début de set x et y");
          }
   }
}
```

Figure 2.19 : Programmation par aspects

Dans l'exemple de la figure 2.19 l'application de l'*aspect* «Inspection» à la classe «Point» lui offre un comportement dans lequel un message sera imprimé avant la modification de x et y, et une nouvelle fonction «print()» est introduite pour imprimer x et y.

II.3.2.4. Sommaire sur les vues en OO

Dans ce qui suit, nous résumons les différents avantages des *vues* en *orientation-objet* :

1- La concentration sur un sous-ensemble spécifique d'attributs et de méthodes relatifs à la *vue* en cours [Heil 90].

2- Une flexibilité au niveau de la manipulation et de la définition des objets ayant des comportements multiples. Cette flexibilité se concrétise au niveau de l'affectation de plusieurs combinaisons de *vues* à un même objet. Par exemple, si dans l'exemple de la classe «Étudiant» et des *vues* «Gradué» et « Sous-gradué », on ajoute les *vues* «Étranger» et «Résident», un étudiant peut être gradué et étranger ou gradué et résident.

3- Les *vues* offrent des changements dynamiques dans le comportement des objets qui peuvent changer, ajouter et supprimer de *vues* dans le temps.

4- Le regroupement des données spécifiques à un domaine précis et la facilité de leur affectation aux objets facilitent la réutilisation en nous permettant de les réutiliser dans des contextes variés.

II.4. Conclusion

Dans ce chapitre, nous avons présenté un aperçu des approches qui existent pour les *vues*. Pour chacune de ces approches, il y a certaines restrictions que nous résumons dans ce qui suit :

- Pour les bases de données relationnelles, une *vue* est centralisante, et la mise à jour à travers des *vues* ne peut pas se faire, et aucun ajout des *vues* n'est pas possible au moment de l'exécution.
- Le modèle de *rôle* consiste à introduire la notion de *vue* au moment de l'analyse. Cette notion disparaît au niveau de la conception et de l'implantation. Ceci implique que la réutilisation séparée ainsi la dynamicité des *rôles* ne sont pas offertes.
- La modélisation par *interface* consiste à définir à l'avance les *vues* des objets. Toutes les *vues* co-existent simultanément et ont une relation fixe. De plus, le

nombre d'interfaces est limité. Dans le cas du langage Java, le nombre d'interfaces n'est pas fixé et l'utilisation des *interfaces* est plus ou moins flexible. Cependant, une fois qu'un ensemble d'interfaces est relié à une classe, toutes les interfaces nous sont accessibles simultanément.

- Avec les relations d'héritage et d'agrégation, le développement est centralisé, les *vues* co-existent en même temps et celles-ci doivent être définies à l'avance.
- Les approches par *sujets* et par *aspects* offrent une dynamicité au niveau de l'intégration de *vues* lors de la conception et de la compilation, et non au moment de l'exécution. Elles souffrent de certains problèmes de composition et d'intégration de *vues* (sujets ou aspects).

La co-existence des *vues*, leur pré-définition et le moment de leur composition influencent la qualité du développement des applications avec *vues*. Par exemple, le développement décentralisé et l'intégration des applications développées par des tierces parties exigent une utilisation dynamique des *vues* propagée jusqu'au moment de l'exécution. Ainsi, la liaison entre l'objet et ses *vues* doit être retardée le plus possible pour avoir plus de flexibilité au niveau du développement, du test et de la maintenance des applications.

Un objet change constamment de *rôles* durant son cycle de vie. Une personne peut avoir un *rôle* d'étudiant, un *rôle* d'employé, un *rôle* de ministre ou un *rôle* de père de famille, tout comme elle peut perdre certains de ces *rôles* pour les acquérir plus tard ou en acquérir de nouveaux. Cela doit être modélisé pour les objets à manipuler. Il faut leur offrir la possibilité d'acquérir et de perdre des *vues* dynamiquement. Ce changement n'est pas défini à l'avance et peut différer d'un objet à un autre. C'est pour cela qu'il faut l'offrir au moment de l'exécution. Les nouvelles approches de programmation par *vues* présentées dans ce chapitre assurent l'intégration des *vues* au moment de l'implantation en définissant et instanciant les classes qui restent constantes le long du cycle de vie de leurs objets. Notre approche assure l'attachement et le détachement dynamique des *vues*. La liaison entre l'objet et ses *vues* est retardée jusqu'au moment de l'exécution grâce au préprocesseur, qui s'occupe de l'implantation de méthodes dont le comportement change selon les *vues* disponibles.

Notre travail consiste à proposer une nouvelle définition de la notion de *vues* susceptible d'englober les cas généraux d'utilisation et de réutilisation des applications développées. Pour la réalisation pratique, des extensions sont ajoutées aux langages de programmation *orientés-objet* afin de supporter la notion de *vues*. Ces extensions seront présentées dans les chapitres qui suivent.

Chapitre III

Programmation par vues : Principes et exemples

L'objectif de notre recherche est de permettre de définir des *aspects fonctionnels* et de les relier, selon le besoin, aux classes pour ainsi offrir à leurs instances des comportements multiples. Les aspects fonctionnels, appelés *vues,* applicables à un objet définissent l'ensemble des comportements que ce dernier peut avoir [Darg 96]. Pendant sa durée de vie, chaque objet est tenu à supporter un certain comportement de base et un ensemble variable de *vues* où chaque *vue* représente un comportement particulier. Notre approche implante, dans une certaine mesure, la notion de classification multiple et dynamique, par laquelle un objet peut appartenir à plusieurs types simultanément et changer de types durant sa vie [Mart 92].

Outre la dynamicité de l'ensemble de comportements supportés par un objet, un aspect important de notre recherche, propre à notre approche, est celui la distinction entre deux concepts : *vues* et *points de vue*. Un *point de vue* représente un aspect fonctionnel générique indépendant de la classe cible [Mili 99a]. Une *vue* est alors le résultat de l'application d'un *point de vue* à une classe donnée. Les *points de vue* peuvent donc être développés de façon décentralisée et sont reliés aux classes selon les besoins des applications. Un *point de vue* est une structure générique qui doit identifier deux types de caractéristiques qui peuvent être appliquées à une classe : i) les caractéristiques «fournies», qui sont celles reliées à un contexte particulier et qui caractérisent la *vue* que le *point de vue* peut donner à la classe et ii) les caractéristiques «requises» qui définissent les exigences sur les classes auxquelles le *point de vue* peut être appliqué. Les caractéristiques fournies et requises par un point de vue sont exprimées sous la forme d'attributs et de méthodes. L'application d'un *point de vue* à une classe donne une *classe vue* adaptant ces attributs et méthodes à la classe. Ainsi, durant l'exécution, un objet sera constitué d'un noyau fonctionnel (appelé *objet de base*) et d'un ensemble de *vues* (voir figure 3.1).

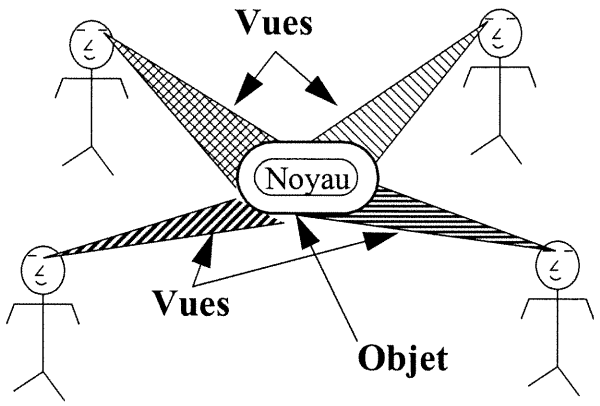


Figure 3.1 : Objet et vues

Dans ce chapitre, nous présenterons les principes de la *programmation par vues* et la motivation des choix importants qui la sous-tendent. Dans la section III.1, nous décrivons les notions de *vues*, de *points de vues*, et justifions le besoin de supporter leurs deux niveaux d'abstraction. La section III.2 explique les principes de la génération des classes *vues* à partir des *points de vues*. La section III.3 décrit une approche structurelle à la représentation d'objets avec *vues*; c'est l'approche que nous avons adopté pour notre implantation de programmation par *vues* pour le langage C++ (voir chapitre VI). La section III.4 décrit le comportement typique d'un objet avec *vues*; un exemple complet montrant le code de *vues* y est présenté. Nous concluons dans la section III.5.

III.1. Principes théoriques

III.1.1. Les vues

Dans notre approche, nous distinguons, pour tous les objets, l'existence d'un comportement de base et de comportements reliés aux *vues* attachées aux objets [Mili 99].

Le comportement de base est défini par des caractéristiques qui ne changent pas avec le changement de l'application. Nous disons de ces caractéristiques qu'elles sont «intrinsèques». Elles constituent le noyau de base qui est commun à tous les objets de la

classe.

Les comportements reliés aux *vues* sont ceux définis par des caractéristiques particulières aux *vues*. Ces caractéristiques définissent les interactions des objets avec les autres objets dans les contextes des *vues*. On dit donc qu'elles sont «extrinsèques».

Prenons l'exemple d'un objet d'une classe «Personne» dont les attributs sont le «nom», l'«adresse» et la «date de naissance». La *vue* «Employé» de cet objet peut lui ajouter les attributs «salaire» et «poste» et la méthode «getcharge()» (pour la charge du travail). La *vue* «Étudiant», elle, lui ajoute les méthodes «getcours()» et «getcharge()» (pour la charge des études). Quand ces deux *vues* sont attachées à l'objet, l'ensemble de ses attributs et de ses méthodes est constitué des caractéristiques de base et des caractéristiques des deux *vues*.

Pour manipuler les *vues*, nous avons défini les fonctions d'attachement et de détachement ainsi que d'activation et de désactivation des *vues*.

L'attachement consiste à ajouter le comportement de la *vue* au comportement de base de l'objet. Le détachement consiste à faire l'inverse; ainsi, le comportement de la *vue* ne sera plus offert par l'objet et il n'y aura plus de lien entre la *vue* et l'objet.

La désactivation d'une *vue* consiste à rendre le comportement, qu'elle ajoute à l'objet, inaccessible. À la différence de la fonction de détachement, le lien entre l'objet et la *vue* existe toujours. La fonction d'activation consiste à rendre le comportement d'une *vue* accessible.

Ces fonctions permettent la manipulation dynamique des *vues* et font en sorte que les objets d'une même classe peuvent avoir des comportements différents.

L'état d'un objet est l'union ($\oplus$) d'un état de base, appelé aussi «état élémentaire», et des états inhérents aux *vues* «actives». Nous définissons alors l'interface I d'un objet comme suit :

$$I = I_0 \oplus I_{v,1} \oplus I_{v,2} \oplus ... \oplus I_{v,n}$$

Où I_0 est l'interface de l'objet de base, et $I_{v,i}$ l'interface de la ième *vue*. Notez que les différentes interfaces ont des points communs avec I_0. Cette décomposition en plusieurs interfaces peut varier d'une application à une autre et représente les comportements pos-

sibles de l'objet. Schématiquement, nous pouvons représenter l'intersection des interfaces par la figure 3.2. Le nombre d'interfaces varie avec le nombre de vues que la classe peut avoir. L'interface de l'objet est délimitée par la ligne foncée dans la figure.

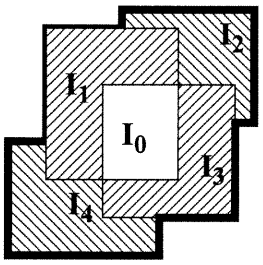


Figure 3.2 : Différentes interfaces pour une même classe

III.1.2. Les points de vues

Le principe des *points de vue* se rapproche du mécanisme de patron de classes de C++ [Stro 97]. Un *point de vue* définit une tranche fonctionnelle compilable et applicable à un ensemble de comportements de base. C'est un type paramétrique ayant une syntaxe qui ressemble à celle des classes. Les *points de vue* visent à définir, une seule fois, les ensembles de caractéristiques fonctionnelles et les appliquer, selon le besoin, aux classes pour générer les *vues*. Les *points de vue* seront réutilisables pour plusieurs applications et applicables à plusieurs classes.

Comme nous l'avons précisé, nous identifions, pour un *point de vue*, deux types de caractéristiques : les caractéristiques «fournies» et les caractéristiques «requises». Les caractéristiques «fournies» sont celles que le *point de vue* ajoute à la *classe de base*. Par ailleurs, les caractéristiques «requises» sont celles qui correspondent à des attributs de la *classe de base* et représentent les exigences sur l'application du *point de vue* sur la classe. Ces attributs sont utilisés dans le code des méthodes du *point de vue*. Quand ces attributs ne sont pas offerts par la classe, le *point de vue* ne peut pas lui être appliquée.

Prenons l'exemple d'une classe «École». Un objet de cette classe a les caractéristiques : «nombre_de_salles», «prix», «liste_professeurs» et «liste_étudiants». Les objets

de cette classe peuvent être manipulés par plusieurs applications. Ils peuvent ainsi comporter divers *points de vue* (figure 3.3). D'un *point de vue* «Financier», les caractéristiques fournies aux objets sont «dépenses», «revenus» et «salaires». Une caractéristique requise peut être «prix» si l'on veut dire que le *point de vue* «Financier» doit porter sur des objets monnayables, c'est-à-dire sur des capitaux. D'un *point de vue* «Opération» les caractéristiques fournies sont «tâche», «durée» et «coût». Pour ce *point de vue*, l'ensemble des caractéristiques requises peut être vide pour dire qu'il peut être appliqué à toutes les classes. Quand les deux *points de vue* sont reliés à la classe «École», ses objets auront, à part le comportement qu'elle définit, les comportements définis dans les deux *points de vue*. Cela correspond à deux *vues* différentes des mêmes objets.

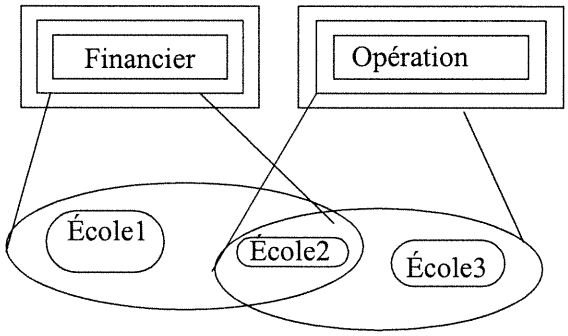


Figure 3.3 : Différents points de vue de l'école

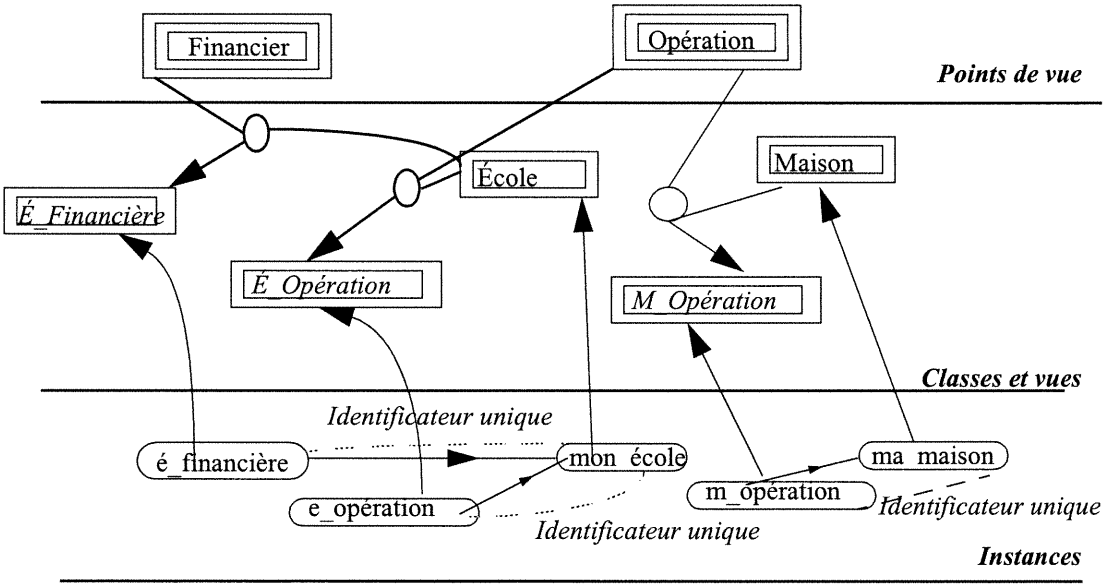
Syntaxiquement, dans la définition des *points de vue* nous identifions la partie **requires** et la partie **provides,** pour les caractéristiques «requises» et «fournies» respectivement.

Un *point de vue* est applicable à une classe si sa partie **requires** est satisfaite par la classe, c'est-à-dire que si pour chacun de ses membres nous pouvons lui faire associer un correspondant dans la *classe de base*. L'application du *point de vue* à une *classe de base* génère une *classe vue* dont les attributs et les fonctions sont ceux de la partie **provides** du *point de vue*. Si un attribut ou une méthode de la partie **requires** du *point de vue* est utilisé dans la partie **provides**, celui-là sera remplacé par son correspondant dans la *classe de base*.

III.1.3. Les niveaux d'abstraction

Les *classes*, les *vues* et les *points de vues* appartiennent à des niveaux d'abstraction différents. Les *points de vues* sont au niveau d'abstraction supérieur et leur définition est indépendante de celle des classes. Ils peuvent être appliqués à ces dernières pour générer les *vues*.

Les *vues* sont des classes dont les instances sont reliées à des instances des *classes de base*. Un objet d'une *classe vue* définit un nouveau comportement pour l'objet de base auquel il est attaché. Les classes et les *vues* appartiennent à un autre niveau d'abstraction, et elles sont instanciables pour générer les objets et leurs *instances de vues*. Les objets de base et les objets *vues* se trouvent au niveau d'abstraction inférieur. Dans la figure 3.4, les *points de vue* «Financier» et «Opération» sont appliqués simultanément aux classes «École» et «Maison».



Légendes:

: Dérivation d'une vue à partir d'un point de vue et d'une classe
: Instance de classe
: Liaison entre l'objet de base et sa vue
: Liaison entre l'instance de vue et son objet de base

Figure 3.4 : Les différents niveaux d'abstractions

III.2. Génération des vues

III.2.1. Déclaration de la vue

Pour utiliser un *point de vue*, l'usager doit d'abord le particulariser à une classe donnée. Ceci est spécifié par un énoncé **viewdef** qui donne un nom à la *vue* obtenue en combinant le *point de vue* et la *classe de base*. Au besoin, l'énoncé pourra faire la liaison entre les attributs de la partie **requires** et des attributs de la *classe de base*. Un exemple d'une déclaration de *vue* est le suivant:

```
viewdef : E_Financière École as Financier();
```

Cette déclaration donne un nom à la *vue* et invite à la génération de la classe *vue* dont les instances seront reliées aux objets de la *classe de base*. La classe *vue* obtenue est unique pour le *point de vue* et la *classe de base* en question. Son code contient des références aux attributs et aux méthodes de la *classe de base* et la caractérise par rapport aux autres classes *vues* générées pour le même *point de vue* et d'autres *classes de base*. Son nom est utilisé lors de la manipulation de ses instances pour les attacher, détacher, activer et désactiver.

III.2.2. Correspondance de la partie « requires »

Pour générer une *vue* à partir d'une *classe de base* et d'un *point de vue,* la partie **requires** de ce dernier est analysée pour trouver les correspondants de ses attributs et de ses fonctions dans la *classe de base*. Pour chacun des attributs, il faut lui faire correspondre un attribut, dans la *classe de base*, ayant le même type. Dans le cas où plusieurs attributs satisferaient cette exigence, nous en choisissons un aléatoirement. Nous avons fait ce choix car nous supposons que dans le cas d'un manque de clarté, l'utilisateur doit spécifier lui-même les correspondances. Il peut toutefois changer la correspondance directement dans la classe *vue* générée.

Pour la correspondance des fonctions, on peut faire appel à plusieurs méthodes. Une des méthodes les plus populaires est celle de l'analyse de l'ensemble des entrées de la fonction et de son ensemble de sorties. Si pour un même ensem-

ble d'entrées deux fonctions donnent les mêmes sorties, nous pouvons conclure que ces deux fonctions offrent le même comportement. Ainsi, pour chacune des fonctions de la partie **requires**, nous définissons un ensemble de pré-conditions ainsi qu'un ensemble de post-conditions et essayons d'y faire correspondre une fonction. Notons que la définition des pré-conditions et des post-conditions n'est pas toujours représentative. En effet, deux fonctions différentes peuvent offrir un même comportement pour un ensemble particulier d'entrées. Elles peuvent aussi satisfaire à un ensemble particulier de conditions sans offrir exactement le même comportement.

Une autre méthode de correspondance de fonctions consiste à faire l'analyse de leurs signatures. Selon cette méthode, une fonction «f» correspond à une fonction «g» si les types de ses paramètres sont des super-types de ceux de «g», et le type qu'elle retourne est un sous-type du type de retour de «g». En d'autres termes, l'entrée de «f» est une généralisation de l'entrée de «g», tandis que sa sortie est une spécialisation de celle de «g». Nous pouvons garantir par cette règle que pour l'ensemble des entrées de «f» nous aurons des sorties spécialisées de «g» [Card 88]. Nous avons, dans notre approche, choisi une variante simple de cette méthode de correspondance. Pour chaque fonction dans la partie **requires**, nous lui faisons correspondre une fonction de la *classe de base* avec le même type de retour et le même ensemble de types de paramètres. Nous avons convenu de ce choix car il est toujours sujet à des modifications explicites selon les besoins de l'utilisateur dans le code de la *vue* générée.

Dans le cas où plusieurs fonctions pourraient correspondre les unes aux autres, le *"mapping"*, c'est-à-dire la correspondance, se fait par indication explicite de l'usager. Lors de la déclaration d'une *vue,* l'usager peut spécifier, dans une liste de correspondance, les fonctions et les attributs qui se correspondent. La liste est formée de couplets de fonctions et d'attributs (<fonct_base, fonct_point_vue>,<att_base, att_point_vue>, ...).

Dans l'exemple de la figure 3.5 la fonction «valeur()» dans le *point de vue* «Entretien» correspond à la fonction «prix()» dans la classe, et l'attribut «prix» dans la classe correspond à l'attribut «valeur». Cette dernière correspondance est spécifiée explicitement dans la déclaration de la *vue*.

```
class Véhicule {
private:
        double taux;
        double prix;
public:
        double getprix() { return prix;}
        void settaux(double d){taux = d;}
        ...
}
// définition des points de vue
viewpoint Entretien {
requires:
        double valeur;
        double valeurÉstime();
        ...
provides:
        double tarif;
        void settaux(double var) { ... }
        void setvaleur() {valeur = valeur*tarif;}
        ...
}
viewpoint Finance {
requires:
        double calculValeur();
provides:
        double valeur;
        void setvaleur() { valeur= calculValeur;}
        void calculAmortissement() {
        calculValeur();
        }
        ...
};
// déclaration de la vue
viewdef:unevue véhicule as Entretien(<prix, valeur>);
```

Figure 3.5 : Correspondance des membres

III.2.3. Définition des attributs et des fonctions de la classe *vue*

En appliquant le *point de vue* à la *classe de base*, c'est sa partie **provides** qui définit le nouveau comportement de la *vue* générée. Les instances de la classe vue sont reliées à celles de la *classe de base* par une variable d'instance qui figurent dans la *classe vue*. Cette variable est utilisée pour faire référence aux attributs et fonctions de la *classe de base* quand c'est nécessaire. En effet, la partie **requires,** déjà analysée, définit un sous-ensemble du comportement assuré par la classe et utilisé par la partie **provides**. Quand les membres de la partie **provides** appellent des membres de la partie **requires**, l'appel sera remplacé par le membre correspondant dans la *classe de base* qui est identifié dans la liste de correspondance définie lors de l'analyse de la partie **requires**. Un appel de délégation se fait à travers la variable d'instance de *la classe de base* vers le membre approprié. Ainsi, la fonction de la partie **provides** sera exécutée dans le contexte de la *classe de base*.

Les instances des classes *vues* ne sont pas créées explicitement. Elles sont créées par attachement aux instances des *classes de base*. Chaque instance de la classe *vue* est liée à une instance de la *classe de base* particulière. Cette dernière peut avoir un ensemble d'instances de ses différentes *vues*. Ces instances font partie des variables de la *classe de base* et nous permettent d'accéder aux variables et aux fonctions des *vues*. Ainsi, les membres de la *vue* générée et les membres de la *classe de base*, sont accessibles. Les membres de la *classe de base* le sont directement et les membres des *vues* par délégation par l'intermédiaire des instances des *vues*. L'utilisateur n'a pas à spécifier si la fonction demandée est celle d'une *vue* ou de la *classe de base*. C'est notre pré-processeur qui s'occupe de la recherche de la *vue* qui l'offre et de faire la délégation.

Étant donné que plusieurs *points de vue* peuvent être appliqués simultanément à une *classe de base*, et qu'ils sont développés indépendamment les uns des autres, nous pouvons avoir des situations où une même fonction est répétée dans deux *vues* (la fonction «setvaleur()» dans la figure 3.5). Dans ce cas, la fonction peut être la même comme elle peut avoir une identification semblable à celle d'une

autre fonction sans offrir nécessairement le même comportement. Lors de l'appel de cette fonction il faut savoir quelle version appliquer ou il faut exécuter un comportement qui est la composition des différents comportements offerts par la « même » fonction.

Nous résumons les différentes situations que peut avoir un attribut ou une fonction de la partie **provides** du *point de vue* :

1- il (elle) figure seulement dans le *point de vue*,

2- son identificateur est le même que celui d'un membre de la *classe de base* mais signifie autre chose, ou

3- il (elle) représente la même chose qu'un autre membre défini dans la *classe de base* ou dans une autre *vue* de la classe mais est défini (e) différemment.

Pour le premier cas, le membre de **provides**, tel quel, fera partie des membres de la *vue*. Pour les deux autres cas, il faut prévoir des règles de composition pour leur donner une nouvelle définition [Ossh 95].

III.2.4. Présence de plusieurs vues

Une attention particulière est accordée à la simultanéité des *vues* d'un même objet du fait qu'une *vue* affecte le comportement interne de l'objet et, par conséquent, d'autres *vues* de cet objet. L'identificateur de l'instance de la *classe de base* est le lien commun entre toutes les *vues* et nous permet d'accéder aux différentes instances de ses *vues*. Certains problèmes reliés à la simultanéité des *vues* sont à traiter et nous les illustrons dans l'exemple de la figure 3.6.

Lors de la définition des *vues* de la classe «Personne» pour les *points de vues* «Emploi» et «Étude», il faut prendre en considération le fait que la méthode «getAdresse()» figure dans les deux *points de vue* ainsi que dans la *classe de base*. Quand cette méthode est adressée, la bonne définition doit être exécutée. Cette définition peut être celle de l'objet de base (ou n'importe quelle autre version des autres *points de vues*), comme elle peut être dérivée de toutes les versions offertes

et actives.

```
class Personne {
private :
    char * nom;
    char * adresse;
public:
    char* get_nom() { return nom;}
    void getAdresse(){cout << adresse;}
    void printInfo() { cout << nom << adresse;}
    ...
};

viewpoint Emploie {
requires :
    char * identification;
    ...
provides :
    double salaire;
    char * t_adresse;
    double calculImpôt() {...}
    char * getAdresse() {return t_adresse;}
    void printInfo()
    {cout<< identification << t_adresse << salaire ;}
    ...
};

viewpoint Étude {
requires:
    char * identificateur;
    .....
provides:
    char * e_adresse;
    char* cours;
    char * getAdresse() {return e_adresse}
    void printInfo()
    {cout<< identificateur << e_adresse << cours ;}
    ...
};
```

Figure 3.6 : Exemple de plusieurs vues

Ainsi, pour résoudre le problème de la multiplicité des définitions des fonctions dans les *vues* d'une même *classe de base*, une *classe vue* est créée pour cette dernière. Elle contient le code, de toutes les fonctions des vues, relatif à l'état de l'objet. Le code d'une fonction définie dans plusieurs *vues* est une *combinaison* de chacune de ses définitions [Darg 97].

En résumé, pour chaque classe «X» pouvant avoir des *vues*, nous faisons correspondre une classe «_Comb_X» contenant toutes les définitions des méthodes de la classe et de ses *vues*. La définition de cette classe est basée sur une syntaxe proche de celle utilisant les directives du pré-processeur C++ *"ifdef"* et *"ifndef"*. Pour la fonction «printInfo()» dans «_Comb_Personne» nous avons la définition suivante :

```
void printInfo() {
// le code de la classe de base
_oid->printInfo();

// appel de la fonction de la vue si elle est active
if (active_view ("Emploie"))
   getview("Emploie")->printInfo();

if (active_view ("Étude"))
   getview("Étude")->printInfo();
}
```

C'est la définition simplifiée de la fonction «printInfo()» qui montre que chaque fois que cette fonction est appelée pour un objet de base, sa définition peut varier selon la liste des *vues* attachées et actives. Une intervention de l'usager est possible pour changer cette définition pour une autre selon ses besoins. L'avantage de cette possibilité réside dans le fait que l'on étudie la possibilité que chaque fonction puisse avoir une signification particulière dans son contexte, ce qui est respecté quand on a plusieurs *vues*.

Pour avoir accès à la nouvelle classe «_Comb_classe» que nous avons créée, une variable d'instance est ajoutée à la *classe de base* qui nous permet de faire les appels de délégation à la *classe de combinaison*. Cette instance est identifiée par «**_combView**»,

dont la classe est «_Comb_X», où « X» est la *classe de base* dans laquelle elle figure. Pour chaque appel à une fonction «f()», de la *classe de base* ou des *vues*, le pré-processeur le remplace par «**_combView**->f()». Dans notre exemple, l'appel à la fonction «printInfo()» sera remplacé par «**_combView**->printInfo()».

III.3. Aspects structurels

Dans notre approche de *vues*, ces dernières peuvent être toutes actives simultanément, comme elles peuvent ne pas l'être. Dans un cas extrême, on peut avoir un objet sans *vues* où seulement son comportement de base est présent. Nous distinguons entre la *programmation par vue unique* et la *programmation par vues multiples*.

Avec le premier type de programmation, un développeur peut utiliser les fonctionnalités disponibles dans juste une *vue*. Il n'est pas conscient de la programmation par *vues* ni de toute la mécanique derrière. Il ne voit que la définition de la classe *vue*. La création d'une instance de *vue* doit créer implicitement l'objet de base et connecter les deux sans que le développeur n'intervienne. Ce type de programmation n'a pas été implanté car il est considéré comme un cas particulier de la *programmation par vues multiples*.

Dans le cas de la *programmation par vues multiples*, le développeur est conscient de l'existence de plusieurs *vues*, et peut potentiellement les utiliser. Toutes les références à l'objet et à ses *vues* passent par l'objet de base, même si différents clients sont intéressés à différents sous-ensembles de *vues*.

La figure 3.7 illustre la structure d'un objet avec ses *vues* dans le cas de *programmation avec vues multiples*. Dans cette figure, l'objet de base a deux variables d'état et supporte deux opérations. Les *vues* de l'objet de base sont représentées par des objets particuliers ayant chacun ses propres caractéristiques. Ces objets «vues» peuvent contribuer à de nouvelles variables d'état («vues» 1 et 2), de nouvelles fonctions (toutes), ou bien simplement, rediriger les messages vers l'objet de base (tous). Ici, l'appel de l'opération «f()» sur la «*vue*1» est redirigé vers l'objet de base, et l'opération «f()» est

exécutée dans le contexte de l'objet de base. La même chose est vraie pour les références aux variables partagées (a pour «vue2», et b pour «vue3»).

L'objet de l'application est celui ayant le comportement défini par l'objet de base et les comportements des différents objets *vues*. Dans la figure 3.7, il est délimité par le rectangle pointillé.

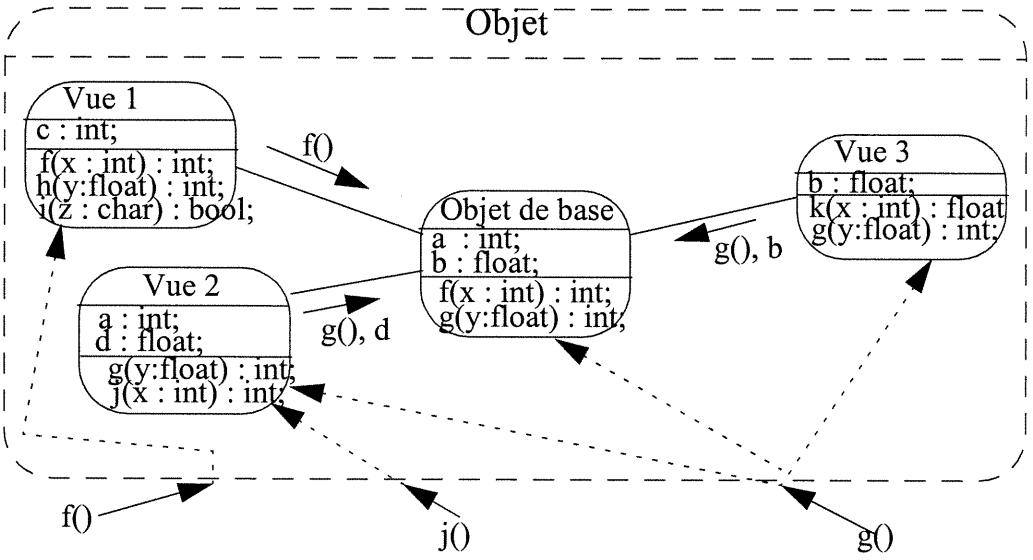


Figure 3.7 : Modèle d'objets avec vues

Notre approche au partage des variables correspond à ce que l'on appelle communément *délégation* [Mili 01]. Par contre, notre approche au partage de comportements (méthodes) est différente des approches traditionnelles à la délégation ou au prototypage, où l'opération dans l'objet, auquel on délègue, est exécutée dans le contexte du *délégateur* [Male 95]. Dans notre cas, nous avons un mécanisme pur et simple de redirection des messages. Ceci a plusieurs implications sur le modèle de programmation, lesquelles seront discutées dans le chapitre six. Conceptuellement, nous considérons que les quatre objets de la figure 3.7 représentent le même objet du domaine d'application. De nouvelles *vues* peuvent être attachées ou retirées à l'objet de base, lui ajoutant ou retirant des variables d'état et des fonctionnalités. Dans les faits, l'objet de base «pointera» vers les

vues actuellement attachées de manière à gérer l'ajout et le retrait de *vues* ainsi que la redirection des messages (*"dispatching"*).

III.4. Aspects comportementaux

III.4.1. Principes

L'ensemble des *vues* d'un objet change selon le besoin de l'application; des *vues* peuvent être ajoutées ou supprimées pour un même objet. Nous devons faire la distinction entre la durée de vie d'une *vue*, et la période durant laquelle elle est *active*, c'est-à-dire la période durant laquelle son comportement est disponible (pour l'objet). Nous distinguons donc entre création et attachement de *vues* d'une part, et activation de *vues* d'une autre part. De même, nous distinguons entre désactivation de *vues*, et détachement et destruction de *vues*. La création et l'attachement peuvent être considérés comme une opération atomique en raison du fait qu'une *vue* n'a aucune raison d'exister toute seule. La même chose est vraie pour les opérations de détachement et d'effacement.

Prenons l'exemple d'un objet de la classe «Personne» avec les *vues* «Employé» et «Étudiant» (section III.1.1.). Cet objet peut acquérir de nouvelles *vues*, notamment la *vue* «Directeur», tout comme il peut en perdre, par exemple, la vue «Étudiant» lorsque les études sont terminées. De même, la *vue* «Employé» peut ne pas être active, tout en existant, pour une certaine période de vie de l'objet «Personne». Dans ce cas, les propriétés de l'«Employé» ne sont pas disponibles pour l'objet.

Les comportements possibles d'un objet «Personne» sont le comportement de base seulement, le comportement d'une seule *vue* ou le comportement des deux *vues* attachées et actives simultanément. Quand une fonction est définie dans plusieurs *vues* simultanément, il faut prendre en considération ses diverses implantations et lui générer une nouvelle implantation qui prend en considération les différentes définitions. Cette nouvelle implantation est celle qui sera exécutée quand la fonction est appelée et quand les *vues* sont actives. Dans notre exemple, quand les *vues* «Employé» et «Étudiant» sont actives, il faut que la fonction «getcharge()» de l'«Employé» et de l'«Étudiant» soient prises en

considération. Le nouveau code de cette fonction peut être défini en se basant sur des règles de composition. Un exemple de règles de composition qui peut être adopté est celui utilisé dans la programmation *orientée-sujet* [Ossh 95].

En tout temps, il faut que l'objet de base ait une liaison avec ses *vues potentielles* et les instances de *vues attachées*, *actives* et *non actives*. Les *vues potentielles* sont les *vues* dont les instances peuvent être attachées à l'objet. Les instances des *vues attachées* sont celles créées pour l'objet en question, et les instances des *vues actives* sont celles *attachées* dont le comportement est disponible pour l'objet. Les instances des *vues non actives* sont les instances reliées à l'objet de base dont le comportement n'est pas disponible. Une instance de *vue* ne peut pas exister sans être explicitement reliée à un objet de base. Son lien avec ce dernier est initialisé lors de sa création et est maintenu le long de sa vie. Le lien entre l'objet de base et ses *vues potentielles* peut être établi lors de la création d'un objet de base et dès que l'on connaît l'ensemble des *vues* applicables à sa classe. Cependant, le lien entre l'objet de base et la liste des instances des *vues attachées* ou *actives* est mis à jour chaque fois qu'une instance d'une *vue* est attachée ou activée pour l'objet.

III.4.2. Exemple

Nous illustrons le comportement d'un objet avec plusieurs *vues* par un exemple de programmation par *vues* telle que nous l'avons implantée pour C++. Par cet exemple, nous aimerions montrer une utilisation des *vues*, mais aussi, soulever les questions qu'il faudra résoudre dans le cas d'un langage typé statique tel que C++.

Considérons l'exemple d'une entreprise de vente par catalogue qui gère une flotte de camions de livraison. La direction des «Finances» perçoit les camions comme étant un «capital amortissable» sur une période donnée. Pour la direction des «Opérations», qui gère les livraisons, les camions sont des «ressources quantifiables». Ces camions peuvent être considérés comme des appareils qui renferment des pièces de rechange, qui nécessitent un entretien régulier et une réparation occasionnelle. On appellera «Camion»

la classe de l'objet de base, contenant la marque, le modèle, le numéro de série, et quelques spécifications techniques telles que la charge et la date d'achat. Toutes ces données sont communes à tous les camions, peu importe sous quel angle ils sont vus. D'un autre côté, nous avons les *points de vue* « Finance», «Opération » et «Entretien» dont chacun spécifie d'autres données relatives à un camion. Un camion peut, à un moment ou à un autre, présenter une combinaison quelconque de ces *points de vue*.

Le *point de vue* «Finance» referme des données telles que «valeur d'achat», «valeur résiduelle» et «taux d'amortissement». Le *point de vue* «Opération» spécifie la «disponibilité», la «charge» et la «réservation». Le *point de vue* «Entretien» spécifie le «type de moteur», la «date de l'entretien» et la «charge». L'application des *points de vue* à la classe «Camion» permet de générer trois classes *vues* dont les instances seront reliées aux instances de la classe «Camion». Schématiquement, un objet «camion» peut être représenté sous forme d'un agrégat (objet de base et ses différentes instances de vues) (figure 3.8).

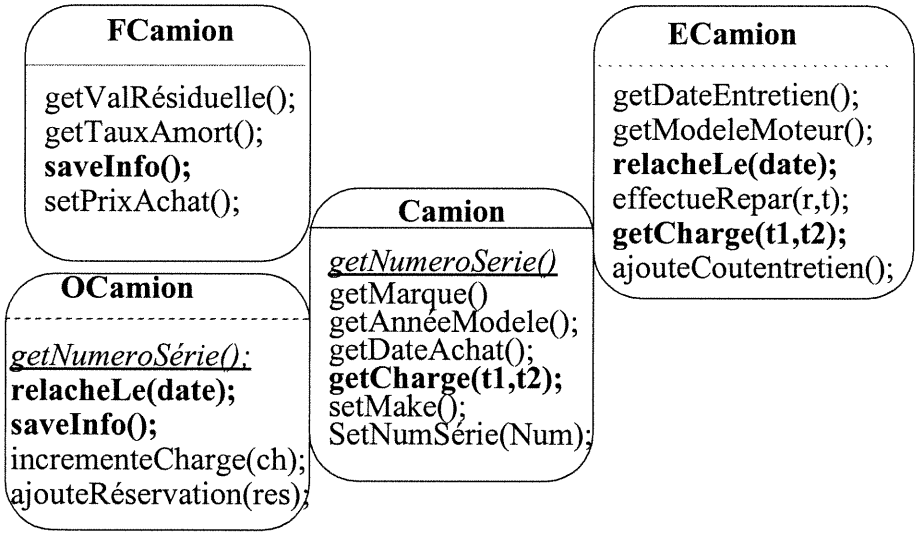


Figure 3.8 : Camion et ses vues

Pour illustrer le mécanisme de base de notre approche, nous représentons simplement l'interface de la *classe de base* «Camion» ainsi que l'interface de la *vue* «FCamion» obtenue par application du *point de vue* financier à la classe «Camion» (figure 3.9).

class Camion	class FCamion // vue Finance
{	{

TypeNSérie getNuméroSérie();
Marque getMarque();
Year getAnnéeModèle();
Date getDateAchat();
float getCharge();
void setNuméroSérie(TypeNSérie);
void setMake(MakeType);
	float getValeurRésiduelle();
	static float getTauxAmortissement();
	float getPrixAchat();
	void saveInfo();

}	}

Figure 3.9 : Classe de base avec sa vue générée

III.4.3. Comportement d'un objet avec vues

Dans ce qui suit nous montrons un exemple d'un code qui manipule des objets de la classe «Camion» avec les vues «FCamion» et «OCamion».

```
#include <camion.h> // La classe de base Camion
#include <fcamion.h> // La vue FCamion
#include <ocamion.h> // La vue OCamion
(a) Camion *monCamion; // création d'un objet de base
(b) cout<<monCamion->getNuméroSérie()<< endl;
(c) monCamion->attach("FCamion");// attacher la vue
(d) monCamion->getPrixAchat(); //fonction de la vue
(e) monCamion->attach("OCamion");// attacher la vue
(f) monCamion->saveInfo();//fonction dans les 2 vues
(g) monCamion->deactivate("OCamion");
(h) monCamion->saveInfo();//fonction de la vue FCamion
(i) monCamion->activate("OCamion");
(j) monCamion->detach("FCamion");
(k) delete monCamion;
```

Selon cet exemple, l'objet «monCamion» est passé par les étapes représentées dans la figure 3.10. Ces dernières seront détaillées dans ce qui suit.

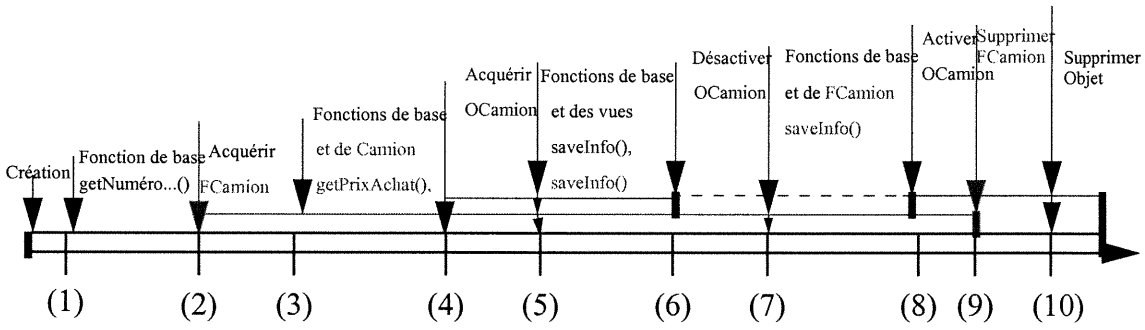


Figure 3.10 : Cycle de vie d'un objet avec vues

(1) Création d'un objet de base et accès à ses membres de base (lignes a et b),

(2) Ajout d'une *vue* (ligne c),

(3) Accès aux membres de base et de la *vue* attachée (ligne d),

(4) Attachement d'une nouvelle *vue* (ligne e),

(5) Accès aux différents membres selon les *vues* actives (ligne f),

(6) Désactivation d'une *vue* (ligne g),

(7) Accès aux membres de l'objet (ligne h),

(8) Activation d'une *vue* (ligne i),

(9) Détachement d'une *vue* (ligne j),et

(10) Suppression de l'objet de base (ligne k).

Les outils de support des *vues,* que nous avons développés, s'occupent de l'analyse et de la transformation du code usager en tenant compte des différentes *vues* actives. Nous supposons que le code des *vues* est déjà généré et qu'il est disponible pour l'usager avec le code de la *classe de base*. Les fonctions d'attachement, d'activation, de détachement et de désactivation sont accessibles grâce à la relation d'héritage entre la *classe de base* et la super-classe **Viewable_classe**. Cette dernière définit les attributs et fonctions communs à toutes les classes. Les outils de transformation de code s'occupent de rem-

placer, dans le code usager, les appels des fonctions qui figurent dans les *vues* par des appels de délégation vers la *vue de combinaison*. Cette dernière s'occupe d'assurer le bon code à exécuter selon les états des vues.

Dans ce qui suit, nous représentons les détails de l'analyse et de la transformation du code de l'exemple présenté ci-haut.

(1) Création d'un objet de base et accès aux membres de base (lignes a et b)

Les instances de la classe «Camion» peuvent être manipulées indépendamment des *vues*. La classe a sa propre interface commune à tous ses objets. Un utilisateur peut créer un objet «Camion» comme suit :

```
Camion* monCamion = new Camion();
```

Cet objet est manipulé comme dans le cas de la programmation conventionnelle (sans *vues*). Toutes ses méthodes peuvent être adressées, leur implantation est disponible et unique. Aucune transformation n'est faite pour la ligne b.

(2) Attachement de la vue FCamion (ligne c)

L'attachement (la création) des instances de la *vue* «FCamion» (ligne (c)) les lie à l'instance de la classe «Camion» qui invoque l'attachement.

Une *vue* est attachée à un objet quand elle lui est spécifiée (créée); elle est active quand ses attributs et fonctions sont disponibles. Pour supprimer une *vue*, il suffit de la détacher. Pour ne plus avoir accès à ses attributs et fonctions elle est désactivée. La liaison entre l'objet de base et l'instance de la *vue* est assurée par la mise à jour de la référence à cet objet de base et celle de la liste des instances des *vues* définie dans la classe de base. Pour attacher une *vue* à un objet de base, il suffit de l'ajouter dans sa liste des instances des *vues*; pour la détacher, elle sera supprimée de la liste.

Dans le cas de la programmation avec une seule *vue*, la *vue* est attachée lors de sa création et elle le demeure pour toute la durée de vie de l'objet. Quand plusieurs *vues* sont applicables à une même *classe de base*, l'utilisation des fonctions d'attachement et de détachement nous permet de les manipuler d'une façon dynamique.

(3) Accès aux membres de la vue (ligne d)

La ligne (d) consiste à faire l'appel de la fonction «getPrixAchat()» qui est définie dans la vue «FCamion». Cette ligne sera transformée en:

```
(d')monCamion->_combView->getPrixAchat();
```

La classe de la variable d'instance **_combView** contient la définition des fonctions des *vues*. Pour cette ligne c'est le code défini dans la vue «FCamion» qui sera exécuté.

(4) Attachement d'une nouvelle vue (ligne e)

Reprenons l'exemple de la figure 3.8, les interfaces des *points de vue* «Opération» et «Entretien» sont présentées dans la figure 3.11. Les *vues* dérivées de ces *points de vue* et de la classe «Camion» peuvent être créées simultanément avec la vue «FCamion».

Les trois *vues* peuvent être attachées en même temps aux objets de la classe «Camion». Cela est assuré par la fonction d'attachement que notre pré-processeur définit et qui initialise l'objet de base dans la *vue* et la liste des *vues* dans l'objet de base.

```
class OCamion{
OCamion(Camion *);
TypeNSérie getNuméroSérie();
void relacheLe(Date);
void saveInfo();
void incrementeCharge(float ch);
void ajouteRéservation
        (Réservation res);
}
// La vue opération
```

```
class ECamion{
ECamion(Camion *);
Date getDateEntretien();
void relacheLe(Date);
void effectueRépar
          (Réparation r,
          Date t);
float getCharge(Date t1,Date t2);
void ajouteCoutEntretien(float);
}
// La vue entretien
```

Figure 3.11 : Les vues OCamion et ECamion

Comme dans le cas de l'attachement de la vue «FCamion», la *vue* «OCamion» est attachée à l'objet «monCamion». Il faut que l'implantation des *vues* utilisées, «FCamion» et «OCamion», soit disponible pour le programme. La directive *"#include"* est utilisée pour importer ces implantations (voir le code).

Au moment de l'exécution, la *classe de base* ainsi que les différentes *vues* sont disponibles pour l'usager qui peut attacher une *vue* à une classe, la détacher ou l'activer et la désactiver. Cela implique la co-existence des *classes de base* et des *vues* en même temps au moment de l'exécution. L'instance de la *classe de base* peut exister indépendamment des *vues*. Nous avons défini explicitement les constructeurs des *vues*. Dans le code ci-haut, l'attachement de la *vue* «OCamion» est fait d'une façon explicite en utilisant la fonction «attach(char*)». Cette fonction crée une instance de la *vue* et l'attache à l'objet de base (qui reçoit le message). À partir de la ligne (e), les comportements de base et des deux *vues* sont disponibles pour l'objet de base.

(5) Accès aux différents membres, de base et des vues (ligne f)

Suite au code de la ligne e, nous pouvons invoquer les membres de la *classe de base* ainsi que ceux des *vues* «FCamion» et «OCamion». L'appel de ces membres ne génèrera pas un message d'erreur. Cependant, si un membre de la *vue* «ECamion», non encore attachée, est appelé, un message d'erreur sera généré. Ainsi, l'appel «monCamion->getDateEntretien()» génèrera un message d'erreur.

À la ligne (f), la méthode «saveInfo()», définie dans deux vues actives, est appelée. Dans ce cas, notre pré-processeur doit prendre en considération les implantations des deux vues «FCamion» et «OCamion» et remplacer l'appel. Donc la ligne (f) peut être remplacée par :

```
(f') monCamion->_comView->saveInfo();
```

Cet appel consiste à déléguer à la *vue de combinaison* la fonction «saveInfo()». Cette vue prendra en considération les deux implantations de la fonction en les invoquant toutes les deux.

(6), (7) Désactivation d'une vue et accès aux membres (lignes g, h)

À la ligne (g), nous invoquons la méthode «deactivate(char* unevue)» sur l'objet de base. Cette méthode est définie par le pré-processeur pour la *classe de base*. Elle consiste à rendre les membres de la *vue* non disponibles pour l'objet de base. La *vue* reste

attachée et c'est seulement son comportement qui est désactivé. Ainsi, la ligne (h) qui appelle la méthode « saveInfo() » sera remplacée par:

```
(h') monCamion->_comView->saveInfo();
```

qui consiste à exécuter le code défini dans «FCamion» seulement.

(8) Activation d'une vue (ligne i)

La fonction «activate(char* unevue)» rend les membres d'une *vue* actifs. Elle est définie par notre pré-processeur pour l'ensemble des *classes de base* et son implantation consiste à vérifier si la *vue* appelée «unevue» est déjà attachée et, dans ce cas, la rendre active. À partir de cette activation nous nous retrouvons dans un état semblable à celui avant la désactivation.

(9) Détachement d'une vue (ligne j)

Le détachement d'une *vue* est l'équivalent de sa suppression. L'instance de la *vue* «FCamion» n'existe plus après un détachement et son comportement n'est plus disponible. La fonction «detach(char* unevue)» est définie par le pré-processeur pour toutes les *classes de base* et consiste à supprimer l'instance de la *vue* appelée «unevue» de la liste des *vues* de l'objet de base.

(10) Suppression de l'objet (ligne k)

La ligne (k) met fin à la durée de vie de l'objet «monCamion». À partir de cette ligne, l'objet, ainsi que sa *vue* «OCamion», n'existent plus.

III.5. Conclusion

Dans ce chapitre nous avons présenté notre approche de programmation par *vues*. Nous avons traité de ses aspects structurels et comportementaux et donné des exemples d'utilisation. Nous avons identifié trois structures abstraites différentes : les *classes de base*, les *vues* et les *points de vue*.

Les classes représentent les objets physiques, les *vues* représentent les *rôles* de ces objets et les *points de vue* représentent des aspects fonctionnels particuliers.

À un moment donné, une instance d'une classe de base peut avoir plusieurs instances de *vues* qui lui sont reliées. Ces instances peuvent être attachées, détachées, activées ou désactivées dynamiquement selon les besoins du développeur.

Chapitre IV

Modèle formel

Ce chapitre présente l'aspect formel de notre approche pour la programmation par *vues*. Nous y parlerons tout d'abord des mécanismes qui ont motivé la notion de *points de vue* qui, basée sur la paramétrisation, constitue un important volet de la recherche qui a été menée. Nous y présenterons ensuite les structures formelles qui sont à la base de la définition théorique des nouvelles structures (*vues, points de vue* et *classes de base*), que nous avons ajoutées aux langages de programmation pour supporter la programmation par *vues*. Pour conclure ce chapitre, nous parlerons des relations d'héritage ou de spécialisation qui peuvent exister entre les différentes constructions sous-jacentes à notre approche. Dans la section IV.1, nous parlerons des mécanismes de paramétrisation dans les langages *orientés-objet*. Le cadre formel pour les *points de vue* et les *vues* est présenté dans la section IV.2. Le modèle formel de *points de vue* est présenté dans la section IV.3. La section IV.4 traite la spécification et la génération de *vues*. Nous étudions la relation de spécialisation entre *points de vues* et *vues* dans la section IV.5.

IV.1. Mécanisme de paramétrisation

La paramétrisation et la généricité sont déjà supportées par plusieurs langages de programmation. Par exemple, Ada nous offre les *Packages* génériques, C++ les *patrons* de classes [Stro 97] et Eiffel les classes *génériques* [Meye 88].

Ces mécanismes permettent de définir des concepts (classes, packages, structures de données, etc.) généraux qui seront instanciés pour des types spécifiques. Par exemple, nous pouvons définir une classe générique «Pile» pour définir un ensemble d'objets de types divers tels que des objets entiers, des caractères ou des types dérivés (figure4.1). Le type générique «T», utilisé dans la définition, peut être substitué par n'importe quel type spécifique «t», et la classe Pile<t> devient une pile d'éléments du type spécifique «t» (figure 4.2).

```
template <class T>
class Pile {
public:
    void empiler(T elem){tab[++tete] = elem;}
    T depiler() {return tab[tete--];}
    int estvide(){return tete == -1;}
    int estpleine(){return tete ==taille -1;}
    Pile(int s = 100):taille(s),tete(-1)
                      {tab = new T[taille];}
    ~Pile() {delete [] tab;}

private:
    T* tab;
    int tete;
    int taille;

};
```

Figure 4.1 : Patron de classe de C++

```
main()
{
 Pile<int> intPile1(5);// Une pile de 5 entiers
 Pile<char> charPile(8); // Une pile de 8 caractères
 intPile1.empiler(77); // On manipule des entiers
 charPile.empiler('A');//On manipule des caractères
 cout << intPile1.depiler() <<endl;
...
}
```

Figure 4.2 : Instanciation de la classe Pile

Cette paramétrisation nous permet de ne définir la classe «Pile» qu'une seule fois pour divers types. Ainsi, la définition peut être utilisée dans un éventail de situations. Cela signifie que les objets adoptent leur type au moment de l'instanciation, et la vérification des types est retardée pour n'être effectuée qu'au moment de l'édition de lien ou de la compilation.

Lors de la définition d'une classe générique, les types utilisés par celle-ci sont ceux ayant des comportements identiques les uns aux autres. Ce sont les paramètres des types représentés par des noms formels qui seront substitués à des types spécifiques au moment de l'instanciation de la classe. Les types substitués peuvent être

soumis à certaines contraintes. Par exemple, il est possible, en Ada et en Eiffel, de spécifier des exigences sur les types génériques en spécifiant certaines de leurs caractéristiques. En C++, les vérifications sur les types applicables qui peuvent être substitués sont effectuées au moment de la compilation. Dans [Myer 97], les auteurs présentent une extension de cette paramétrisation dans Java pour spécifier des exigences particulières sur les types.

Nous aimerions introduire, dans le cadre de la définition que nous faisons des *points de vue*, les exigences sur les classes pour lesquelles un *point de vue* est applicable, et ajouter à ces classes de nouveaux attributs et fonctions. Ainsi, notre définition des *points de vue* consiste à spécifier un paramètre qui est une classe et à définir, dans la partie **requires** de la définition (figure 4.3), un ensemble d'exigences sur ce paramètre. L'ajout de nouveaux attributs et fonctions pour la création de la vue se fait dans la partie **provides.**

Le *point de vue*, défini dans la figure 4.3, peut être appliqué à toute classe «C» ayant une variable de type «T» et une fonction de type «T2» avec un paramètre de type «T3». Il fournit à la classe «C» une nouvelle fonction «g()».

```
viewpoint Un_pointdevue <class C> {
requires:
   T var1;
   T2 f(T3 par1);
provides:
   T g() {return var1;}
};
```

Figure 4.3 : Exemple de point de vue

L'instanciation du *point de vue* pour un paramètre spécifique, soit une classe donnée, génère une nouvelle classe (un nouveau type statique) appelée *vue*. Cette *vue* regroupe des caractéristiques de la classe (de base) et des caractéristiques du *point de vue*.

Reprenons l'exemple du *point de vue* «Financier» et de la classe «Camion» présenté au paragraphe III.4. du chapitre trois. La fonction «getValeurResiduelle()», définie dans le *point de vue* «Financier», peut être offerte par un objet de type «bien capital». Un «ordinateur», tout comme un «camion», peut être considéré comme un objet de ce type. Ainsi, d'un *point de vue* financier, les deux objets, «camion» et «ordinateur», doivent offrir cette fonction. L'idée étant que si nous pouvons définir les caractéristiques des «biens capitaux», nous pourrons les voir sous le même *point de vue* «Financier». Cela nous permet de définir une seule fois les fonctions de ce *point de vue* et de les appliquer aux différentes classes.

Notons finalement que l'ajout du concept de *point de vue* est fait sur la base du polymorphisme dynamique [Myer 97], qui permet l'implantation et la définition des abstractions génériques. Le polymorphisme fait en sorte qu'un objet d'un programme fasse référence, au moment de l'exécution, à des objets appartenant à plus d'un type ou classe [Oder 97]. Il consiste à offrir la possibilité de décrire des données ou des algorithmes dont la structure est indépendante du type des éléments manipulés. L'objectif est de regrouper des objets ayant un ensemble de caractéristiques communes tout en appartenant à des classes différentes.

IV.2. Définitions

Nous utilisons un style de spécifications algébriques pour décrire les classes (que nous avons appelées *classes de base*), les *vues* et les *points de vues*. Dans ce qui suit, nous présentons certaines de ces spécifications algébriques qui sont utilisées pour nos définitions.

IV.2.1. Les TYPES

Un *TYPE* est un ensemble de *types* plus élémentaires, de *variables* et d'*opérations*. Une *opération* est décrite par sa signature et ses pré-conditions et post-conditions. Une classe, qui regroupe un ensemble de variables et d'opérations, peut être représentée sous forme d'un *TYPE*. Par exemple, la classe «Camion» peut être

définie comme suit :

```
TYPE Camion
types
   TypeNSérie, TypeId, TypeMarque, Year, Date, float;
variables
   numéroSérie  : TypeNSérie;
   marque  : TypeMarque;
   anModèle  : Year;
   id  : TypeId;
operations
   getNuméroSérie(t   : Camion)  :TypeNSérie
           pré-conditions
           post-conditions
   setNumeroSérie (t  : Camion, s   : TypeNSérie) :Camion
           pré-conditions
           post-conditions
           ...
endtype
```

Formellement, nous représentons un *TYPE* par un triplet <*types, vars, opns*>, où *types* est un sous-ensemble de l'ensemble des types *TYPES*, *vars* est un sous-ensemble du produit cartésien *NOMS* × *TYPES*, et *opns* est un ensemble de triplets <σ, *P, Q*>, où σ est la signature de la procédure et *P* et *Q* ses pré-conditions et post-conditions respectivement. Une signature σ peut être vue comme une n-uplet de la forme <*type*> (<$type_1$>,...,<$type_n$>), où le premier *type* représente le type de retour de la fonction et les subséquents, les paramètres. *P* et *Q* sont des expressions booléennes des *variables* et des *opérations*. Nous faisons la distinction entre les *sortes* qui sont des symboles dénués de sens sémantique, les *types* qui correspondent à des *algèbres* et les *théories* qui sont une sorte d'*exigences* sur les *sortes* qui peuvent être satisfaites par des *types*.

IV.2.2. Théories

Nous décrivons les *théories* dans le même style que les *TYPES*, avec deux différences :

Tout d'abord, les «composantes» d'une *théorie* sont des *sortes* au lieu d'être des *types*. Deuxièmement, les *opérations* sont définies par leurs *signatures* et par l'ensemble d'*axiomes* qu'elles satisfont. Un exemple d'une théorie est donnée dans ce qui suit :

théorie <nom de théorie>
  sorts
    $<sort_1>,...,<sort_n>$
  variables
    $<var_1>$: $<sort_{i1}>,...,<var_k>$: $<sort_{ik}>$;
  opérations
    $<opération_i>$ ($<var_1>$: $<sort_{i1}>,...,$
      $<var_k>$: $<sort_{ik}>$) : $<sort_{i0}>$;
  axioms
    $<opération_i>$ {instructions;}
    ...
endthéorie

IV.2.3. Relations entre les TYPES et les théories

Nous avons utilisé les *TYPES* pour représenter les *classes* et les *vues,* et les *théories* pour représenter les *points de vue*. Nous avons aussi défini un ensemble de relations entre les *TYPES* eux-mêmes d'un côté et les *TYPES* et les *théories* d'un autre côté :

- La relation de sous-typage entre les *TYPES* nous permet de relier deux *TYPES* qui se correspondent, un général et un autre plus spécifique.

- La relation de satisfaction nous permet de relier un *TYPE* à une *théorie* pour vérifier si la *théorie* peut être appliquer à ce *TYPE.*

- La relation de *"subsumption"* relie deux *théories* de la même façon que la relation de sous-typage le fait entre les *TYPES.*

IV.2.3.1. Sous-typage

Soient T_1 = <$types_1$, $vars_1$, $opns_1$> et

T_2 = <$types_2$, $vars_2$, $opns_2$> deux types.

Nous dirons que T_1 est un sous-type de T_2, ou $T_1 \subseteq T_2$ s'il existe deux correspondances Σvars et Σopns telles que:

1. Pour tout v2 = <nom2, type2> ∈ vars2, la variable v1 = <Σvars(nom2), type2> ∈vars1
2. Pour tout $op_2 = <\sigma_2, P_2, Q_2>$ où $\sigma_2 = (<n_{2,1}, t_{2,1}>, ..., <n_{2,k}, t_{2,k}>) \in opns_2$, il existe une opération $op_1 = <\sigma_1, P_1, Q_1>$ où $\sigma_1 = (<\Sigma opns(n_{2,1}), t_{1,1}>, ..., <n_{2,k}, t_{1,k}>) \in opns_1$ telle que:

   - $t_{1,1} \subseteq t_{2,1}$, et pour tout i, $t_{2,i} \subseteq t_{1,i}$ (contravariance) et,
   - P_1 -> P_2 ($\Sigma vars_{-1}(vars_2)$,$\Sigma opns_{-1}(opns_2)$) et
     Q_2 -> Q_1 ($\Sigma vars_1(vars_2)$,$\Sigma opns_{-1}(opns_2)$)

L'expression P_2 ($\Sigma vars_{-1}(vars_2)$,$\Sigma opns_{-1}(opns_2)$) signifie le prédicat P_2, où nous avons remplacé les variables et opérations de T_2 par les variables et opérations de T_1 auxquelles elles correspondent. Cette définition est semblable à celle du sous-typage dans le langage Eiffel où l'on permet le renommage des variables et des opérations [Meye 88].

Généralement parlant, nous écrirons $T_1 \subseteq_\Sigma T_2$, *où* Σ *représente la paire* <Σvars,Σopns>, ou bien $T_1 \subseteq T_2$, lorsqu'il n'y a pas de renommage, i.e. Σ = identité.

D'une manière informelle, cela signifie que pour chaque variable du type T_2 nous avons une variable correspondante de même type dans T_1, et pour chaque opération de T_2, une variable correspondante dans T_1, qui est plus générale.

IV.2.3.2. Satisfaction

Un *TYPE* T_1 = <$types_1$,$vars_1$,$opns_1$> satisfait une *théorie* TH_2 = <$sorts_2$, $vars_2$, $opns_2$, $axioms_2$>, si et seulement si, il existe trois correspondances Σ_{types},Σ_{vars}, et Σ_{opns} telles que:

1. Σ_{types} définit une bijection de $types_1$ à $sorts_2$,

2. Pour tout $v_2 = <nom_2, sort_2> \in vars_2$, la variable $v_1 = <\Sigma_{vars}(nom_2),\Sigma_{types}(sort2)> \in vars_1$,

3. Pour tout $op_2 = \sigma_2$ où $\sigma_2 = (<n_{2,1},s_{2,1}>,...,<n_{2,k},s_{2,k}>) \in opns_2$, il existe une opération $op_1 = <\sigma_1,P_1,Q_1>$ où:

- $\sigma_1 = (<\Sigma_{opns}(n_{2,1}),\Sigma_{types}(t_{2,1})>,...,<n_{1,k},\Sigma_{types}(t_{2,k})>) \in opns_1$, et
- $[P_1 \text{ -> } Q_1] \Rightarrow [axiomes_2 (\Sigma_{types} (sortes_2),\Sigma_{vars}(vars_2),\Sigma_{opns}(opns_2)]$

Cette dernière implication est un raccourci (abusif) pour dire que le comportement des *opérations* du *type*, tel que décrit par leurs pré-conditions et post-conditions, est consistent avec les *axiomes* mais qu'il peut être plus *défini* que ce que n'exige les *axiomes*. Nous définissons la relation de satisfaction entre *TYPE* et *théorie* avec la notation $T \Rightarrow_\Sigma TH$ pour dire que T satisfait la théorie TH sous la correspondance $\Sigma = <\Sigma_{types},\Sigma_{vars}, \Sigma_{opns}>$.

IV.2.3.3. "Subsumption"

La relation de *"subsumption"* est définie entre les *théories* de façon analogue à la relation de sous-typage entre les *TYPES*, à une correspondance $\Sigma = <\Sigma_{sorts},\Sigma_{vars}, \Sigma_{opns}>$ près. Nous écrivons $TH_2 \Rightarrow_\Sigma TH_1$ pour dire que TH_2 *"subsume"* (décrit une sous-classe de) TH_1.

IV.2.3.4. Propriétés des relations

Propriété 1 : *transitivité du sous-typage*

Soient T_1, T_2, T_3 trois *TYPES* tels que $T_1 \subseteq_\Sigma T_2$ et $T_2 \subseteq_{\Sigma'} T_3$ alors $T_1 \subseteq_{\Sigma' \circ \Sigma} T_3$

Propriété 2 : *satisfaction et sous-typage.*

Soient T_1 et T_2 deux *TYPES* tels que

$T_1 \subseteq_\Sigma T_2$ et TH une *Théorie* telle que $T_2 \Rightarrow_{\Sigma'} T_3$
alors $T_1 \Rightarrow_{\Sigma' \circ \Sigma} T_3$

Propriété 3 : *satisfaction et subsumption.*

Soient T_1 un *TYPE* et TH_1 et TH_2 deux *théories* telles que

$T_1 =>_{\Sigma} TH_1$ et $TH_1 =>_{\Sigma'} TH_2$

alors $T_1 =>_{\Sigma' o \Sigma} TH_2$

A partir d'une *théorie* que nous pouvons considérer comme étant un type entièrement paramétré par les noms des fonctions et les types, nous pouvons définir toute une hiérarchie de *théories* obtenues en fixant l'un des paramètres. Prenons l'exemple que voici :

Dictionary<T,V> => Dictionary<String,V> =>Dictionary<String,Person>

Nous partons du cas où T et V sont des paramètres vers le cas où seul V demeure un paramètre, mais où T a été fixé à «String». Par la suite, nous fixons V à «Person» pour obtenir la classe concrète «Dictionary<String,Person>» .

Supposons que TH soit une *théorie* avec le paramètre de *type* (une *sorte*) «s_i». Si nous fixons «s_i» à un type «t_i», nous décrivions la *théorie* résultante comme suit : $TH' = TH[s_i -> t_i]$, et ceci peut se généraliser à plusieurs paramètres.

Pour tout sous-ensemble «B» de «A», nous définissons Σ_{A-B} comme étant la correspondance qui est identique à Σ_A sur «A-B» et qui est l'identité sur «B». Nous avons:

$T =>\Sigma' TH[(s-> \Sigma types(s)) s \varepsilon S]$, où $\Sigma' = <\Sigma types\text{-}S, \Sigma vars, \Sigma opns>$

La même chose est vraie pour les variables et les opérations.

IV.3. Modèle formel

Notre modèle de *points de vues* utilise des *paramètres de types* ou *théories*, qui sont décrits en termes de propriétés qui doivent être satisfaites par les paramètres actuels.

Formellement, un *point de vue* peut être considéré comme un *TYPE* paramétré par une *théorie*, que l'on écrit VP[TH], où TH est une théorie qui décrit le type des objets auxquels le *point de vue* peut être appliqué. Nous illustrons la syntaxe par

l'exemple de la figure 4.5.

```
théorie ThéorieCapital
  sorts TypeAge, TypePrixAchat, void, ... ;
  variables
          dateAchat  : TypeAge, prix  :
      TypePrixAchat...;
  opération
           setDateAchat(age:TypeAge)  : void;
           ...
  axioms
           setDateAchat {... ;}
           ...
endthéorie
```

Figure 4.4 : Exemple de Théorie

```
viewpoint CapitalFinancier
  requires ThéorieCapital [TypeAge -> Year,
                           TypePrixAchat -> TypeMonnaie]
  provides
      variables
      Year _périodeAmortissement;
      TypeMonnaie _valeurRésiduelle;
      opérations
      TypeMonnaie getValeurRésiduelle ()
           {return _valeurRésiduelle;}
      void setPériodeAmortissement(Year y)
           {_périodeAmortissement = y;}
                ...
end viewpoint
```

Figure 4.5 : Exemple de point de vue

Les exigences relatives aux classes d'objets (les paramètres génériques) auxquelles nous pouvons appliquer le *point de vue* sont décrites dans la partie **requires**. Ici, nous donnons le nom d'une *théorie* préalablement définie («ThéorieCapital»), figure 4.4) que nous avons partiellement spécifiée ou «déparamétrisée», puisque

nous avons fixé son paramètre (une *sorte*) «TypeAge» au type (classe) actuel «Year».

Nous avons aussi substitué le symbole «TypePrixAchat» (un nom de *sorte* dans la *théorie*) par le symbole «TypeMonnaie», mais sans lui affecter un *type* particulier; d'autre part, une *théorie* peut être utilisée dans plusieurs *point de vues*. Dans notre exemple la classe à laquelle le *point de vue* peut être appliqué doit satisfaire la théorie «ThéorieCapital». Toute occurrence de «TypeAge» utilisée dans la définition de la *vue* sera remplacée par le type «Year», et les occurrences de «TypePrixAchat» le seront par «TypeMonnaie» ou par un de ses sous-types.

De plus, les types paramétriques définis dans la partie **requires** sont utilisés dans la partie **provides**, et c'est le générateur de *vues* qui effectue les vérifications et les transformations de code de la partie **provides** en faisant, au besoin, des appels de délégation vers la partie **requires** ou ses correspondants de la classe de base. L'interface des *vues* générées à partir des *points de vue* est limitée à la clause **provides**.

Si on veut exporter une méthode ou une variable de la *classe de base* dans la *vue*, il faut la spécifier dans provides et montrer, dans la partie requires, la variable (ou méthode) à laquelle elle correspond. D'autres variantes syntaxiques pour la spécification de *points de vue* ont été prévues, dont les spécifications et extensions de théories "*in-line*" (figure 4.6). Dans ce cas au lieu de spécifier une *théorie* déjà existante comme «ThéorieCapital», nous définissons nos exigences sur les classes en énumérant les membres que la *classe-théorie* doit contenir. Ceci correspond à la phase d'écriture du code de la *théorie* au lieu de la passer en paramètre. La correspondance entre les membres de la *théorie* et les membres de la classe sera faite sur la base des *types* en tenant compte de la relation de sous-typage entre ces derniers.

```
viewpoint CapitalFinancier
  requires
      variables
           TypeAge          dateAchat;
           TypePrixAchat    prix;
      opérations
           void             setDateAchat (TypeAge);

  provides
          variables
            ...
          opérations
            ...

end viewpoint
```

Figure 4.6 : Exemple de point de vue avec paramètre *in-line*

Dans l'exemple de la figure 4.6, nous avons défini explicitement (en mode *"in-line"*) les exigences sur les classes auxquelles le *point de vue* peut être appliqué. Ces classes doivent avoir une variable d'un type (ou sous-type) qui satisfait le type «TypeAge» et une autre variable qui satisfait le type «TypePrixAchat». Elles doivent aussi avoir une méthode de type «void» avec un paramètre de type «TypeAge». Les variables et les méthodes spécifiées dans la partie **requires** sont utilisées dans la partie **provides**. Lors de la définition de la *vue,* il suffit de remplacer ces variables et ces méthodes par leurs correspondants dans la classe.

IV.4. Spécification et génération des vues

Une *vue* est générée par application d'un *point de vue* VP[Th] à un *TYPE* (une classe) «T» qui satisfait la théorie «Th» pour une correspondance Σ, c'est-à-dire T=>ΣTh. Nous avons alors : V = T **as** VP [$(s_i \text{ ->}\Sigma(s_i))_i$, $(v_j \text{ ->}\Sigma(v_j))_j$, $(op_k \text{ ->}\Sigma(op_k))_k$]

Les substitutions mises à l'oeuvre pour générer les *vues* à partir des *points de vues* touchent aussi bien les noms des *types*, que les noms des variables et des fonc-

tions. La génération des *vues* doit transformer le code source du *point de vue* pour rediriger les messages et les références aux variables et fonctions, appartenant à la *classe de base* (contenues dans la partie **requires**), vers l'objet de base correspondant. Par exemple, si une méthode «f» de **provides** fait appel à la méthode «getDateAchat()» de **requires**, alors le code:

```
provides:
     void f(...)
          {
          ...
          Date d= getDateAchat();
          ...
          }
```

sera transformé en:

```
     void f(...)
          {
          Date d = _camion->getDateAchat(..);
          }
```

où « _camion » est une variable du type «Camion*» ajoutée automatiquement à la *vue*. Les références aux variables sont traitées de façon similaire. Dans les deux cas, il faut s'assurer que les variables et méthodes citées dans **requires** ont des correspondants spécifiques dans la *classe de base*. Les membres de la *vue* sont construits principalement à partir des membres **provides** du *point de vue,* et ceux-là qualifient le domaine du *point de vue* et caractérisent les objets de la *vue* résultante par rapport aux objets de la *classe de base*. Les membres dans **provides** peuvent utiliser des membres de la *classe de base*, ceux-là sont référés par délégation en se servant de l'identificateur de l'objet de base présent dans la *vue*.

IV.5. Relation d'héritage

Parmi les principaux avantages de l'introduction des *points de vue* représentant des aspects fonctionnels, se trouvent la réutilisation et le développement décentralisé. Ces aspects fonctionnels, développés indépendamment les uns des autres peuvent être appliqués à un vaste ensemble de classes. Ils ont une forme particulière de *classes génériques* et en héritent quelques caractéristiques ne serait ce que d'un point de vue concept de base. Les *vues* sont de pures classes définies selon la syntaxe particulière du langage de programmation et comportent toutes les caractéristiques communes des classes. Une de ces caractéristiques est la possibilité de dériver une classe à partir d'une autre classe qui existe déjà grâce à la relation d'héritage. Or, la définition des *vues* est faite à partir des *points de vue*, et si une relation peut exister entre deux *vues* cela veut dire que cette relation est liée aux parties génératrices, *les points de vue* dans notre cas. La possibilité de faire appel à une relation d'héritage entre les *vues* nous a poussés à étudier la possibilité de se prévaloir de cette relation entre les *points de vue*.

L'héritage, sous toutes ses formes (spécialisation, sous-typage, et classification) intervient à trois différents niveaux dans la programmation par *vues* :

1. la spécialisation-subsumption des *points de vue,* comme mécanisme de spécification incrémentale et de réutilisation,
2. le sous-typage, et plus généralement la «substituabilité» comportementale, de *vues* générées de *points de vues* hiérarchiquement liés, ou de *vues* générées pour des *classes de base* hiérarchiquement liées, et
3. le sous-typage dynamique, et plus généralement la «substituabilité» comportementale d'objets de base hiérarchiquement liés auxquels on a rattaché des *vues*, qui peuvent être dérivées de *points de vue* hiérarchiquement liés.

IV.5.1. Spécialisation de points de vues

Comme dans le cas de la spécialisation des classes, il est parfois intéressant de dériver des *points de vue* à partir de ceux déjà existant. Prenons l'exemple du *point de vue* «Étude» défini dans la figure 4.7. Ce *point de vue* fournit un comportement, relié aux études, pour les classes auxquelles il est appliqué. Dans ce même contexte des études, nous pouvons imaginer un *point de vue* «ÉtudesUniversitaires» qui, à part le comportement relié aux études déjà offert par le *point de vue* «Étude», offre le comportement lié à l'université en particulier. Notons par exemple les différents projets de recherche offerts, les subventions et les séminaires (figure 4.8). La dérivation de ce nouveau *point de vue* à partir de celui qui existe déjà nous fait sauver la répétition de la partie déjà spécifiée.

```
viewpoint Étude{
requires:
    char * identificateur;
    .....
provides:
    char * e_adresse;
    char* cours;
    char * getAdresse() {return e_adresse}
    void printInfo()
    {cout<< identificateur << e_adresse << cours ;}
    ...
};
```

Figure 4.7 : Point de vue Étude

Formellement, nous définissons une relation entre *points de vue* qui correspond à la spécialisation et à l'extension. Nous l'appellerons **spec**. Intuitivement, un *point de vue* «VP_1» **"spec"**ialise un *point de vue* «VP_2» si «VP_1» est applicable au moins aux mêmes *TYPES* (classes) que «VP_2» et s'il offre une extension ou spécialisation des fonctionnalités offertes par «VP_2» pour ces types.

```
viewpoint ÉtudeUniversitaire {
requires:
    char * identificateur;
    char * diplôme;
    .....
provides:
    char * e_adresse;
    char* cours;
    char * recherche;
    char * getAdresse() {return e_adresse}
    void printInfo()
    {cout<< identificateur << e_adresse << cours ;}
    char * listsubvention() {...}
    ...
};
```

Figure 4.8 : Point de vue ÉtudeUniversitaire

Soient VP_1 [Th_1] et VP_2[Th_2] deux *points de vues*. On définit la relation VP_1 [Th_1] **spec** VP_2[Th_2]

comme suit:

- Il existe une correspondance Σ telle que $Th_1 \Rightarrow_\Sigma Th_2$
- Pour tout type T et toute correspondance Σ' telle que $T \Rightarrow_{\Sigma'} Th_1$, il existe une correspondance Σ'' telle que

  provides($VP_1[Th_1 \rightarrow \Sigma' T]$) $\subseteq_{\Sigma''}$ **provides**($VP_2[Th_2 \rightarrow \Sigma' o \Sigma_T]$)

La notation **provides**(VP[Th]) est utilisée pour référer au type-interface défini par la clause **provides** du *point de vue*. Ainsi, la relation de spécialisation entre les *points de vue* implique des relations (de satisfaction et de sous-typage) entre les parties **requires** et **provides** pour des classes spécifiques (figure 4.9). Notre modèle permet la "**spec**"ialisation multiple de *points de vues* et la *"subsumption"* multiple de *théories*, pour les mêmes raisons que nous permettons l'héritage multiple en général.

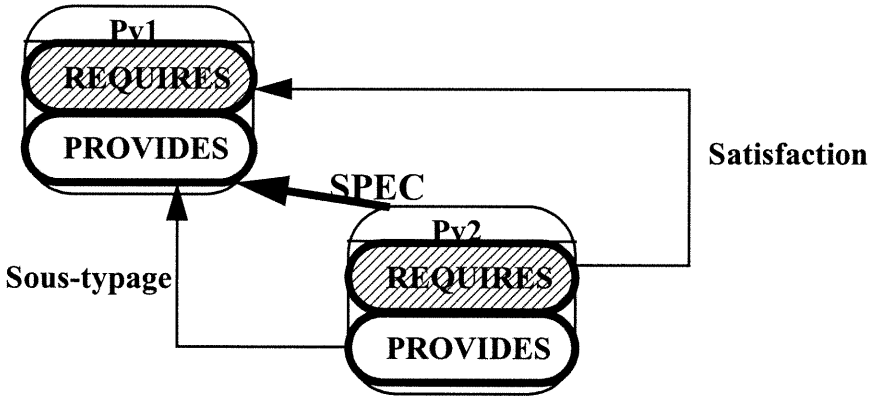


Figure 4.9 : Spécialisation de points de vue

IV.5.2. Spécialisation de vues

Prenons l'exemple où VP[Th] est un *point de vue* et «T» est un type satisfaisant la théorie «Th», par exemple, $T =>\Sigma_{Th}$. Si «V» est la vue V = T **as** VP [Th ->Σ_T] et «T'», un sous type de «T», où $T' \subseteq \Sigma'_T$, pour une correspondance Σ'. Nous aurons $T' =>\Sigma o \Sigma'_{Th}$ et, ainsi, le *point de vue* «VP» sera aussi applicable à «T'». Ce fait soulève deux questions. Premièrement, une instance de «V» fonctionnera-t-elle avec «T'» comme elle fonctionne avec «T»? Deuxièmement, si V' = T'**as** VP [Th ->$\Sigma o \Sigma'_{T'}$], quelle est la relation entre «V» et «V'»?

La réponse à la première question est oui, grâce au polymorphisme (*"subtyping polymorphism"* [Card 85]); «V» est un agrégat qui possède une composante dont le type statique est «T», et ce type peut changer dynamiquement à «T'».

La deuxième question est une instance du problème connu des types génériques [Pals 90]. Par exemple, considérons la classe paramétrée PILE[T] et deux types «T_1» et «T_2» tels que $T_1 \subseteq T_2$; quelle est la relation entre PILE[T_1] et PILE[T_1]? Si l'interface de PILE[T] contient la fonction «void push(T)» qui place une valeur au sommet de la pile, alors «PILE<T_1>::push(T_1)» n'est pas contravariante avec «PILE<T_2>::push(T_2)». Pour étudier les différents cas, nous considérons la *vue* comme l'union de trois interfaces, correspondant aux deux clauses du *point*

de vue, i.e. **requires**(V) et **provides**(V). Naturellement, si V = T **as** VP et V' = T' as VP où T ⊆ T', alors **requires**(V) ⊆ **requires**(V').

IV.5.2.1. Sous typage entre vues

Prenons l'exemple où VP[Th] est un *point de vue* et T et T' sont deux types satisfaisant la théorie Th tels que T ⊆ T'. Dans le cas où V = T **as** VP et V' = T' **as** VP, nous aurons V ⊆V' si :

- aucune méthode dans l'interface **provides** de VP ne prend l'objet de base (T ou T') comme argument, et
- T est une extension pure de T', par exemple, tous les types qui composent T' se retrouvent tels quels dans T.

Sachant qu'une méthode qui fait partie de l'interface **provides,** et donc de la *vue* générée, peut prendre comme argument l'objet de base lui-même, la version de cette méthode pour «V» ne sera pas contravariante avec la version de «V'»; la première condition interdit de telles méthodes. Dans les faits, cette condition n'est pas restrictive car, en principe, l'interface **provides** est indifférente de la technique d'implantation et ne devrait pas être consciente de l'objet de base. En vertu de la deuxième condition, aucune composante du type «T» n'est un sous-type de la composante correspondante de «T'». En fait, ceci correspond au sous-typage tel que supporté par les langages de programmation.

Notons enfin qu'une relation de sous-typage ne veut pas forcément dire héritage et compatibilité à l'affectation. Considérons en effet la classe «Camion» et sa sous-classe «Camionnette», et les *vues* «FCamion» et «FCamionnette», générées à partir du *point de vue* «CapitalFinancier». Il se peut que «FCamionnette» implante un sous-type de «FCamion», mais si «FCamionnette» n'est pas une *sous-classe* de «FCamion», le compilateur ne permettrait pas l'affectation suivante (ligne (5)):

```
(1)  Camion c;
(2)  Camionnette ct;
(3)  FCamion* fc = new FCamion(c);
(4)  FCamionnette* fct = new FCamionnette(ct);
(5)  fc = fct;
```

Pour la rendre permissible, il suffit de redéfinir l'opérateur "=" pour les arguments (FCamionnette,FCamion), ou de remplacer cette affectation par une autre qui fait un *cast* explicite par exemple : (5')fc = (FCamion*)fct. Ce problème ne se présente pas quand on utilise le langage Java car nous n'avons pas besoin de sous-classe pour le sous-typage.

IV.5.3. Objets avec vues et héritage

Prenons l'exemple où «T» est un type et VP_1, ...,VP_k sont k *points de vues* applicables à «T», à partir desquels on a généré k *vues* V_1, ...,V_k, qu'on a importées (avec #include) dans un programme. Soit «x» un objet de type «T» auquel on a attaché *m vues* parmi les *k potentielles*, dont certaines sont actives et d'autres ne le sont pas. Soit «T'» un sous-type de «T» et VP_{k+1},...,VP_{k+n} *n* autres *points de vue* applicables à «T'», dont certains sont des spécialisations de *points de vue* «VP_i» pour 1<=i<=k, à partir desquels on a généré les *vues* V_{k+1},...,V_{k+n}, qu'on a importé dans le même programme. Soit «y» un objet du type «T'» auquel on a attaché p *vues* parmi V_1,...,V_{k+n}. La question majeure que l'on se pose est, peut on écrire :

```
(1)x = y;
```

tant sur le plan validité statique des types, que sur le plan «substituabilité comportementale».

Pour répondre à la question, nous avons défini trois types pour chaque objet «x» :

i) son *type de base*, représenté par la fonction «typeDeBase(x)», et qui représente le type (T) de «x» dans le sens du langage de programmation,

ii) son *type actuel*, qui consiste en le *type de base*, plus les *vues* présentement attachées, et représenté par la fonction «typeActuel(x)», et

iii) son *type maximal*, qui consiste en le type de base, plus toutes les *vues* potentielles, qu'elles soient présentement attachées ou pas, représenté par la fonction «typeMaximal(x)».

Nous pouvons calculer le «typeDeBase(.)» et le «typeMaximal(.)» au moment de la compilation, alors que le «typeActuel(.)» n'est disponible qu'au moment de l'exécution. Ces trois types satisfont la propriété suivante :

$$\text{typeDeBase}(x) \subseteq \text{typeActuel}(x) \subseteq \text{typeMaximal}(x)$$

Si T ⊆ T', et si x et y sont deux objets de type T et T', respectivement, apparaissant dans la même portée (espace de noms), alors :

$$\text{typeDeBase}(x) \subseteq \text{typeDeBase}(y)$$

$$\text{typeMaximal}(x) \subseteq \text{typeMaximal}(y)$$

Par contre, il n'y a pas de relation entre «typeActuel(x)» et «typeActuel(y)».

Notons que la possibilité d'avoir des *vues* générées à partir du même *point de vue* pour des classes hiérarchiquement liées, ou à partir de *points de vue* qui sont hiérarchiquement reliés, soulève un éventail de nouveaux problèmes. Tout d'abord, notons que si nous avons créé une *vue* pour un objet de base, nous devons refuser toutes les requêtes ultérieures de création de la même *vue* pour cet objet : les *vues* ayant leurs propres variables d'états, on risque de perdre des données historiques. Il en résulte que chaque objet de base doit maintenir une *table* des *vues* indexée par les (noms des) *points de vue* correspondants. Cela étant dit, que faire si après avoir attaché une instance de «FCamion» (*point de vue* «CapitalFinancier») à un camion «x», on essaie de lui attacher une instance de «FECamion» (*point de vue* «CapitalFinancierEpuisable», qui est une spécialisation de «CapitalFinancier»)? Pour bien répondre à cette question, nos outils doivent générer des structures statiques contenant la hiérarchie de *points de vues* que nous pourrons consulter durant l'exécution. Notons que notre implantation des *points de vues* ne supporte pas la spécialisation de *points de vues*, et cette question sera reléguée aux versions ultérieures.

IV.6. Conclusion

Nous venons de voir dans ce chapitre le modèle formel de *vues* et de *points de vue* de notre approche.

Nous avons présenté les *points de vue* comme étant des tranches de code génériques paramétrées sur un ensemble de *types* et de fonctions et applicables aux classes pour générer les *vues*. Nous avons montré les formalismes et les notations formelles de ***TYPES*** et de *théorie* utilisés pour la définition des *points de vue* ainsi que leur relations.

La génération des *vues* se fait par une étude de correspondance entre les membres de la classe et ceux du *point de vue*. Il faut tout d'abord établir un *"mapping"* entre la partie **requires** et la classe et trouver pour chaque membre de la partie **requires**, un correspondant dans la classe. Une fois cette liste complétée, elle sera utilisée pour définir les membres de la *vue*. Ces membres sont ceux de la partie **provides** du *point de vue* alors que certaines transformations y sont ajoutées.

Finalement, nous avons parlé des relations qui peuvent relier les *points de vue* et les *vues*. En particulier, nous avons présenté des relations de spécialisation/ généralisation en les définissant théoriquement.

Ce chapitre termine la présentation formelle de notre approche. Dans le chapitre suivant, nous parlerons de l'analyse pratique des *points de vues* ainsi que les imputées aux outils développés dans le but d'offrir la programmation par *vues*.

Chapitre V

Support pour la programmation par vues en C++

Notre implantation de la programmation par vues a été guidée par quatre contraintes majeures:

- Introduire la programmation par *vues* dans les langages existants, plutôt que développer notre propre langage sur mesure,
- ne pas toucher à la machine virtuelle (ou modèle d'exécution) du langage hôte pour s'assurer de la portabilité de notre solution,
- introduire le moins de nouvelles constructions possibles dans le langage hôte, et
- introduire le moins de surcharge intellectuelle possible aux programmeurs pour utiliser la programmation par *vues*.

Ces contraintes nous ont imposé une approche basée sur la transformation de code pour supporter la programmation par *vues*. Ainsi, nous avons choisi de supporter la programmation par *vues* en C++, qui est (était) un langage de programmation très répandu. Les constructions et pratiques de programmation que nous avons introduites dans le langage C++ sont traitées par un ensemble d'outils pré-processeurs au compilateur C++ standard. Ces outils permettent une manipulation des concepts et mécanismes de la programmation par *vues* de manière plus ou moins transparente pour l'usager, notamment, sans une grande surcharge dans la programmation conventionnelle. Dans ce chapitre, nous décrivons les fonctionnalités des outils et les principes de leur implantation.

La section V.1 donne une vue d'ensemble des outils. La section V.2 décrit la structure des objets avec vues en C++ et les fonctionnalités de gestion des *vues*. La section V.3 décrit la syntaxe des *points de vue*. La section V.4 décrit la syntaxe pour la spécification des *vues* ainsi que le traitement sous-jacent. Dans la section V.5, nous décrivons les transformations que nous effectuons sur le code de l'usager de la programmation par *vues* pour que cette utilisation soit transparente. Les limites de nos outils sont décrites dans la section V.6. Nous concluons

dans la section V.7.

V.1. Vue d'ensemble

Notre travail d'implantation est divisé en deux parties. La première porte sur l'analyse syntaxique du code en entrée et du stockage des données analysées dans les structures de données. La deuxième traite de la génération des nouvelles classes, des nouvelles fonctions et du code C++ qui seront utilisés au moment de l'exécution.

L'entrée de nos outils de support contient, à part le code C++, la syntaxe relative aux *vues* et aux *points de vues* et les fonctions d'attachement, de détachement, d'activation et de désactivation des *vues*. La définition des *points de vue* ressemble à la définition des classes de C++. La déclaration des *vues* se fait par une instruction qui lie une classe à un *point de vue*. Cette instruction sera traitée par le générateur des *vues* afin de générer les classes *vues* avec du code C++. Les fonctions qui permettent la manipulation des *vues* (attach, detach, activate et desactivate) sont prédéfinies dans des classes à part. Elles seront utilisées comme de simples fonctions C++. La figure 5.1 montre l'ensemble des outils développés, leurs entrées et leurs sorties.

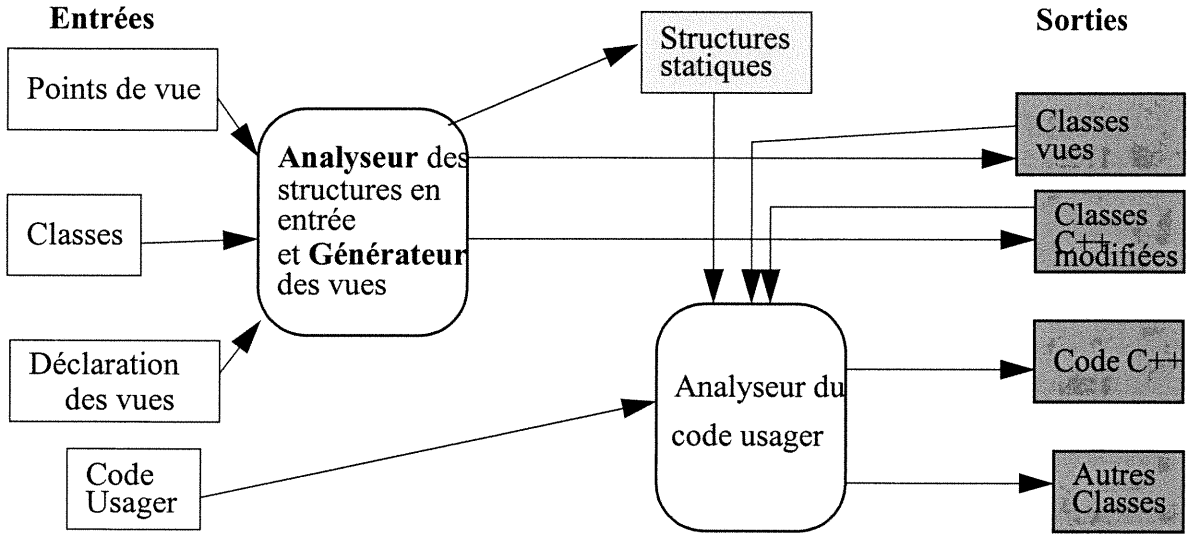


Figure 5.1 : Support pour les vues

V.2. Structure des objets avec vues

Les objets manipulés sont des instances des *classes de base* et peuvent comporter un certain nombre d'instances de *vues*. Ces dernières sont accessibles seulement à travers les objets de base auxquels elles sont reliées. On y accède pour exécuter des fonctions qui leur appartiennent. La liaison entre les objets et leurs instances de *vues* en est une à deux sens. D'un côté, les objets maintiennent une liste de leurs *vues* potentielles auxquelles un accès est effectué quand une fonction d'une *vue* attachée et active est demandée. De l'autre côté, chaque instance d'une *vue* contient une variable d'instance qui correspond à l'objet de base auquel elle est reliée. Cette référence est utilisée pour déléguer les appels des fonctions et des attributs de la *classe de base* à l'objet de base. Ainsi, l'exécution des fonctions de la *vue* qui utilise des attributs et des fonctions de la *classe de base* se fait dans le contexte de l'objet de base. La figure 5.2 schématise les liens entre un objet et sa vue.

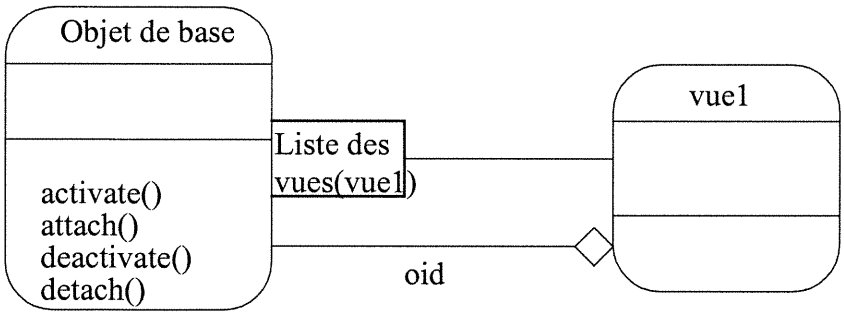


Figure 5.2 : Lien entre un objet et sa vue

Les fonctions qui permettent la manipulation des *vues* sont ajoutées aux *classes de base* (voir figure 5.2), car c'est à partir de l'objet de base que les appels d'attachement, de détachement, d'activation et de désactivation sont faits. Ces fonctions seront définies dans une superclasse dont héritent toutes les *classes de base*. Nous démontrons, ci-dessous et sous forme de commentaires, le comportement de chaque fonction. Le code C++ des fonctions ainsi que les classes dans lesquelles elles sont définies seront présentées dans le chapitre six.

```
attach( char* unevue) {
// vérifier si la vue appelée "unevue" est une vue
// potentielle de l'objet
   // si oui, vérifier si elle est déjà attachée
       // si oui rien faire
       // sinon, créer une instance de la vue "unevue"
       // établir les liens entre l'instance de la classe
       // et celle de la vue en modifiant la variable oid
       // de l'instance de vue et la liste des vues
       // attachées et actives de l'objet de base.
   // sinon rien faire
}

detach (char * unevue ){
// vérifier si une instance de la vue "unevue"
// est déjà détachée
   // sinon, rien faire.
   // Si oui, supprimer l'instance de vue et mettre à
   // jour la liste d'instances attachées et actives.
}

activate(char* unevue) {
// vérifier si la vue appelée "unevue" est attachée
// la rendre active
}

deactivate(char* unevue){
// vérifier si la vue appelée "unevue" est attachée et active
// la désactiver
}
```

À part ces fonctions, nous avons ajouté aux *classes de base* des variables d'instances pour emmagasiner les informations sur les instances des *vues* manipulées. Parmi ces variables se trouve une liste statique des *vues potentielles* de la classe ainsi qu'une liste des instances des *vues* présentement reliées à l'objet de base. Les *vues potentielles* sont celles générées à partir de la classe et les *points de vue*. Leurs instances peuvent être reliées aux instances de la *classe de base*. Pour les classes *vues* nous avons ajouté une variable d'instance qui représente l'objet de base pour lequel une instance de *vue* est créée. Nous y retrouvons aussi un constructeur, défini par le générateur de *vue*, qui consiste à créer une instance de *vue* pour un objet de base. Les classes *vues* contiennent une variable d'instance qui représente l'objet de base. Un exem-

ple concret d'une *classe de base* et d'une *classe vue* ayant été générées est donné dans la figure 5.3. Nous y retrouvons les listes qui ont été ajoutées à la *classe de base*, notamment la liste *"static"* des *vues potentielles* de la classe et la liste des instances de *vues*, dans laquelle nous spécifions si ces dernières sont actives ou non.

```
class Camion
{
public:
Camion();
TypeNSérie getNumeroSérie();
Marque getMarque();
Year getAnnéeModele();
Date getDateAchat();
float getCharge();

protected:
void setNumeroSérie(TypeNSérie);
void setMake(MakeType);

private:
TypeId _id;
static Vue     _vuepossible[];
Vue _vues[].;
} // La classe de base
```

```
class FCamion
{
public:
FCamion(Camion *);
float getValeurRésiduelle();
static float getTauxAmortisse-
ment();
float getPrixAchat();

protected:
void setPrixAchat(float);

private:
Camion* _camion;
float _prixAchat;
} // La vue Finance
```

Figure 5.3 : La classe Camion avec sa vue financière

V.3. Les points de vue

Les *points de vue* sont développés indépendamment des applications qui les utilisent et sont des structures génériques réutilisables. Nous présentons ci-après la syntaxe utilisée pour leur définition et la manière dont l'analyse est effectuée.

V.3.1. Format des points de vue

Les *points de vue* sont des structures de données abstraites qui encapsulent des données et des fonctions. Leurs membres de la partie **requires** sont ceux qui doivent avoir des correspon-

dants dans les classes auxquelles ils seront appliqués afin de générer des *vues*. Ceux de la partie **provides** sont ceux qui figureront dans la *vue* générée.

Les règles de grammaire de la définition d'un *point de vue* et de ses données et fonctions ressemblent à celles de la définition d'une classe avec ses membres privés, protégés ou publiques dans C++. Leur syntaxe est la suivante :

```
viewpoint_déclaration::
            viewpoint identificateur définition_Du_Pv
définition_Du_Pv  :: '{' déclaration '}' ';'
déclaration ::
           requires ':' déclarations_variables
         | requires ':' déclarations_méthodes
         | provides ':' définitions
déclarations_variables ::
          // déclarations des variables avec les règles C++
déclarations_méthodes ::
          // déclarations des méthodes avec les règles C++
définitions :: // définitions des méthodes selon C++
identificateur :: // identificateur C++
```

L'écriture en gras correspond aux jetons terminaux. Ces derniers font partie de la syntaxe du langage de programmation utilisé. Les autres jetons sont non terminaux. Ils seront raffinés par des règles de grammaire pour générer à la fin des jetons terminaux.

La déclaration d'un *point de vue* est faite par le mot réservé **viewpoint**, qui est suivi de son identificateur. Ce dernier doit respecter les règles de syntaxe des identificateurs de C++. Il correspond à une chaîne de caractères alphanumériques qui ne contient pas de caractères spéciaux et qui ne commence pas par un chiffre. Il peut toutefois contenir le caractère « _ ». Cet identificateur est suivi des accolades contenant la définition des attributs et des fonctions du *point de vue* et d'un point virgule comme dans le cas de la définition d'une classe C++. Entre les accolades nous avons les attributs et les fonctions du *point de vue* définis dans les parties **requires** et **provides**. Le mot réservé **requires** est suivi de la déclaration des membres requis et le mot réservé **provides** est suivi de la définition des membres fournis du *point de vue.* Le code dans la partie **provides** obéit à la syntaxe des attributs et fonctions de C++.

Ces règles de grammaire qui correspondent aux *points de vue* sont intégrées dans l'ensemble des règles de C++ au même niveau que les règles de déclaration et de définition des classes.

V.3.2. Analyse des points de vue

L'analyse syntaxique du fichier d'entrée est effectuée à l'aide d'un programme «Lex» qui reconnaît les jetons et les fait passer à l'analyseur «Yacc» contenant les règles de grammaire du langage (à la C++). Chaque instruction du programme en entrée doit respecter un ordre dans l'ensemble des règles de grammaire et son analyse est ascendante. À partir des jetons terminaux qui forment le code en entrée, nous pouvons monter dans l'analyse pour trouver les règles générales formées des jetons non terminaux dans les règles de grammaire. Si de telles règles n'existent pas, l'analyseur génère un message d'erreur.

Au moment de l'analyse, un arbre syntaxique est construit au fil de son évolution Il renferme toutes les informations sur les règles de grammaire parcourues et les instructions analysées. Cet arbre sera utilisé pour assurer la vérification du code, la génération des structures de données qui emmagasinent les informations et la génération ultérieure du code des *vues*. Il nous indique en tout temps l'état de l'analyse et spécifie dans quel ensemble de règles il se trouve. Les structures de données que nous avons définies lors de l'analyse du code contiennent l'ensemble des *points de vue* et des *classes* qui sont correctement analysés. Les détails de l'implantation de toutes les structures de données utilisées sont donnés dans le chapitre suivant.

À part les informations versées dans l'arbre syntaxique et dans les structures de données, notre outil doit, pour chacun des *points de vue*, mémoriser un identificateur unique. Cela se fait, comme dans le cas d'une *classe*, dans une table de symboles. Cette table contient tous les identificateurs des *classes*, des *points de vue* et des *vues*. Les attributs et les fonctions définis dans le *point de vue* sont analysés. Ils respectent les règles de grammaire de C++ et doivent être uniques pour un même *point de vue*. Chacun des membres est représenté dans une structure de données par son identificateur, son type et, dans le cas des méthodes, la liste de ses paramètres assortis de leurs types.

V.4. Les vues

L'introduction des *vues* dans le code en entrée se fait au moyen de déclarations liant une *classe de base* à un *point de vue*. Le générateur des *vues* traite l'analyse de cette déclaration, la vérification de l'applicabilité du *point de vue* à la classe en établissant la correspondance entre les membres de **requires** du *point de vue* et les membres de la classe, et la génération des *vues,* dont le code ressemble à celui d'une classe C++.

V.4.1. Syntaxe de la déclaration des vues

Lors de la déclaration d'une *vue,* il faut spécifier i) son identificateur, ii) la *classe de base* et iii) le *point de vue* à partir desquels la génération sera établie. Il faut, à cette fin, s'assurer d'abord que tous les membres de la partie **requires** du *point de vue* comportent des correspondants dans la *classe de base*. Cette étape n'est pas toujours facile à franchir, surtout lorsque l'on se trouve en présence de plusieurs correspondants. Par exemple, si dans la partie **requires** du *point de vue* on présente une variable de type entier et que nous avons, dans la *classe de base*, plusieurs variables de type entier, il ne nous sera pas possible de savoir quelle est la variable de la *classe de base* qui correspond à la variable du *point de vue*. Pour remédier à ce problème, nous avons permis à l'usager d'établir une liste de correspondance entre la *classe de base* et le *point de vue*, laquelle sera appelée à résoudre les problèmes de choix multiples. Après l'analyse des correspondances, le générateur traite la définition des membres de la *vue* à partir des membres de la partie **provides**, et ce, en se basant sur les correspondances établies. Les règles de grammaire qui correspondent à la déclaration des vues sont les suivantes :

```
déclaration_vue ::
    viewdef identificateur ':'
      identificateur as identificateur'('correspondance')'
correspondance::
      '<'class_method','viewpoint_method'>' correspondance
     | '<' class_var ',' viewpoint_var '>'correspondance
     | ϕ
class_method :: IDENTIFIER
viewpoint_method :: IDENTIFIER
class_var :: IDENTIFIER
viewpoint_var :: IDENTIFIER
```

Dans la déclaration d'une *vue*, on retrouve le nom, soit l'identificateur de la *vue* (tag_name), l'identificateur de la *classe de base* et l'identificateur du *point de vue*, lesquels sont séparés par le mot réservé **as**. La classe et le *point de vue* doivent avoir été déclarés préalablement, sinon la génération de la *vue* ne pourra pas être faite. L'usager identifie, à partir de la liste de correspondance, l'ensemble de variables et de méthodes de la classe assortis des correspondants explicites qui se trouvent dans le *point de vue*. Les éléments de la liste de correspondance doivent figurer dans la *classe de base* et la partie **requires** du *point de vue*.

Un exemple de déclaration de classe, *point de vue* et *vue* d'une classe est présenté dans la figure 5.4. On y montre que la variable requise «valeur» du *point de vue* peut comporter deux correspondants dans la classe, soit «taux» et «prix», ce qui donne lieu à une ambiguïté. La liste de correspondance résoud cette ambiguïté en faisant correspondre «prix» à «valeur».

```
class Véhicule {
private:
  double taux;
  double prix;
public:
  double getprix() { return prix;}
  void settaux(double d){taux = d;}
};

viewpoint Entretien {
requires:
  double_valeur;
  double valeurÉstime();
provides:
  double tarif;
  void settaux(double var) { cout << valeurÉstime();}
  double getCoutEntretien() {return tarif;}
  void setvaleur() {valeur = valeur*tarif;}
};

viewdef EVéhicule  : véhicule as Entretien
                                  (<prix, valeur>);
```

Figure 5.4 : Déclarations de classe, point de vue et vue

V.4.2. Générateur des vues

Une fois une déclaration d'une *vue* rencontrée, le générateur de *vues* doit générer un code C++ d'une classe représentant la *vue*. Cette classe consiste en une forme comparable à la forme et aux caractéristiques qui constituent une classe C++ définie explicitement. Elle est définie par construction à partir des membres de la classe et du *point de vue* qui figurent dans sa déclaration. Elle contient, parmi ses membres privés, une référence à l'objet de base pour lequel une instance est créée. Cette référence est appelée **oid**, dont le type est un pointeur vers la *classe de base*, et sera utilisée pour les appels de délégation aux membres de sa classe de base.

Le générateur des *vues* doit tout d'abord s'assurer que l'identificateur de la *vue* spécifiée n'a pas déjà été défini et que la classe et le *point de vue* ont déjà été déclarés et définis. Cette vérification est effectuée au moyen d'un processus d'évaluation des structures de données construites par notre outil lors de l'analyse des classes et des *points de vue*. Par la suite, le générateur de *vues* étudie la possibilité que le *point de vue* soit appliqué à la classe, processus mené par l'analyse de l'ensemble des membres requis du *point de vue*. Il faut, pour chacun de ces membres, trouver un correspondant si celui qui lui est assorti n'est pas explicitement spécifié dans la liste de correspondance. Ce dernier est un membre auquel a été attribué un même rôle dans la *classe de base* (un substituant). Nous nous sommes limités, pour cette correspondance, à la vérification des types des variables et de la signature des fonctions. Ainsi, le code de *vue* généré de la figure 5.4 est présenté dans la figure 5.5.

```
class EVéhicule {
private:
  double tarif; // défini dans le point de vue
  Véhicule * oid;
public:
  EVéhicule(){ ... } // constructeur
  double getCoutEntretien() {return tarif;}
  void settaux(double var) { cout<< oid->getprix();}
  void setvaleur() {oid->prix= oid->prix*tarif;}
};
```

Figure 5.5 : La vue générée

V.5. Transformation du code usager

Le travail de l'analyseur du code usager consiste à mener analyse de chaque instruction du code usager et d'effectuer sa transformation lorsque besoin il y a, sinon elle ne sera pas changée dans le fichier final. Certains problèmes sont reliés à l'introduction des *vues* et à leur manipulation dynamique, et sont traités lors de l'analyse du code usager. Citons par exemple, le typage statique de C++ qui nous permet de toujours connaître le type des objets manipulés de même que l'ensemble des méthodes et attributs accessibles par ces objets. La programmation par *vues* sous-entend que les objets changent de comportements et que l'ensemble des membres accessibles par l'objet n'est plus statique. Le comportement des objets n'est connu qu'au moment de l'exécution et change selon l'ensemble des *vues* actives. Il est ainsi encore tôt, au moment de la compilation, pour savoir si un comportement donné sera disponible au moment de l'appel. L'analyseur du code effectue alors la transformation du code pour éviter de générer une erreur au moment de la compilation et pour reporter la vérification jusqu'au moment de l'exécution. Une *vue* peut être utilisée si sa définition est importée dans le programme. C'est le générateur de *vues* qui s'occupe de sa génération, et c'est le pré-processeur C++ qui effectue son importation au moyen de la directive "*include*". Nous présentons ci-après divers cas de comportement définis dans le code usager.

V.5.1. Cas de comportement uniquement défini

Dans le cas de comportement uniquement défini, une fonction peut être définie une seule fois, soit dans la *classe de base*, soit dans une des *vues*. Les figures 5.4 et 5.5 nous montrent la fonction «getprix()», qui est uniquement définie dans la *classe de base*, et la fonction «getCoutEntretien()», qui est uniquement définie dans la *vue* de la classe. Un exemple d'un code usager d'un programme qui manipule des objets de la classe «Véhicule» avec la vue «Entretien» est donné dans la figure 5.6.

```
(1) #include "Véhicule.h"
(2) #include "EVéhicule.h"
(3) Véhicule * unVéhicule = new Véhicule();
(4) EVéhicule * eunVéhicule = new EVéhicule(unVéhicule);
(5) cout << unVéhicule->getprix(); // de la classe de base
(6) cout << unVéhiculee->getCoutEntretien(); // de la vue
(7) unVéhicule->settaux(400.0); // de la classe et de la vue
(8) unVéhicule->déactivate("EVéhicule"); // Vue plus active
(9) unVéhicule->settaux(300.0); // de la classe de base
(10)unVéhicule->setvaleur(); // de la vue non active
```

Figure 5.6 : Le code usager

Lors de l'analyse du code de la figure 5.6, les deux cas relatifs au comportement unique sont ceux des lignes (5) et (6). Le générateur du code usager traite l'analyse des fonctions appelées et doit trouver leurs implantations. Nous avons, dans ces deux lignes, la fonction «getprix()» définie dans la *classe de base*. Son implantation est alors disponible pour l'objet et aucun changement n'est nécessaire pour accéder à son implantation. Ainsi, la ligne (5) demeure telle quelle dans le code usager final. Pour la ligne (6), la fonction «getCoutEntretien()» est définie dans une seule *vue*. La classe de base de l'instance «unVéhicule» n'a pas d'accès direct à l'implantation de la fonction, d'où une transformation est nécessaire. Elle consiste à faire une délégation à l'implantation dans la vue. Ainsi la ligne (6) est transformée en:

```
(6') (EVéhicule *)(unVéhicule->getView("EVéhicule"))
                                    ->getCoutEntretien();
```

La fonction «getView(char * unevue)» effectue la recherche de l'instance de la *vue* appelée «unevue» passée en paramètre. La liste des instances de *vues* fait partie des membres de la *classe de base* et est mise à jour chaque fois que les états des *vues* changent. Dans notre exemple, la *vue* «EVéhicule» contient la fonction «getCoutEntretien()». Le compilateur ne générera ainsi aucun message d'erreur pour la ligne (6') comme il le ferait si le code n'était pas changé.

V.5.2. Cas de comportement défini dans plusieurs composantes

Outre le fait qu'elle ajoute de nouveaux membres, une *vue* peut redéfinir des membres de la *classe de base* comme elle peut les généraliser/spécialiser. De même, une *vue* peut comporter des membres qui figurent dans d'autres *vues* de la même *classe de base.* Ces membres peuvent être définis par enrichissement de fonctionnalités, selon un nouveau *point de vue*, tout comme ils peuvent consister en des redéfinitions complètes. Les conflits se présentent quand plusieurs *vues* sont actives et qu'elles offrent les mêmes fonctions. Quelle fonction doit-on appliquer ou comment faut-il générer un nouveau comportement résultant de plusieurs fonctions?

Pour résoudre ce problème, nous avons associé, pour chacune des *classes de base* «X», une classe «_Comb_X» qui renferme la définition des méthodes qui figurent à la fois dans plusieurs *vues* ou qui figurent dans la *classe de base* et dans une ou plusieurs *vues*. La définition de ces méthodes est formée de l'union des définitions dans les diverses *vues*, c'est-à-dire que si une méthode est définie dans la *classe de base* et dans une *vue*, la méthode résultante consiste à exécuter tout d'abord la méthode de base, suivi de celle de la *vue*. Si elle figure dans plusieurs *vues* actives, il faut mener son exécution dans chacune de ces *vues*. C'est le générateur de *vue* qui traite la gestion de ces appels en définissant la *vue combinaison*. Les corps des méthodes de «_Comb_X» peuvent être définis dans des fichiers C++ séparés, auxquels l'utilisateur peut accéder pour apporter des changements sur les définitions. Cela permet une flexibilité au niveau du choix du code à exécuter et du comportement désiré.

Il faut, lors de la définition des membres de «_Comb_X», prendre en considération tous les scénarios possibles d'attachement et d'activation de *vues*. On consultera la liste des instances de *vues* attachées à l'objet de base pour savoir si leurs comportements doivent être pris en considération. Un exemple de définition de la méthode «settaux(double)» de la figure 5.4, définie dans la classe de base et offerte par la vue «EVéhicule», est présenté dans la figure 5.7.

```
void _Comb_Véhicule::settaux(double x)
{ oid->settaux(x); // settaux est définie dans Véhicule
  if (active("EVéhicule"))
      // appel du comportement dans la vue
      oid->getView("EVéhicule")->settaux(x);
}
```

Figure 5.7 : Définition d'une méthode de _Comb_Véhicule

La classe «_Comb_X» est définie comme une *classe vue* de la *classe de base*. Pour chacune des instances de la *classe de base*, une instance de cette *vue* est automatiquement créée, attachée et activée. Elle le sera tout au long de la durée de vie de l'instance de base **oid** pour laquelle elle a été créée et qu'elle contient dans ses variables d'instance. Parmi les membres de la *classe de base* se trouve la variable d'instance **_combView**, qui sera utilisée pour les appels des membres de la *vue de combinaison.*

Notons que l'exemple de la figure 5.7 n'effectue le test que pour la seule *vue* qui existe. L'implantation de la classe *vue de combinaison* devient plus lourde quand le nombre de *vues potentielles* de la classe de base s'élargit considérablement.

Les lignes (7) et (9) de l'exemple de la figure 7.6 nous montre des appels à une fonction définie dans deux composantes, soit la classe et la *vue*. Ces lignes seront transformées de la manière suivante :

(7') `unVéhicule->_combView->settaux(400.0);`

(9') `unVéhicule->_combView->settaux(300.0);`

Dans le premier cas, la *vue* « EVéhicule » est active, alors l'appel de la fonction consiste à faire deux appels consécutifs, c'est-à-dire l'implantation de la classe de base et l'implantation de la *vue*. Dans le deuxième cas, la *vue* étant inactive, seul le code de la *classe de base* sera exécuté.

Notons, finalement, que pour le cas des fonctions des *vues* définies dans une seule *vue* ou dans plusieurs *vues*, leur appel peut être remplacé par l'appel à la *vue de combinaison* qui traite l'acheminement.

V.5.3. Générateur de vues revisité

Le générateur des *vues*, tel que présenté au paragraphe V.4, suppose que si une fonction de la *classe de base* est utilisée par la *vue*, c'est son implantation dans la *classe de base* qui sera exécutée. Prenons l'exemple d'une classe «C» ayant la fonction «save()». Cette dernière consiste à faire la sauvegarde de toutes les variables d'instances de la classe «C», tout comme un *point de vue* «Pv1» applicable à la classe «C» qui contient dans sa partie **requires** une fonction «maj()» qui correspond à la fonction «save()». Lors de la génération du code de la *vue* résultant de l'application du *point de vue* à la classe, l'appel de la fonction «maj()» sera remplacé par un appel de la fonction «save()» par délégation en utilisant l'identificateur de l'objet de base. L'instruction «maj()» sera remplacée par «oid->save()» sans qu'aucune transformation dans le code de la fonction «save()» ne se fasse et sans possibilité que cette même fonction ne soit définie dans d'autres *vues* de la *classe de base*.

Or, si c'est le cas, l'invocation de la fonction «save()» doit permettre la mise à jour de toutes les variables d'instances de la *classe de base* et de ses *vues*, puisque c'est la manipulation du même objet avec plusieurs variables d'instances qui influencent son état. Le remplacement de l'appel par la version de la *classe de base* n'est donc ni approprié, ni sécuritaire (*"unsafe"*). C'est une instance de *"broken delegation"* qui ne prend pas en considération la délégation vers toutes les implantations disponibles. Pour remédier à ce problème, chaque appel à un membre de la *classe de base* doit être remplacé par un appel de délégation à la version définie dans la *vue de combinaison*, qui prend en considération, à la fois, la version de base et celles des *vues* actives. Donc, l'appel «oid->save()» sera remplacé par «oid->_combView->save()», ce qui implique l'invocation de toutes les fonctions de sauvegarde définies dans les *vues* actives et, par la suite, la sauvegarde de l'état globale de l'objet, de la base et de ses *vues*.

V.6. Problèmes et limitations

Certaines hypothèses ont été émises pour pouvoir résoudre automatiquement des conflits et des ambiguïtés. Ces hypothèses sont reliées d'une part à l'analyse de la correspondance établie entre le *point de vue* et la *classe de base* et, d'autre part, à la génération du code de la

vue de combinaison.

Pour établir et vérifier la liste de correspondance entre la classe et le *point de vue*, nous avons supposé que l'on doive faire correspondre à chaque membre requis dans le *point de vue* un membre de la *classe de base* ayant une forme, un identificateur, un type et une signature similaires. Dans le chapitre 4, nous avons insisté sur le fait que cette vérification ne suffit pas toujours. Il faut aussi voir l'ensemble des post-conditions et des pré-conditions de même que les règles de covariance. La vérification de type et de signature n'est pas toujours fiable. Donnons l'exemple de la même fonction définie dans une classe et dans un *point de vue* ayant ses paramètres dans un ordre quelconque. Selon les règles de correspondance touchant la signature, on ne peut trouver une correspondance pour cette fonction, et le générateur de vues n'acceptera pas l'application du *point de vue* à la classe. Une même fonction peut aussi comporter des paramètres par défaut dans une classe et non dans un *point de vue*. Donc, pour pouvoir établir la correspondance, il faut que le comportement de la fonction soit le même ou qu'elle représente la même chose dans les contextes de classe et de *point de vu*e. Nous avons essayé de minimiser ces problèmes liés à la liste de correspondance explicitement spécifiée lors de la déclaration de la vue.

Nous avons, outre les problèmes reliés à la liste de correspondance, mis en relief ceux qui touchent la définition de la *classe vue de combinaison*. Cette classe doit résoudre les conflits relatifs à l'existence de plusieurs implantations d'une même fonction, dans plusieurs *vues*, attachées simultanément à la *classe de base*. Nous avons supposé que dans le code généré, chaque fonction définie dans une *vue* attachée doit être exécutée. Or, si c'est le même comportement que la fonction offre dans deux *vues*, l'exécution est une répétition inutile pouvant parfois avoir des effets sur les variables qu'elle met à jour. Dans le cas où le comportement ne serait pas le même, l'exécution de tous les codes pourrait ne pas être la meilleure solution. On peut, par exemple, relever des comportements conflictuels. Lors de notre implantation, nous avons émis l'hypothèse que, dans tous les cas, l'utilisateur peut accéder aux fichiers que notre outil génère et changer le code de la *vue de combinaison*.

V.7. Conclusion

Dans ce chapitre, nous avons présenté la syntaxe relative aux *points de vue* et aux *vues* ainsi que les règles de grammaire ajoutées au langage C++. Nous avons parlé des différentes fonctionnalités offertes par nos outils de support pour les *vues*. L'analyseur des *points de vue* traite la vérification de la syntaxe des *points de vue* ainsi que la génération des structures de données internes pour la génération des *vues*. Le générateur de *vues* traite, pour sa part, le code des *vues* assorties des caractéristiques qui les lient aux classes de base. Nous avons aussi parlé des problèmes reliés à la génération des *vues* ainsi qu'à la génération d'une *vue* particulière qui représente le code des fonctions figurant dans plusieurs *vues* d'une même classe. Enfin, nous avons parlé du générateur du code usager qui génère un code C++ et qui permet de retarder la décision concernant les types des objets et leurs différents comportements.

Chapitre VI

Implantation des outils de support de vues

Dans le chapitre V, nous avons décrit les fonctionnalités des outils de support pour la programmation par *vues* en C++. Ces fonctionnalités étaient décrites principalement en termes des entrées-sorties des outils pré-processeurs. Dans ce chapitre, nous décrivons la conception et l'implantation de ces outils, qui constitue une grande partie de cette recherche.

La section VI.1 donne une vue d'ensemble. Dans la section V.2, nous décrivons les structures de données de l'arbre syntaxique abstrait généré et utilisé par nos outils. Les structures de données nécessaires à la gestion des *vues* durant l'exécution sont décrites dans la section VI.3. Nous concluons dans la section VI.4.

VI.1. Vue d'ensemble

Nos outils sont développés à partir d'une implantation partielle de la grammaire C++ élaborée en 1991 par James A. Roskind. Les règles de grammaire implantées sont suffisantes pour analyser un code C++ qui ne traite pas les «exceptions», c'est-à-dire les instructions "*throw*", "*try*" et "*catch*".

Nous supposons que le code en entrée est déjà traité par le pré-processeur C++ qui traduit toutes ses directives en code C++. Ainsi, le code que nos outils traitent est un code C++, sans les directives du pré-processeur, auquel nous avons ajouté la syntaxe relative aux *vues* et aux *points de vue.*

Les chaînes de caractères en entrée sont captées par l'analyseur lexical «Lex» qui les transforme en des *jetons* qu'il reconnaît et qui ont été définis par nous. Ces *jetons* correspondent aux instructions qui figurent dans le fichier d'entrée (mots réservés et autres). Les structures de données utilisées au moment de l'analyse sont définies selon la syntaxe du langage C. C'est là une exigence de l'analyseur lexical «Lex» qui ne supporte pas les classes, à la différence de l'analyseur «Flex». Le choix de l'analyseur et du langage de développement résulte

d'un simple hasard relié à des travaux antérieurs.

L'analyseur sémantique «Yacc», qui définit les règles de la grammaire, fait une analyse ascendante de l'ensemble des *jetons* qui sont lus. Il construit l'arbre syntaxique en suivant une séquence des règles de la grammaire. Ces dernières sont celles du langage C++ auxquelles nous avons ajouté des règles pour supporter la programmation par *vues*.

L'entrée des outils développés est un ensemble de fichiers qui regroupent les définitions des *points de vue*, les déclarations des *vues*, les définitions des classes C++ (ou *classes de base)* et le code usager (voir figure 6.1). Nous exigeons que les fichiers qui contiennent des instructions propres à la manipulation des *vues* portent l'extension «.cv». Ces fichiers seront analysés et transformés pour générer les classes *vues* et le code usager écrit en C++. À la sortie, nous aurons, pour chaque fichier muni de l'extension «.cv» , deux fichiers de même nom portant les extensions «.h» pour les déclarations et «.c» pour les définitions. À part les fichiers transformés, nous avons à la sortie d'autres fichiers contenant des définitions des nouvelles classes et des structures de données que nous avons introduites.

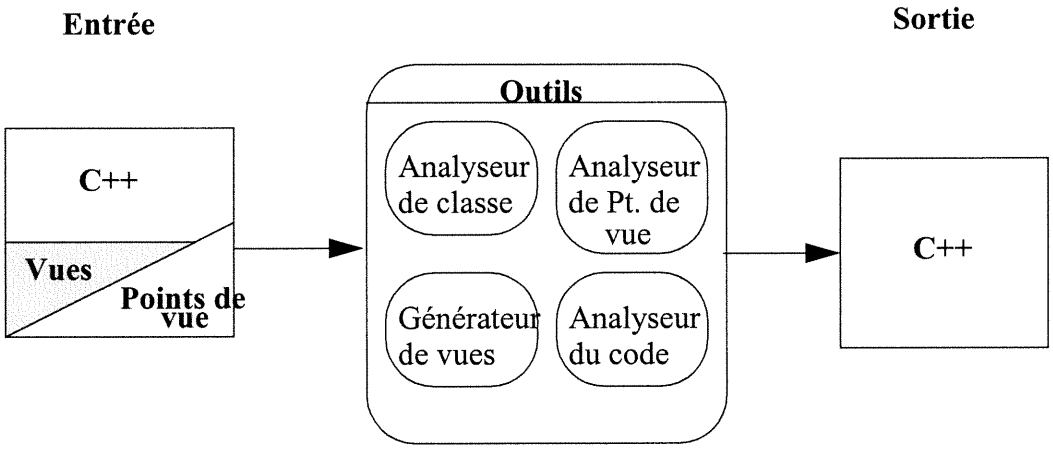


Figure 6.1 : Entrée-Sortie des outils

Deux types de structures de données ont été définis. Nous avons, d'un côté, celles qui sont utilisées au moment de l'analyse pour stocker les informations et qui ne font pas partie de la sortie. D'un autre côté, nous retrouvons les structures de données générées pour la sortie et qui seront utilisées par l'application au moment de l'exécution. Dans ce chapitre, nous présentons ces diverses structures de données, leur hiérarchie ainsi que les fichiers C++ générés.

VI.2. Structures de données utilisées pour l'analyse

Les structures de données utilisées au moment de l'analyse du code sont celles qui permettent de faire la représentation de l'arbre syntaxique, l'analyse du code du fichier source et le stockage des informations sur les *points de vue*, les *vues* et les classes.

L'arbre syntaxique est construit au fur et à mesure que nous avançons dans le fichier d'entrée. Il contient les informations sur le code lu selon les règles de la grammaire définies avec l'analyseur «Yacc». Chaque fois qu'un jeton analysé avec «Lex» apparaît, il est ajouté à l'arbre. Les jetons terminaux sont les feuilles de cet arbre et les autres forment ses noeuds internes. La structure des noeuds de l'arbre est définie dans la figure 6.2. Elle est relative à un jeton et représente son nom, son identificateur s'il est un jeton terminal, son type et ses descendants. Les sous-arbres de l'arbre syntaxique représentent une instruction, un bloc d'instructions du programme, lequel peut être une déclaration (de classe, de méthode, de variable, de *points de vue* ou de *vue*) ou une définition.

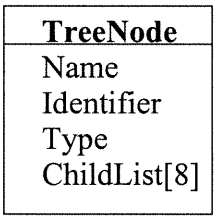


Figure 6.2 : Structure des noeuds de l'arbre syntaxique

Lors du parcours du fichier en entrée et de la génération de l'arbre syntaxique, d'autres structures de données sont générées pour contenir des informations sur les classes, les *points de vues* et les *vues*. Ces structures sont représentées dans les figures 6.3, 6.4 et 6.5.

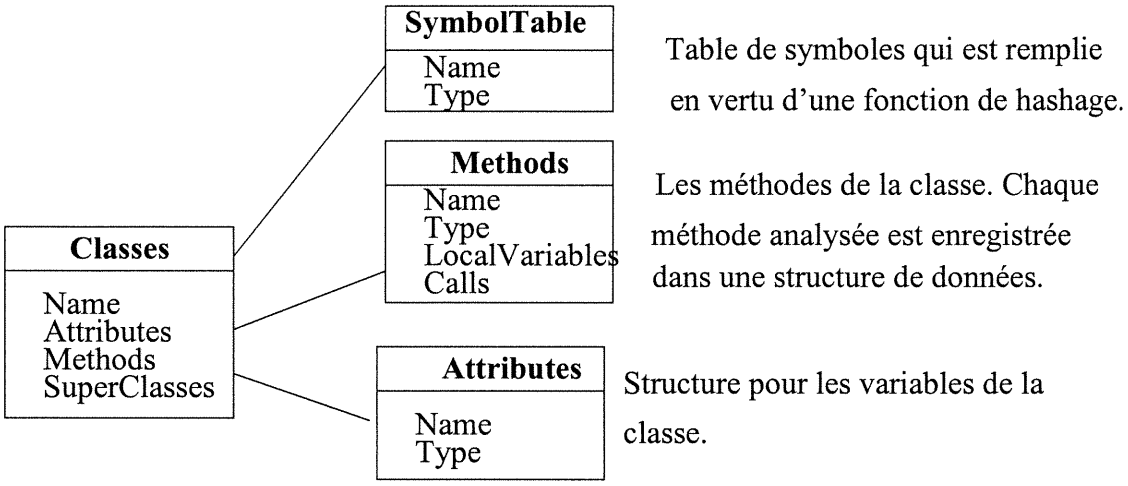


Figure 6.3 : Structures des classes et de leurs membres

Les *points de vues* ainsi que les *vues* sont insérés dans des structures de données qui ressemblent à celles des classes.

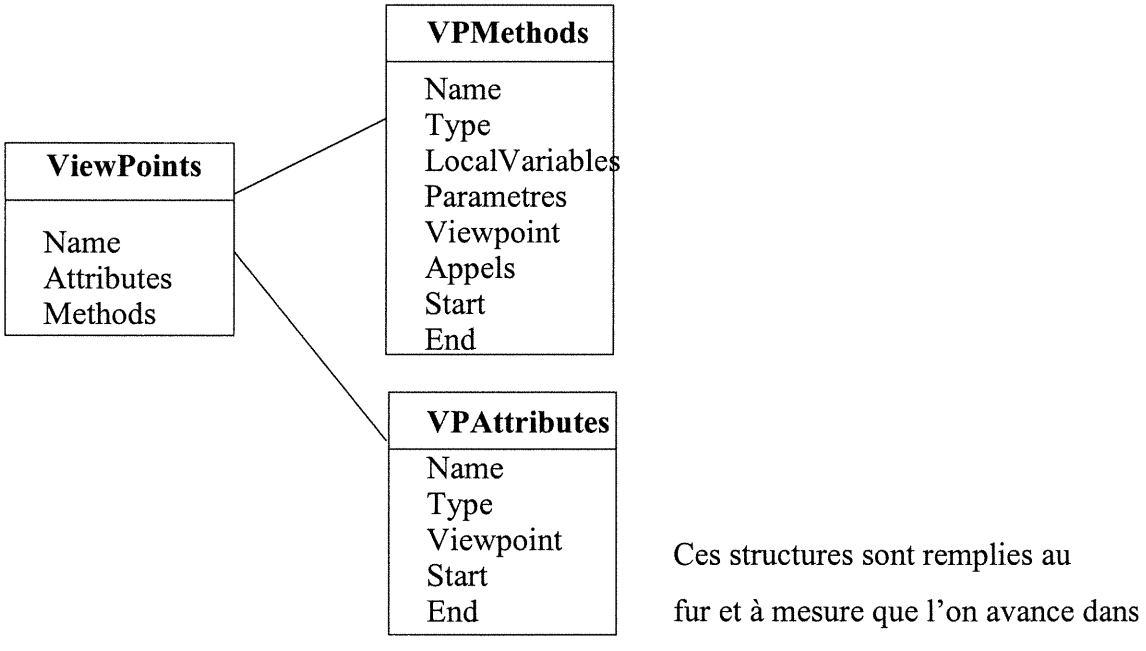


Figure 6.4 : Structures des points de vues et de leurs membres

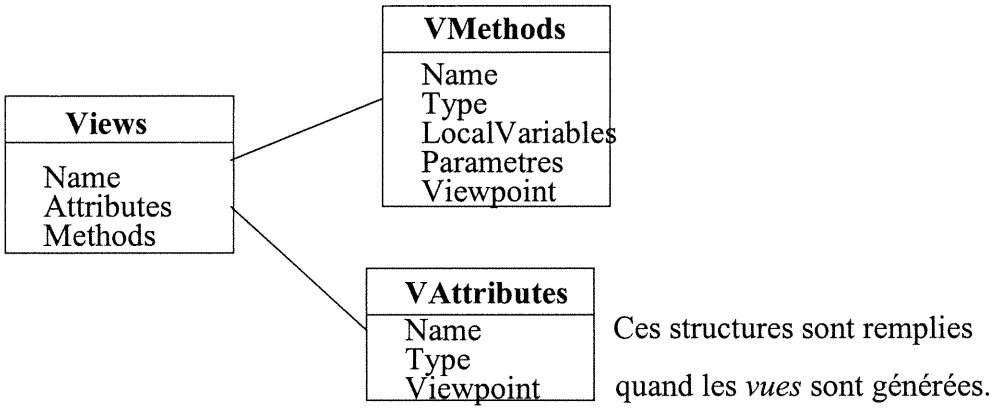


Figure 6.5 : Structures des vues et de leurs membres

Outre les structures qui sont présentées ci-dessus, nous avons défini des structures aux fins des informations concernant la correspondance entre les *classes de base* et les *points de vues*. Ces structures sont utilisées pour la génération des *vues* et pour la génération des listes des *vues potentielles* des classes.

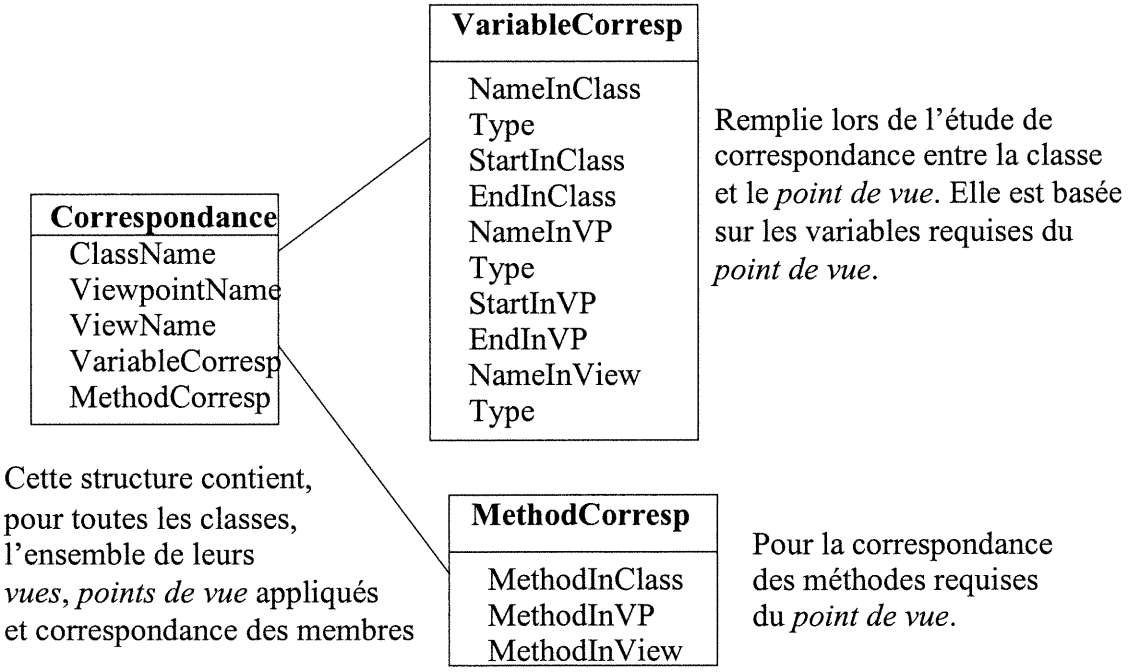


Figure 6.6 : Structures de correspondance entre classes et points de vue

VI.2.1. Implantation des structures

Les implantations sont faites par des structures de données qui respectent la syntaxe du langage C. Nous les représentons en expliquant les choix que nous avons fait pour l'implantation.

L'arbre syntaxique

L'arbre syntaxique contient toutes les informations utiles à l'analyse du code en entrée. Il nous dit si nous sommes à analyser une instruction ordinaire, une instruction de définition ou une instruction de déclaration d'un *point de vue*, d'une classe, d'une méthode ou d'une variable. Nous y enregistrons, pour chaque instruction, la règle de grammaire à laquelle elle correspond. Chaque règle doit respecter un ensemble de raffinement de règles qui conduit à un ensemble de jetons terminaux. Cela signifie que chacune des règles qui forment un noeud comporte des sous-arbres descendants qui représentent les raffinements. La racine de l'arbre est un seul noeud qui représente l'ensemble de toutes les instructions à analyser. L'implantation du noeud de l'arbre est présentée dans la figure 6.7. Nous y retrouvons l'identificateur du noeud, l'identificateur du jeton (dans le cas d'un noeud terminal), la liste de ses descendants, son emplacement dans la table des symboles (dans le cas des déclarations ou des définitions), les commentaires, le type et l'emplacement dans le fichier d'entrée. On y voit aussi les identificateurs qui nous disent si nous sommes à analyser une variable, une méthode ou un *point de vue*. Notons que les champs de cette structure seront remplis en fonction de la règle analysée. Ainsi, ils ne sont pas tous remplis pour toutes les règles.

```
typedef struct treeNode
{  char *symbName; /* symbole non terminal */
   char *tokenId;/* symbole terminal */
   treePt childList[MAXCHILDREN]; /* les descendants */
   struct symtab *symbValue;
   /* pointeur dans la table de symbole */
   char *comment; /* les commentaires rencontres */
   struct listType *pType;
   char *nameNode; /* nom du noeud de l'arbre */
   char *typeNode;
   long start; /* où apparaît le noeud dans le code */
   char *instance;
   int * parite; /*1 method, 2 variable et 3 paramètre */
   int * isViewpoint; /* 1 ou 0 */
   /* indique si on est en train d'analyser
   un point de vue (1) ou non (0) */
} TreeNode;
```

Figure 6.7 : Noeud de l'arbre syntaxique

Les points de vue

Les *points de vue* sont stockés dans une liste qui renferment les informations relatives à leurs attributs et à leurs fonctions des parties **requires** et **provides**. La structure de cette liste contient, pour chaque *point de vue*, son identificateur, les listes de ses variables et de ses méthodes des parties **requires** et **provides**, l'emplacement de sa déclaration ou de sa définition dans le code d'entrée, et un pointeur vers le *point de vue* suivant (voir figure 6.8).

```
typedef struct nodeOfViewpoint
{  char *nameViewpoint;
   ptVar listVarRequires; /* Variables requises*/
   myptMethod listMethodRequires; /* Methodes requises */
   ptVar listVar; /* Variables fournies */
   myptMethod listMethod; /* méthodes fournies */
   long startViewpoint; /* début de la définition */
   long endViewpoint;
   ptViewpoint nextViewpoint;
} NodeViewpoint;
```

Figure 6.8 : Points de vues

Les variables et les méthodes des points de vue

```
typedef struct nodeOfVar
{  char *nameVar;
   char *typeVar;
   ptViewpoint definingViewpoint; /* PV de la variable*/
   long startVPVar; /* début de la définition */
   long endVPVar;
   ptVar nextVar;
} NodeVar;

typedef struct nodeOfMethod
{  char *nameMethod;
   char *typeMethod;
   ptVar parameterMethod /* paramètres de la méthode*/
   ptVar variableMethod /* paramètres de la méthode*/
   ptViewpoint definingViewpoint;/*où elle est définie*/
   ptlist_appel Appel[100]; /* les appels dans le code */
   long startVPMethod; /* début de la définition */
   long endVPMethod;
   myptMethod nextMethod;
} NodeMethod;
```

Figure 6.9 : Les variables et les méthodes

Les noms des *points de vue* sont uniques. Ainsi, leur liste nous permet de vérifier l'existence d'un *point de vue* lors de la création (génération) d'une *vue*. Nous accéderons aux listes des méthodes et à celles des variables par le truchement du générateur des *vues* afin de trouver la liste de correspondance qui existe entre la classe à laquelle nous voulons appliquer le *point de vue*, et afin de créer les variables et les fonctions de la *vue* à générer.

Les variables et les méthodes sont représentées par des structures de données qui seront consultées au moment de l'analyse du code d'entrée. Ces structures sont présentées dans la figure 6.9. Nous obtenons, pour chacune des variables, l'identificateur, le type, l'emplacement dans le fichier et le *point de vue* dans lequel elle est définie. Pour chacune des méthodes, nous obtenons le type, l'identificateur, la liste de paramètres (assortis de leurs types), la liste des appels qu'elle fait, le point de vue dans lequel elle est définie ainsi que son emplacement dans son fichier (son début et sa fin).

La figure 6.10 représente les structures internes et la manière dont elles sont liées entre elles et utilisées par les outils.

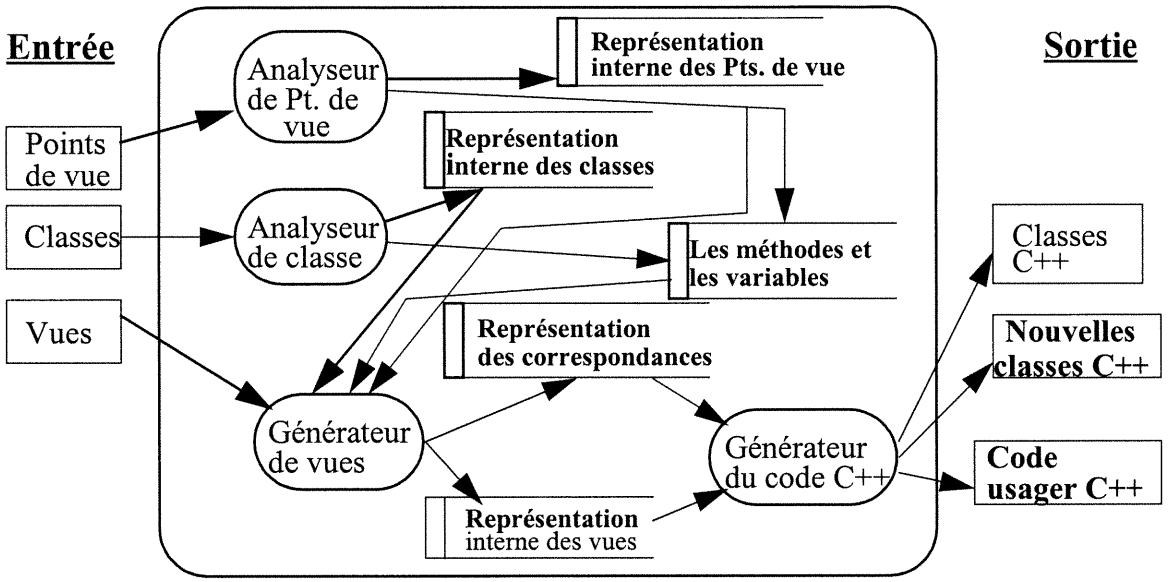


Figure 6.10 : Liaison entre les structures des données et les outils développés

VI.3. Structures de données consultées durant l'exécution

Afin de supporter la programmation par *vues,* nous avons ajouté quelques fonctions aux classes C++ définies dans le code d'entrée. Ces fonctions sont regroupées dans une super-classe appelée **Viewable_classe** et dont héritent toutes les classes définies dans le programme. De même, nous avons regroupé les caractéristiques communes à toutes les classes *vues* dans une super-classe, que nous avons appelée **View**. Toutes les classes *vues* héritent de cette classe. Les caractéristiques communes aux classes sont les suivantes : la liste de leurs *vues potentielles*, la liste des instances de *vues* créées pour chacun de leurs objets, l'instance de la *vue de combinaison* **_combview** ainsi que les fonctions d'attachement, de détachement, d'activation et de désactivation.

La liste des *vues potentielles* est statique et renferme la liste des classes *vues* générées à partir de la classe et des *points de vue*. Elle est consultée chaque fois qu'une demande d'attachement de *vue* est faite. En outre, elle est initialisée par une fonction que nous créons pour chaque application et qui constituera la première instruction exécutée par le programme.

Cette fonction fait appel aux structures de correspondances générées au moment de l'analyse.

La liste des instances des *vues* est mise à jour chaque fois qu'une instance de *vue* est attachée, détachée, activée ou désactivée pour l'objet de base.

L'instance de la *vue de combinaison* est utilisée pour faire des appels de délégation vers les fonctions de sa classe, laquelle définit les fonctions qui sont présentes simultanément dans plusieurs *vues.*

La super-classe **View** regroupe les caractéristiques communes à toutes les classes *vues.* Elle contient le nom de la *vue* et la variable d'instance **oid** qui représente l'objet de base auquel l'instance de la *vue* est liée. Les constructeurs et destructeurs sont propres à chaque *classe vue*. Ils sont ainsi définis explicitement par le générateur des *vues* et ne font pas partie de la super-classe des *vues*.

Le nom de la *vue* est utilisé par les fonctions d'attachement, de détachement, d'activation et de désactivation des *vues*.

La variable d'instance **oid** est utilisée pour faire des appels de délégation vers les fonctions et attributs de la *classe de base*.

À des fins d'implantation, nous avons défini d'autres classes et fonctions que nous représentons dans la figure 6.11.

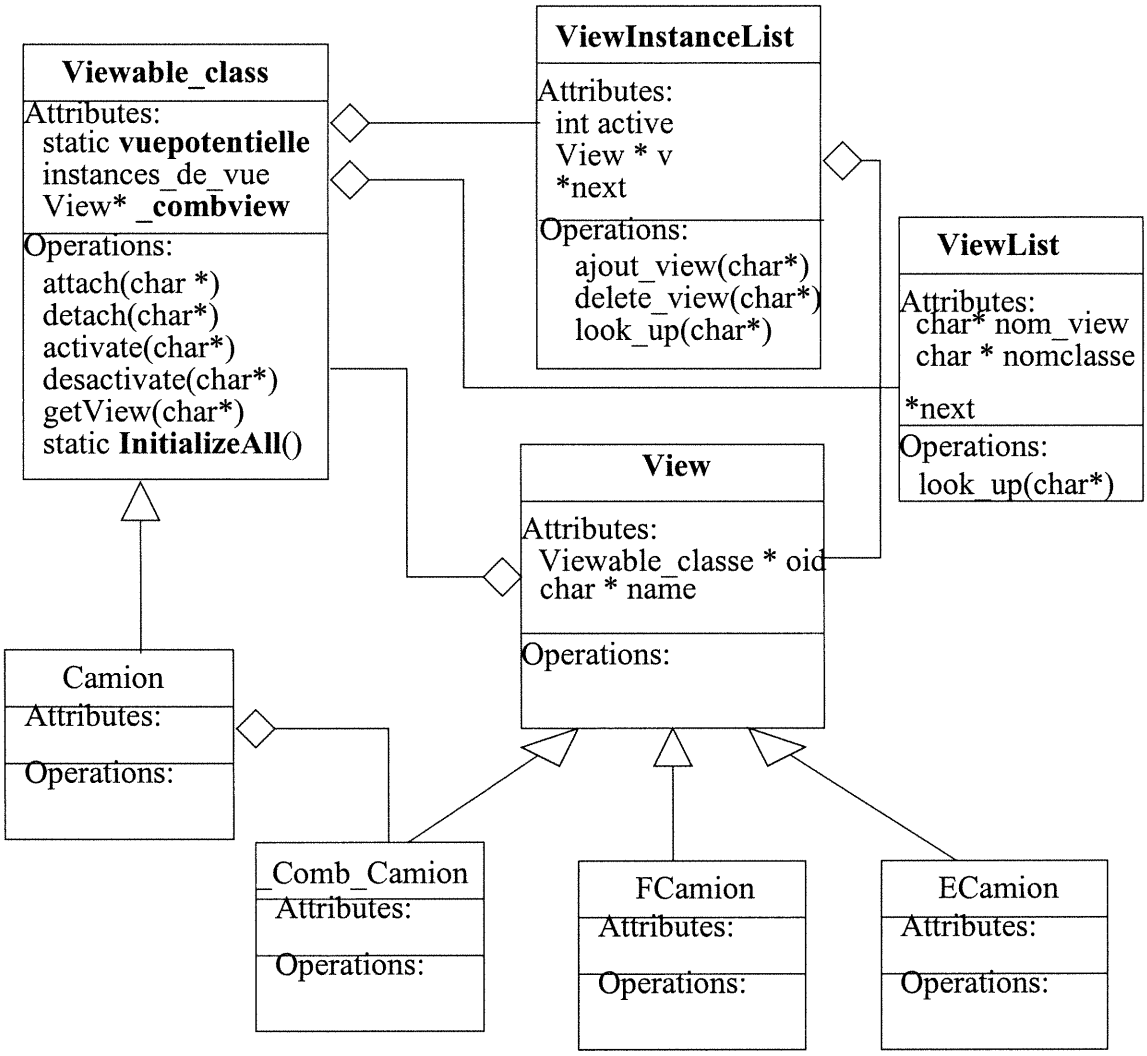


Figure 6.11 : Hiérarchie des classes générées

VI.3.1. Description des classes

La classe View

La classe **View** est la super-classe de toutes les *vues*. Elle contient une variable d'instance de la classe **Viewable_classe**, qui représente l'objet de base auquel l'instance de la classe *vue* est liée; elle contient aussi un nom qui est utilisé par les fonctions de manipulation des instances des *vues*. Ce nom est utilisé pour s'assurer que nous ne créons pas plusieurs

instances de *vues* à partir de la même classe *vue* et pour le même objet de base. Dans cette classe *View*, nous avons redéfinit l'opérateur "**!=**" qui nous permet de comparer deux noms de *vues*.

La classe ViewInstanceList

Elle représente l'ensemble des instances des *vues* reliées à un objet de base particulier et leurs caractéristiques. Nous avons, pour chacune des instances, le nom de sa classe de même qu'une information selon laquelle cette classe est active ou non active.

La classe ViewList

Elle représente la liste des *vues potentielles* d'une classe, le nom de la *vue* ainsi que le nom de la classe pour laquelle elle a été créée.

La classe Viewable_classe

C'est là la super-classe de toutes les classes. Chacune de celles qui sont définies dans le programme hérite explicitement de **Viewable_classe**, et cet héritage est assuré par une instruction que nous ajoutons au début de toutes les classes.

La super-classe contient la liste des instances des *vues* pour l'objet de la classe pour chacun de ses objets ou de ceux de ses descendants. Cette liste renferme les instances des *vues* et le nom de leur classes *vues*. Elle contient aussi la liste des *vues potentielles* et l'instance de la *vue de combinaison*. Dans sa partie publique, on trouve les méthodes relatives à la manipulation des *vues*, soit **attach**, **activate**, **detach** et **desactivate**.

Les classes Camion, FCamion, ECamion et _Comb_Camion

Ces classes sont introduites à titre d'exemple dans la figure 6.11. La classe «Camion» représente une *classe de base.* Les classes «FCamion» et «ECamion» représentent, pour leur part, des classes *vues* qui sont générées par notre outil de génération de *vues*, et la classe «_Comb_Camion» représente la *vue de combinaison* de la classe «Camion».

VI.3.2. Implantation des classes View et Viewable_class

Le code de certaines classes définies ci-haut est présenté dans ce qui suit:

```
// Super classe de toutes les classes
class Viewable_classe {

private:
 // Les variables d'instances
// la liste des vues possibles de la classe (statique)
// La liste des instances des vues attachées aux objets
// L'instance de la vue de combinaison
   static ViewList *vuepotentielle;
   ViewInstanceList * instance_de_vue;
   View *_comb_view;

 public:
 // Fonction pour attacher une vue dont on connaît le nom
   void attach(char * v) {
     if (vuepotentielle->look_up(v)!=0)//Est-elle applicable
      exit(1); // Pas dans la liste des vues potentielles
     else
     { v * unevue = new v(this);
           // créer une instance de vue pour l'objet
       instance_de_vue->ajout_view(v); // ajout à la liste
     }
    }


// Super classe des vues
class View {
 private:
   Viewable_class *_oid; // référence à l'objet de base
   char *aname; // pour connaître le nom de la classe
   friend int operator !=(View *v1, View *v2)
        {return ((strcmp(v1->aname,v2->aname)!=0)?1:0);}
};
```

```
// Définition des fonctions de manipulation des vues

// Supprimer une instance de vues de la liste
void detach(char * v){
  instance_de_vue->supprimer_view(v);}

// Activer l'instance de la vue v
void activate (char * v) {
  if (instance_de_vue->look_up(v)) // chercher l'instance
    {ViewInstanceList * unevue =
                        instance_de_vue->look_up(v);
    unevue->active =1;}
  }

// désactiver la vue
void desactivate(char * v) {
 if (instance_de_vue->look_up(v))
    {ViewInstanceList * unevue =
                        instance_de_vue->look_at(v);
    unevue->active =0;}
  }

// Fonction qui initialise la liste des vues potentielles
    de toutes les classe.
static void initializeAll();

}; // fin de Viewable_class
```

Informellement, la fonction **attach** prend, en paramètre, le nom d'une *vue,* crée une instance pour cette vue en paramètre et l'attache à l'objet de base. Cette fonction consiste en une mise à jour de la liste des instances des *vues* de l'objet de base et de sa référence **oid**. Précisons que le rôle de la fonction de détachement est opposé à celui de la fonction **attach**. Cette fonction supprime l'instance de la *vue* et la retire de la liste des instances des *vues*.

La fonction **activate** rend disponibles les attributs et les fonctions de la classe *vue*, passée en paramètre. Quand l'instance de la *vue* est attachée à un objet de base, son comportement est disponible ou ne l'est pas. Nous précisons si chaque instance des *vues* est attachée et si elle est active ou non.

VI.3.3. Code des fonctions définies dans le code C++ généré

Constructeur d'une vue

Le constructeur d'une classe *vue* est défini par le générateur des *vues*. Il consiste à initialiser l'instance de la *classe de base* (**_oid**) et le nom de la *vue*, à activer l'instance de la *vue* créée et à l'ajouter à la liste des instances des *vues* actives de l'objet de base **_oid**. Un exemple du constructeur de la *vue* «FCamion» nous est donné dans la figure 6.12.

```
FCAmion::FCamion(Camion *core) // constructeur
{
aname = new char[8];
strcpy(aname,"FCamion");
_oid->activate(aname);
_oid = core;
}
```

figure 6.12 : Le constructeur de la vue FCamion

Fonction d'initialisation des vues potentielles d'une classe

Lors de l'analyse des classes, des *points de vues* et de la génération des *vues*, nous avons créé des structures de données visant à stocker les informations relatives aux *vues* générées. Ces structures sont utilisées pour générer le code de la fonction qui fait l'initialisation des *vues potentielles* des classes. Dans le cas de la classe «Camion» à laquelle nous appliquons le *point de vue* «Finance», la fonction d'initialisation, que nous avons appelée **InitializeAll()**, est définie comme suit :

```
void Viewable_class::initializeAll()
{
 // variable d'instance statique de Viewable_class
 // regroupe les vues possibles de toutes les classes
 vuepotentielle = new ViewList;

 // La première classe Véhicule
 vuepotentielle->nomclasse = new char [6 +1];
 strcpy(vuepotentielle->nomclasse, "Camion");

 // Pour la classe Camion, on fait l'initialisation
 // de la liste de ses vues potentielles
 struct Liste *laliste, *preced;
 laliste = new Liste; // laliste contiendra les vues

 // La première vue, FCamion, de la classe
 laliste ->vue = new char [ 7+1];
 strcpy(laliste->vue, "FCamion");
 laliste->next = NULL;
 preced = laliste; // utilise s'il y a d'autres vues

 // listevue est la variable qui contient la liste des
 // vues potentielle de la classe appelée nomclasse
 vuepotentielle->listevue = laliste;

 // Une seule classe, le reste des vues potentielles
 // des classe étant vide
 vuepotentielle->next = NULL;
 }
```

VI.4. Conclusion

Nous avons vu, dans ce chapitre, les diverses structures de données générées et utilisées par nos outils de support aux fins de la programmation par *vues*.

D'une part, nous avons les structures de données internes qui sont utilisées durant l'analyse et qui contribuent à la génération du code C++ final et, d'autre part, les structures de données générées pour l'exécution et utilisées et intégrées telles quelles dans le code C++. En conclusion, nous insistons sur le fait que ce travail au chapitre de l'implantation donne un sens pratique à la programmation par *vues*, sans laquelle la recherche n'est que pure théorie. Ce fut

un travail ardu étant donné la complexité et le nombre important de règles de la grammaire manipulée et qui sont interreliées.

Chapitre VII

Exemple et discussion

Dans ce chapitre, nous présentons un simple exemple de la programmation par *vues* qui nous permet de voir son utilité et ses avantages par rapport à d'autres types de programmation. Par cet exemple, nous définissons les étapes de développement d'une application qui manipule des objets ayant deux *vues*. Cette application résulte de l'intégration de deux applications qui manipulent les mêmes objets.

La section VII.1 donne un aperçu de l'exemple. Dans la section VII.2 nous décrivons les deux applications et leurs classes dans le cas de développement sans *vues*. Dans la section VII.3 nous présentons le cas de développement avec *vues*, nous décrivons les *points de vue*, les *classes de base* et les *vues* de l'application à développer. Le code des *points de vue* ainsi que des classes et des *vues* générées sera donné. Un exemple d'un code usager qui permet la manipulation des objets assortis de leurs *vues* sera aussi présenté. Une discussion sur la programmation par *vues* est faite à la section VII.4 et une comparaison entre la programmation par *vues* et d'autres approches est faite dans la section VII.5. Nous concluons à la section VII.6.

VII.1. Choix de l'exemple

L'exemple est inspiré de celui qui a été proposé par les chercheurs de la programmation *orientée-sujet* [http : //www.research.ibm.com/POS]. Il consiste en deux applications : une concernant la «Paie» du personnel d'une entreprise et une autre touchant la «Localisation» des employés de cette dernière. Les deux applications manipulent les objets d'une classe «Employé». Ainsi, en les intégrant, cette classe adoptera deux comportements différents, soit un pour chacune des applications. Nous pouvons représenter les comportements, reliés aux applications, par des *vues* de la classe «Employé».

VII.2. Description

Pour les deux applications de «Paie» et de «Localisation», nous définissons les mêmes classes «Employé» et «Liste_Employé». La classe «Employé» représente les employés, et la classe «Liste_Employé» représente des listes des employés que nous pouvons manipuler. Dans les deux applications, ces classes sont munies de caractéristiques communes, qui représentent leurs caractéristiques de base. Elles peuvent aussi comporter des caractéristiques propres à chaque application. Ce sont les caractéristiques reliées aux *points de vue* qu'elles peuvent avoir. Nous expliquons ci-après comment ces deux applications peuvent être développées simultanément, soit en partant du cas où les applications sont développées indépendamment les unes des autres, vers celui où on les compose ensemble, donnant ainsi lieu à un comportement multiple des objets qu'elles manipulent simultanément. Ce processus fait l'objet de notre étude (comportement multiple et intégration).

VII.2.1. L'application de paie

Dans cette application, les classes «Employé» et «Liste_Employé» définissent des objets ayant les attributs et les fonctions nécessaires à la réalisation des tâches de la paie. Les attributs d'un employé sont sa «matricule», son «nom», le «poste» qu'il occupe et le «salaire» qu'il reçoit. La classe «Employé» est définie par l'ensemble de ces attributs, la méthode «print()», qui imprime les informations d'un employé, et les méthodes «set_...» et «get_...», qui permettent l'accès aux attributs et leur modification.

La classe «Liste_Employé» est définie comme suit : une instance «unemployé» de la classe «Employé», un pointeur à l'employé «précédent» et un autre au «suivant», une méthode «print()» qui imprime la liste, une méthode «insert()» qui ajoute un élément à la liste, et les méthodes «set_...» et «get_...» qui servent à modifier les variables d'instance d'une liste et qui permettent d'y accéder (voir figure 7.1).

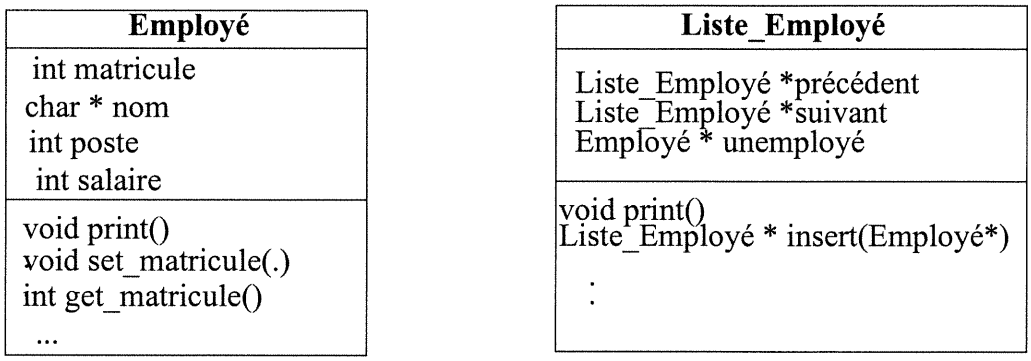


Figure 7.1 : Classes pour l'application de «Paie»

VII.2.2. L'application de localisation

Comme l'application de «Paie», l'application de «Localisation» définit les classes «Employé» et «Liste_Employé». La classe «Employé» renferme un «identificateur», un «nom», une «adresse_détaillée» (c.-à-d. rue, numéro d'appartement, province, ville et code postal) et les fonctions d'accès et de modification des attributs. La classe «Liste_Employé» est définie de la même façon que celle de l'application de «Paie» (voir figure 7.2).

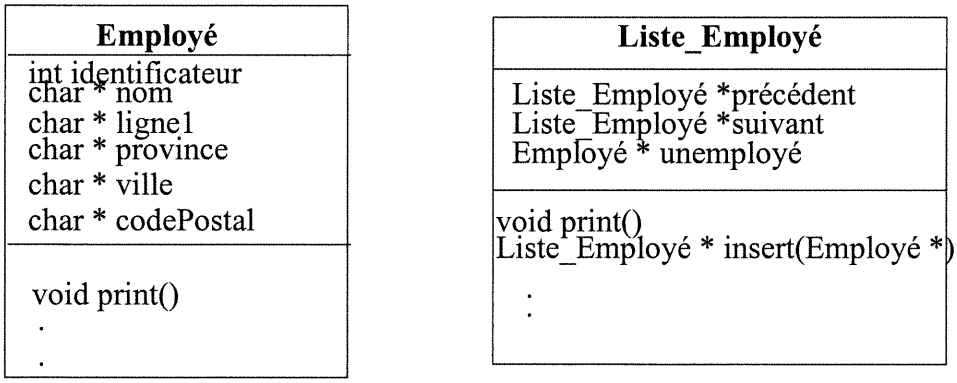


Figure 7.2 : Classes pour l'application de «Localisation»

Notons ici que pour l'application de «Localisation», nous n'avons pas pu réutiliser les classes définies dans l'application de «Paie», car les deux applications sont développées indépendamment l'une de l'autre, et aucune collaboration entre les équipes de développement n'a été prévue. Cette indépendance offre une sécurité des données car

chaque application manipule ses propres objets sans interférence. Cependant, nous avons une redondance de définitions de fonctions, de classes et de lignes de code, principalement en raison du fait que les deux applications manipulent les mêmes objets.

Il nous faut, dans nombre de situations, composer des applications qui manipulent les mêmes objets, et une réutilisation de ces dernières pourrait entraîner une réduction des coûts de développement. Dans le cas de composition des deux applications, il faut prendre en considération l'existence des composants déjà définis et des objets qui y sont manipulés simultanément. Les méthodes traditionnelles de composition consistent à utiliser la relation d'héritage entre les classes pour spécialiser une classe déjà existante par un ensemble d'attributs et de fonctions. Le travail de composition consiste, par exemple, à passer du cas où une application est déjà développée au développement d'une nouvelle application qui fait usage des composants de la première. Il est aussi possible de développer les deux applications et de les composer simultanément par la suite.

VII.2.3. Développement avec réutilisation

Soulevons ici la question de la réutilisation et de la composition, alors que la réutilisation permet d'accroître la productivité, et la composition résulte de la réutilisation des composants. Supposons que l'application de «Paie» est déjà développée. Le développement de l'application de «Localisation» consiste, par la suite, à utiliser les définitions des classes «Employé» et «Liste_Employé» et à les spécialiser pour la nouvelle application. La relation d'héritage nous permet de créer la classe «EmployéLocalisation», qui hérite de la classe «Employé». Pour sa part, la classe «Liste_Employé» sera réutilisée telle quelle dans l'application de «Localisation» (voir figure 7.3).

Dans la classe «EmployéLocalisation», il faut redéfinir la fonction «print()» pour imprimer les nouveaux attributs d'un employé. Les autres fonctions et attributs de la classe «EmployéLocalisation» sont ceux qui sont hérités de la classe «Employé» et auxquels nous avons ajouté de nouveaux attributs et de nouvelles fonctions.

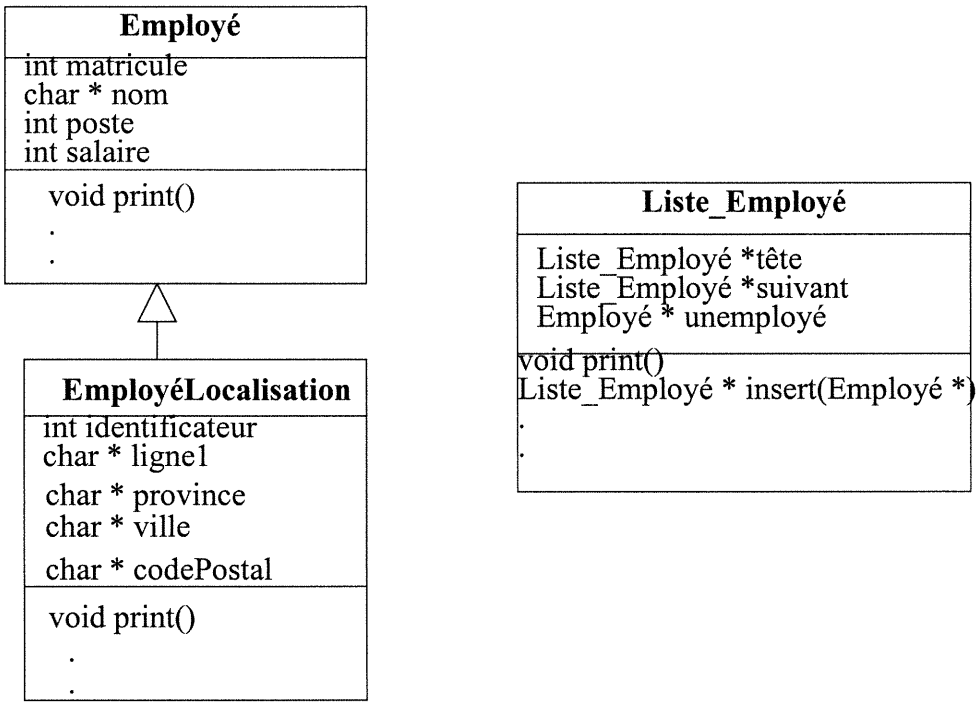


Figure 7.3 : Composition sous forme d'héritage

Remarquons que l'attribut «salaire» de la classe «Employé» fait partie des attributs de la classe «EmployéLocalisation», bien qu'il ne soit pas utile pour l'application de «Localisation». C'est là un des inconvénients de l'utilisation de la relation d'héritage.

Soulignons aussi un problème relié à la nomenclature des attributs et des fonctions dans les classes des deux applications. Par exemple, les variables «matricule» et «identificateur», définies respectivement dans les classes «Employé» et «EmployéLocalisation», identifient d'une façon unique un employé. Ainsi, elles devaient être représentées par une seule variable pour les deux classes.

Finalement, en raison d'un défaut de l'utilisation de la relation d'héritage et de son applicabilité, il faut faire la distinction entre un développement planifié, où l'on possède les deux applications, et un développement parallèle. Dans le cas d'un développement planifié, on veillera à ne pas utiliser le même nom des classes, et ce, pour pouvoir en faire l'héritage multiple par la suite. Si, dans notre exemple, les deux applications sont définies et nous désirons les composer, il faut que la classe «Employé» de chacune des applications porte un nom différent. Nous pouvons, lors de la composition par héritage

multiple, hériter des deux classes qui représentent les employés. Or, dans un cas plus général, nous avons accès à des applications développées indépendamment. Ainsi, nous ne pouvons pas garantir que des noms différents seront attribués aux classes. Par conséquent, l'application de la relation d'héritage multiple ne sera pas évidente. Nous pouvons, pour résoudre ce problème, vérifier l'utilisation de *"namespace"*, qui consiste à identifier les classes par leurs applications.

VII.3. Développement avec vues

Il est possible d'opérer la composition des deux applications en utilisant le développement avec *vues* de la manière suivante : i) créer deux *points de vue* qui correspondent aux deux applications, ii) créer les *classes de base* et iii) dériver les *vues* à partir des classes et des *points de vue*.

Pour un développement avec *vues*, nous définissons le *point de vue* «Paie» qui détermine les caractéristiques reliées à la «Paie», et le *point de vue* «Localisation» qui détermine les caractéristiques reliées à la «Localisation». Nous définissons aussi la *classe de base* «Employé» qui regroupe les caractéristiques de base d'un employé. Au moyen de nos outils, nous générons les *vues* «LEmployé» et «PEmployé» en appliquant les *points de vue* à la classe (voir figure 7.4).

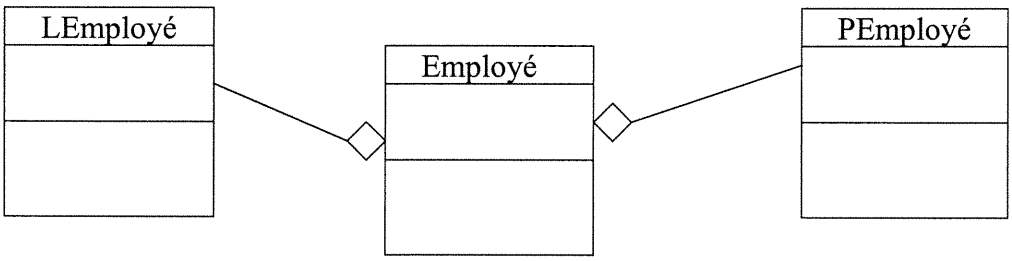


Figure 7.4 : Une classe comportant deux vues

VII.3.1. Les points de vue

Le point de vue Paie

Dans un contexte de paie, nous pensons à une rémunération pour un travail qui consiste en un nombre d'heures et un taux horaire spécifiques. Nous pouvons aussi faire appel aux fonctions de déductions pour les impôts et les assurances, les cumuls des reve-

nus bruts et des déductions, le bulletin de paie, etc. Pour simplifier notre exemple, nous allons nous limiter aux fonctions et attributs que voici : une «matricule», un «poste», un «salaire», une fonction «print()» pour imprimer les données, et les fonctions de modification et d'accès aux données. Selon le *point de vue* «Paie», ceci est applicable aux classes dont les instances peuvent, par une matricule, identifier les employés d'une façon unique. Donc, la partie **requires** du *point de vue* est formée de la «matricule». Les autres attributs et fonctions qui ont été cités forment la partie **provides** et sont propres au *point de vue* (figure 7.5).

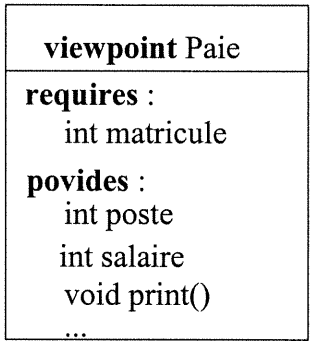


Figure 7.5 : Le point de vue Paie

La déclaration du *point de vue* «Paie», dont nous donnons la définition ci-après, se fait comme suit :

```
viewpoint Paie {
    requires :
      int matricule;
    provides :
      int poste;
      int salaire;
      void print();
      int get_poste();
      void set_poste(int);
      int get_salaire();
      void set_salaire(int);
  };
```

La définition des fonctions du *point de vue* «Paie» est la suivante :

```
int   Paie :: get_poste() {return(poste);}
void  Paie :: set_poste(int X) {poste=X;}

int   Paie :: get_salaire() {return(salaire);}
void  Paie :: set_salaire(int X) {salaire=X;}
```

```
void Paie :: print () {
  cout << matricule;
  cout << get_poste()<<" "<<get_salaire()<<endl;
}
```

Le point de vue Localisation

Quand l'on parle de la localisation, on pense à une adresse complète comportant un nom de rue, un code postale, un numéro d'appartement, une ville et une province. Cette adresse est relative à une chose que l'on peut identifier. Ainsi, le *point de vue* «Localisation» est défini par un «identificateur», qui constitue sa partie **requires**, une «adresse» et une fonction «print()», qui constituent sa partie **provides** (voir figure 7.6).

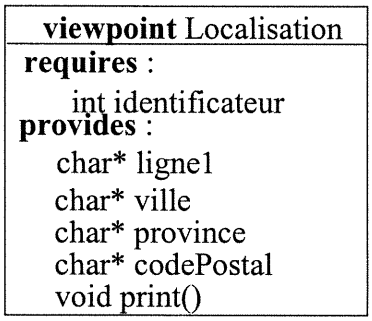


Figure 7.6 : Le point de vue Localisation

Voici le code de la déclaration du *point de vue* «Localisation» :

```
viewpoint Localisation {
    requires :
     int identificateur;
    provides :
     char* ligne1;
     char* ville;
     char* province;
     char* codePostal;
     void print();
     char* get_ligne1(char* ); void set_ligne1(char* );
     char* get_province(char*);
     void set_province(char* );
     char* get_ville(char* ); void set_ville(char* );
     char* get_codePostal(char*);
     void set_codePostal(char*);
};
```

La définition des fonctions du *point de vue* «Localisation» :

```
char* Localisation :: get_ligne1(char* X)
          {X=new char [strlen(ligne1) +1];
          strcpy(X,ligne1);return(X);}
void  Localisation :: set_ligne1(char* X)
          {ligne1=new char [strlen(X) +1];
          strcpy(ligne1,X);}

char* Localisation :: get_province(char* X)
{X= new char [strlen(province) +1];
strcpy(X,province); return(X);}
void  Localisation :: set_province(char* X)
{province = new char [strlen(X) +1];
strcpy(province,X);}

char* Localisation :: get_ville(char* X)
{X= new char [strlen(ville) +1];
strcpy(X,ville); return(X);}
void  Localisation :: set_ville(char* X)
{ville = new char [strlen(X) +1];
strcpy(ville,X);}

char* Localisation :: get_codePostal(char* X)
{X = new char [strlen(codePostal) +1];
strcpy(X,codePostal); return(X);}
void  Localisation :: set_codePostal(char* X)
{codePostal = new char [strlen(X) +1];
codePostal= new char strcpy(codePostal,X);}

void Localisation :: print () {
char * T;
cout << identificateur;
cout<<get_ligne1(T)<<endl;
cout<<get_ville(T);
cout<<","<<get_province(T)<<endl;
cout<<get_codePostal(T)<<endl;
};
```

Notons que les *points de vue* «Paie» et «Localisation» peuvent être appliqués à des classes autres qu'«Employé», ce qui constitue pour elles un avantage majeur. Par exemple, le *point de vue* «Paie» peut s'appliquer à la classe «Professeur».

VII.3.2. Les classes de base

Les *classes de base* «Employé» et «Liste_Employé» renferment les attributs et les fonctions nécessaires à la définition de leurs objets, indépendamment des applications qui y feront appel (voir figure 7.7).

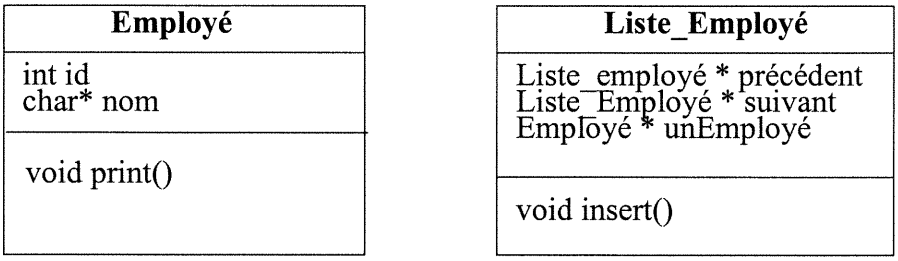


Figure 7.7 : Les classes de base

La classe « Employé »

```
// Déclaration
class Employé {
      private :
        char* nom;
        int id;
      public :
        void print();
      };

// Définition
void Employé :   : print () {
      cout<<id<<nom<<endl;
}
```

La classe « Liste_Employé »

```
// Déclaration
class Liste_Employé {
    private :
      Liste_Employé* précédent;
      Liste_Employé* suivant;
      Employé*       unEmployé;
    public :
        void print();
};
```

```
// Définition
void Liste_Employé :  : print () {
      Liste_Employé* P;
      for (P=précédent; P!=0; P=P->suivant())
      {(P->unEmployé)->print(); cout<<endl;}
}
```

Cet exemple nous amène à conclure qu'à la différence du développement sans vues, les deux classes ne sont définies qu'une seule fois. Elles renferment les membres de base communs, utiles à toute application qui les utilise. Cela offre une sécurité et un contrôle d'accès des attributs et fonctions de la partie **provides** d'un *point de vue* qui seront ajoutés aux classes de base.

VII.3.3. Les vues

Nous pouvons appliquer les *points de vues* «Paie» et «Localisation» à la classe «Employé» pour générer les *vues* «PEmployé» et «LEmployé». Le code de la déclaration des *vues* est le suivant :

```
viewdef PEmployé :  Employé as Paie();
viewdef LEmployé :  Employé as Localisation();
```

À partir de ces déclarations, le générateur des *vues* génère les *vues* présentées ci-après :

```
/************************************************/
/* fichier contenant le résultat de notre analyse */
/************************************************/
#include "entete.h" //Importer les nouvelles classes

// Première vue
class PEmployé  :  virtual public View{
      private :
        int poste;
        int salaire;
      public :
        PEmployé(Viewable_classe *core)
          {aname= new char[9];
           strcpy(aname, "PEmployé");
           oid = core;
           core->activate("PEmployé");
```

```
        }

      void print() {
       cout<<oid->id(); // appel de délégation
        cout<<get_poste()<<" "<<get_salaire()<<endl;
      }


      int get_poste(){return(poste);}
      void set_poste(int X){poste = X;}
      int get_salaire(){return salaire;}
      void set_salaire(int X){salaire =X;}
};

// Deuxième vue
class LEmployé  :  virtual public View {
    private :
      char* ligne1;
      char* ville;
      char* province;
      char* codePostal;
    public :
      LEmployé(Viewable_classe *core) {
        aname = new char[9];
        strcpy(aname, "LEmployé");
        oid = core;
        core->activate( "LEmployé");
      }

  void print () {
  char * T;
  cout << oid->id; // appel de délégation
  cout<<get_ligne1(T)<<endl;
  cout<<get_ville(T);
  cout<<"," <<get_province(T)<<endl;
  cout<<get_codePostal(T)<<endl;}

  char* get_ligne1(char* X)
        {X=new char [strlen(ligne1) +1];
         strcpy(X,ligne1);return(X);}
  void set_ligne1(char* X)
         {ligne1=new char [strlen(X) +1];
          strcpy(ligne1,X);}
```

```
    char* get_province(char* X)
           {X= new char [strlen(province) +1];
            strcpy(X,province); return(X);}
    void set_province(char* X)
           {province = new char [strlen(X) +1];
            strcpy(province,X);}


    char* get_ville(char* X)
           {X= new char [strlen(ville) +1];
            strcpy(X,ville); return(X);}
    void set_ville(char* X)
           {ville = new char [strlen(X) +1];
            strcpy(ville,X);}

    char* get_codePostal(char* X)
           {X = new char [strlen(codePostal) +1];
            strcpy(X,codePostal); return(X);}
    void set_codePostal(char* X)
           {codePostal = new char [strlen(X) +1];
            codePostal= new char strcpy(codePostal,X);}
};
```

Chaque vue générée hérite de la super classe «View», tel que précisé dans les chapitres 5 et 6 du présent document. Le constructeur du *code de la vue* initialise le nom de la classe *vue* et la variable d'instance **oid** qui représente l'objet de base pour lequel l'instance de la *vue* est créée. Les données et les fonctions définies dans la classe sont celles de la partie **provides** du *point de vue*. Dans le cas des fonctions qui font appel aux attributs et aux fonctions de la partie **requires**, leurs appels seront remplacés par des appels de délégation vers les fonctions et les attributs correspondants dans la classe de base.

VII.3.4. Exemple du code usager

Outre les *vues*, nos outils génèrent une fonction C++ qui initialise la liste des *vues potentielles* des classes. Cette fonction est appelée «initializeAllViews» et est la première opération qui doit être exécutée dans la fonction principale de l'application. Elle fait partie des fonctions de la classe «Viewable_classe». Voici donc un exemple de code pour cette fonction ainsi que de code usager avec *vues*:

```
/*définition de la liste des vues potentielles des
 classes */
void Viewable_classe :  : initializeAllViews() {
allViews = new ViewList;
allViews->nomclasse = new char[7+1];
strcpy(allViews->nomclasse, "Employé");
liste *laliste, *preced;
laliste = new liste;
laliste->vue = new char[8 + 1];
strcpy(laliste->vue, "PEmployé");
laliste->next = preced;
preced= laliste;
laliste = new liste;
laliste->vue = new char[8 + 1];
strcpy(laliste->vue, "LEmployé");
laliste->next = preced;
preced= laliste;
allViews->listevue = preced;
allViews->next = NULL;
}
```

Voici un exemple de code usager que nos outils ont transformé :

```
/* la fonction main transformée */
int main()
{
// La première instruction à exécuter est
//l'initialisation des vues potentielles. C'est une
//fonction static de Viewable_classe
Viewable_classe :  : initializeAllViews();
Employé* p = new Employé;
p->attach("vue1");
char *var;
cout << "Entrez le poste de l'employé \n";
cin >> var;
p->_comb_view->print(); // print de la vue est invoqué
// le code usager initial était p->print();
p->set_poste(var);
if (strcmp(p->get_poste(),"directeur")
   {p->detach(vue1);
   p->_comb_view->print(); // la vue est détachée
   // alors print de la classe de base
   }

}
```

Nous avons montré, dans cet exemple de code, comment l'usager peut manipuler les vues avec les fonctions que nous avons prédéfini. Nous avons aussi spécifié la manière dont le code usager est transformé pour faire des appels de délégation vers les bonnes versions des fonctions. Cela se fait par l'intermédiaire de la *vue* qui définit les fonctions qui ne sont pas définies simultanément dans plusieurs *vues*, ou qui sont définies dans la classe de base et dans une ou plusieurs *vues*.

VII.4. Discussion

Après avoir présenté un exemple simple de programmation avec *vues*, nous procédons à l'évaluation de ce mode de programmation en présentant ses avantages et ses inconvénients d'un point de vue implantation.

Nous pouvons résumer le travail d'implantation par i) l'implantation des *points de vues*, ii) l'implantation des *classes de base*, iii) la génération des *vues* et iv) la transformation du code usager. Chacune de ces tâches est accomplie grâce à un ensemble d'outils que nous avons développés et qui ont rendu la programmation par *vues* transparente à l'usager et semblable à la programmation traditionnelle. Ainsi, le passage à ce nouveau mode de programmation se fait sans que trop de changements n'aient besoin d'être apportés, et sans faire appel à l'introduction de nombreux nouveaux éléments.

Ce qui suit nous permettra d'évaluer le temps consommé pour le développement par *vues*, le nombre de lignes de code ajouté et l'espace mémoire utilisé en regard de la programmation traditionnelle.

i) **Temps de développement** - Il faut, à cet égard, compter le temps nécessaire pour effectuer la conception et la définition des *points de vue* et le temps requis à la génération du code C++ final.

- Le développement des *points de vue* est une tâche indépendante des applications qui les utilisent. Cependant, ils doivent exister avant de procéder au développement de l'application. Nous croyons que le temps mis pour le développement des *points de vue* est négligeable par rapport aux gains qui peuvent découler de leur réutilisation.

- La génération du code C++ est le résultat du travail de nos outils de support des *vues*. C'est un travail de pré-traitement qui consiste à convertir un code avec *vues* vers un code C++, facilitant ainsi la tâche du développeur. Le temps qu'il faut à ce processus se compare à celui de la compilation, ce qui est raisonnable au chapitre des gains, et à celui de l'automatisation, ce qui constitut un élément fort important. Tout comme dans le cas du nombre de *points de vue* et de *vues* manipulées, plus nous avons de *points de vue* et de *vues*, plus le travail est long.

ii) **Lignes de code** - Nous parlons des lignes de code de l'application et de celles qui sont ajoutées en guise d'outils de support des *vues* :

- La programmation par *vues* engendre moins de redondance dans les définitions des classes, ce, grâce à l'utilisation des *points de vues.* Ces derniers permettent la séparation entre les caractéristiques fonctionnelles et les caractéristiques de base. Ainsi, comme nous l'avons spécifié au début, les classes n'ont pas à être définies plusieurs fois pour nombre d'utilisations. Ce sont les *points de vue* qui leur seront appliqués à mesure des besoins du développement. L'on ajoute cependant certaines lignes de code pour supporter la manipulation des *vues*, ajouts qui se concrétisent par la modification des classes pour hériter de la super-classe «viewable_classe», l'introduction des nouvelles classes avec les fonctions d'attachement-détachement et d'activation-désactivation, la définition de nouvelles fonctions, telles que la fonction d'initialisation des *vues potentielles*, et la définition de la classe de combinaison pour chaque classe.
- Les lignes de code ajoutées aux fins du développement des outils de support des *vues* touchent le changement de la grammaire C++ et la génération des structures de données internes, et seront utilisées pour générer le code C++. Leur nombre peut s'élever à quelques milliers, ce qui n'a pas facilité notre travail d'implantation. En fait, nous avions à manipuler quelques centaines de règles de grammaire pour introduire celles que nous avons développées.

iii) **Espace mémoire utilisé** - Nous avons choisi, au moment de l'implantation, de représenter les aspects fonctionnels par des instances de *vues* attachées aux objets de base. Ce processus consiste à faire appel à plusieurs objets physiques qui

représentent un seul objet réel. Chaque instance de *vue* regroupe ses propres attributs et fonctions qui, selon la programmation traditionnelle, devaient être présents dans l'*objet*, accompagnés des attributs et des fonctions des autres *vues*. Nous pouvons ainsi conclure que l'espace mémoire utilisé pour supporter la programmation par *vues* est plus ou moins le même que celui utilisé lors de la programmation traditionnelle. La programmation par *vues* nous permet de n'avoir en mémoire que les attributs et les fonctions nécessaires à l'application courante. Les attributs et les fonctions relatives aux *vues* non-attachées n'occupent pas l'espace mémoire et ne sont pas accessibles, ce qui offre, outre plus d'espace mémoire, une sécurité en ce qui a trait aux données.

VII.5. Comparaison avec d'autres approches

Dans le paragraphe précédent, nous avons évalué notre travail d'implantation en regard d'une implantation traditionnelle. Notre approche comporte cependant certains avantages par rapport à une programmation traditionnelle. Ils ont trait aux nouvelles approches de développement qui facilitent la décentralisation du développement, l'intégration des applications et la réutilisation des composants. Notons les exemples de la programmation *orientée-sujet* et de la programmation *orientée-aspect*.

VII.5.1. Critères de comparaison

Nous avons choisi certains critères de comparaison que nous avons jugés importants et qui distinguent les nouvelles approches de programmation [Bend 00]. La réutilisation des aspects fonctionnels, la dynamicité de la composition de ces aspects et la facilité de l'application des approches sont des critères importants qui influencent la qualité du développement d'un système logiciel. Nous faisons ci-après une brève description de chacun des critères de comparaison choisis :

a) Le Degré de réutilisation : Ce critère permet de voir si les aspects fonctionnels manipulés dans chaque approche sont réutilisables. Il revête une grande importance en raison du fait qu'il amène à accroître la productivité et nous permet de réaliser des systèmes logiciels de meilleure qualité.

b) La dynamicité de la composition : Ce critère permet de montrer à quel stade de développement d'un système logiciel se fait l'intégration des aspects fonctionnels. Il est très important que cette intégration soit effectuée tard dans le cycle de développement. Nous pouvons ainsi nous assurer plus de liberté et de dynamicité dans la manipulation.

c) La facilité de mise en oeuvre : Ce critère permet d'évaluer l'ensemble des contraintes et des ajouts des concepts que chaque approche impose. Il est important que les nouvelles méthodes d'implantation se rapprochent des méthodes conventionnelles. Sinon, leur utilisation sera limitée.

VII.5.2. Comparaison

Dans l'exemple qui apparaît ci-dessus, nous avons présenté deux scénarios de programmation. Le premier est traditionnel et ne fait pas appel à la notion de *vues*. Nous avons démontré que dans ce scénario, la réutilisation se fait au niveau des classes déjà définies par la relation d'héritage, mais que le contrôle d'accès et la sécurité des données n'y sont pas respectés. L'application de la relation d'héritage suppose qu'on possède le code des deux applications et que les noms des classes n'engendrent aucun conflit, ce qui suppose une collaboration dans le développement des applications.

Dans le deuxième scénario, nous avons utilisé le support des *vues* pour offrir des comportements multiples des mêmes classes. Nous y avons défini des *points de vue* et des *classes de base* et avons, par la suite, généré les *vues*. Le contrôle d'accès et l'attachement dynamique des comportements y sont offerts. Ce travail pouvait se faire par la programmation *orientée-sujet*. Nous pouvons, en effet, supposer que ces deux applications avaient déjà été développées. Leur composition consiste à générer de nouvelles classes, lesquelles renferment les données et les fonctions relatives aux deux applications (sujets). En ce qui a trait à la programmation *orientée-aspect,* nous pouvons, comme pour les *points de vue*, définir les aspects et les intégrer aux classes. Une réutilisation des aspects est possible pour d'autres classes et d'autres applications; mais comme dans le cas de la programmation *orientée-sujet*, une fois une classe dérivée, elle regroupe toutes les caractéristiques.

Nous allons maintenant discuter de chacune de ces approches en se basant sur les critères de réutilisation, de dynamicité de la composition et de facilité de mise en oeuvre.

VII.5.2.1. Degré de réutilisation

La réutilisation est un critère important qui permet de mettre en relief le paradigme de développement. Elle permet d'accroître la productivité et de diminuer le coût de développement des systèmes logiciels. Les nouvelles approches de programmation nous permettent de développer des aspects fonctionnels et de les composer. Dans le cas de la programmation par *sujets*, les aspects fonctionnels sont en effet représentés par des *sujets* développés séparément qui sont, par la suite, composés pour générer de nouveaux *sujets*. Chaque *sujet* est spécifique et auto-satisfaisant, regroupe un ensemble de classes bien définies et ne peut pas être réutilisé pour plusieurs applications mais simplement composé avec d'autres sujets. Dans les cas de programmation par *vues* et par *aspects*, nous faisons appel à de nouveaux concepts de *points de vue* et d'*aspects*. Ces derniers définissent un aspect particulier qui ne fait pas partie d'une application particulière. Ils sont génériques et s'appliquent à plusieurs classes.

VII.5.2.2. Dynamicité de la composition

Il nous faut un mécanisme qui permet aux objets de changer leur comportement au cours de leur durée de vie, surtout lorsque ces changements s'opèrent dans le contexte d'une même application. Un tel changement dynamique de comportement est toutefois problématique car les programmations *orientée-sujet* et *orientée-aspect* intègrent plusieurs domaines fonctionnels au moment du codage ou au moment de la compilation. Précisons que ce comportement devient statique au moment de l'exécution.

Pour offrir le changement dynamique de comportement, la vérification des types doit être retardée jusqu'au moment de l'exécution, où on sait quel est le comportement qui est disponible.

Nous avons voulu résoudre ce problème en déléguant à la *vue de combinaison* la vérification de comportement qui s'attarde aux *vues* attachées et actives. Nous avons utilisé une liste de *vues* potentiellement applicables aux classes qui nous permettent de

retarder la vérification du type jusqu'au moment de l'exécution. Avec la programmation *orientée-sujet*, la composition dynamique ne peut être faite que si nous maintenons l'ensemble des sujets avant et après la composition. Ceci mène à une explosion combinatoire du nombre de sujets lors de grandes applications. De même, pour la programmation *orientée-aspect*, le «tissage» dynamique des *aspects* n'est pas possible, car une fois l'*aspect* intégré, la démarche arrière n'est plus réalisable.

Dans la figure 7.8, nous représentons le cycle de développement d'un logiciel, depuis l'analyse jusqu'à l'exécution, et nous montrons le point de composition pour chacune des approches.

Cette figure nous présente deux aspects, soit les aspects A et B. Ces derniers sont identifiés au moment de l'analyse et peuvent être composés si nous appliquons le développement par modèle de *rôle* (voir chapitre 2). Au niveau de la conception, la composition des *sujets* et des *aspects* peut être faite en fonction des approches *orientée-sujet* et *orientée-aspect*. Cette composition génère un nouvel aspect (aspect A_B, A et B) à partir duquel les aspects d'origine ne sont pas identifiables. La programmation par *vues* nous permet d'intégrer et de manipuler les *vues* dynamiquement lors de l'exécution. Nous n'avons cependant pas pu éviter certains problèmes reliés à la dynamicité des *vues,* problèmes reliés au *"type safety"*. En permettant qu'un objet puisse présenter des comportements divers, l'usager peut manipuler toutes les fonctions de cet objet sans avoir à prendre en considération les *vues* attachées et actives de cet objet. Ceci ne peut être vérifié qu'au moment de l'exécution.

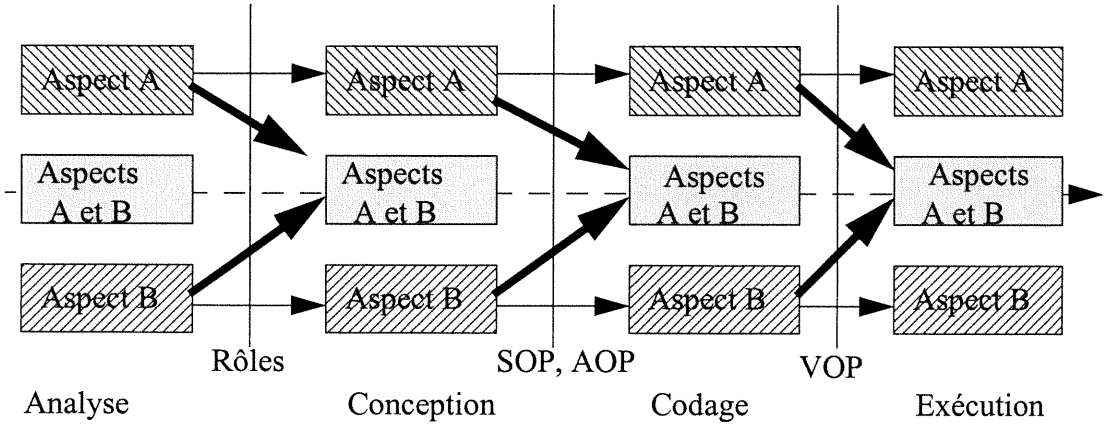


Figure 7.8 : Dynamicité de la composition

VII.5.2.3. Facilité de mise en oeuvre

Pour chacune des approches, de nouveaux concepts et de nouveaux outils restent à être définis et utilisés. Ces ajouts doivent être minimisés pour faciliter l'application des approches, car l'usager ne doit pas avoir à s'occuper des détails de la composition et à beaucoup changer ses habitudes de développement.

Pour la programmation *orientée-sujet* à partir de *sujets* déjà définis, l'outil génère un nouveau *sujet* en se basant sur des règles de composition prédéfinies. Cette composition présente des limites quant à la nature des *sujets* composés. Ces derniers doivent manipuler les mêmes classes, sinon, la composition ne peut pas générer de nouveaux *sujets*.

Pour la programmation *orientée-aspect*, des *aspects* sont à définir en respectant certaines règles de grammaire. Cette définition ressemble à la définition des classes comportant certaines annotations (**advice, after, before**) et spécifiant où et comment les propriétés de la classe ont été modifiées. Un outil est développé pour assurer l'intégration des *aspects* dans les classes, car lors de l'ajout à une classe de nouveaux attributs et de nouvelles méthodes, l'on ne prend pas en considération les attributs et les fonctions qui portent les noms déjà existants. Il n'y a pas de règles de composition, mais l'on assiste plutôt à une redéfinition automatique.

Dans notre approche, une nouvelle syntaxe est ajoutée au langage de programmation pour permettre la définition des *points de vues* et la manipulation des *vues*. L'outil de développement par *vues* permet la transformation du code au moyen de cette nouvelle syntaxe, vers un code compilable. Certaines fonctionnalités sont ajoutées et peuvent être manipulées sans transformation. C'est le cas de l'utilisation des fonctions d'attachement, de détachement, d'activation et de désactivation. Notons ici que l'on fait face à certains problèmes lors de l'implantation, et que ces derniers doivent être pris en considération. Par exemple, les données d'une classe qui correspondent à la partie **requires** d'un point de vue doivent être publiques si l'on veut que cette classe effectue la délégation dans le code des classes *vues*.

VII.6. Conclusion

Les comportements dynamiques, introduits dans la programmation par *vues*, sont représentés par des entités fonctionnelles génériques appelés *points de vue* et applicables aux classes, et sont réutilisables pour divers types d'objets. Notre approche face à la programmation par *vues* se distingue principalement par le fait que les aspects fonctionnels des objets appelés *vues* puissent être ajoutés à un objet et retirés de celui-ci lors de l'exécution. Les *vues* sont implantées par des objets contenant des données et pouvant exécuter des fonctions.

La programmation par *sujets* consiste à composer des tranches compilables (des *sujets*) pouvant être exécutables à partir d'une application *orientée-objet.* En principe, la composition de *sujets* permet d'intégrer, grâce à des règles explicites, des applications dont la «collaboration» n'a pas été planifiée. En pratique, il faut observer certains édicats de programmation, voire même préplanifier la composition.

Dans la programmation par *aspects*, un *aspect* représente une facette fonctionnelle de l'entité d'un programme. Certains aspects répondent à des préoccupations spécifiques qui s'appliquent à plusieurs opérations ou à plusieurs classes, et ce, indépendamment les unes des autres, notamment la production d'une trace des appels. D'autres peuvent impliquer conjointement plusieurs classes ou opérations pour réaliser une tâche commune, soit le maintien d'une contrainte entre plusieurs objets. Malgré l'absence de modèle formel de spécification et de réutilisation d'aspects, et grâce à sa simplicité, la programmation par *aspects* a suscité beaucoup d'intérêt dans la communauté. Tout comme la composition de *sujets*, l'introduction ou l'«injection» d'*aspects* se fait au moment de la compilation. Par conséquent, lors de l'exécution, tous les *aspects* cohabitent dans les classes.

Chapitre VIII

Conclusion

Dans ce travail, nous avons abordé le problème de comportement dynamique des objets dans une même application. Nous avons proposé une approche de développement qui permet une intégration facile des composants développés indépendamment et une diversité dans les rôles que les objets peuvent avoir. Une nouvelle technique de programmation a été proposée. C'est la programmation par *vues* qui nous permet de manipuler les objets au moyen de plusieurs *vues* représentant différents comportements. Elle répond à un besoin de plus en plus marqué dans l'industrie du logiciel : le besoin de supporter plusieurs domaines fonctionnels dans le cadre d'une même application, sachant que ces domaines peuvent avoir été développés de façon indépendante et que différents clients veuillent différentes combinaisons de fonctionnalités. Dans ce qui suit, nous allons présenter les apports de notre travail ainsi que ses directions futures.

VIII.1. Contributions

Nouvelle méthodologie de développement

Nous avons proposé une nouvelle méthodologie de développement basée sur l'utilisation d'aspects fonctionnels génériques et sur la manipulation des objets avec des comportements qui varient dynamiquement. Nous avons appelé les aspects fonctionnels *points de vue*. Un *point de vue* est une entité indépendante qui peut être appliquée à plusieurs classes. Il incarne un rôle pouvant être joué par des objets de types divers. Appliqué à une classe, il génère une *vue* particulière de cette classe qui est l'interprétation de ce rôle. L'avantage des *points de vue*, à part leur réutilisation pour différentes applications et leur développement indépendant, est la séparation entre les aspects fonctionnels que les objets peuvent acquérir et perdre et les caractéristiques de base qui les identifient et les particularisent par rapport à d'autres objets.

Nous avons aussi défini les notions de *classes de base* et des classes *vues*. Les *classes de base* représentent les objets avec leurs caractéristiques de base. Les objets des classes *vues* sont la représentation physique des *vues* des objets de base. À tout moment, nous avons une séparation entre les objets *vues* et les objets de base qui nous facilite la manipulation dynamique des *vues* et qui différencie notre approche de celles qui lui sont comparables. Une liaison est maintenue entre chaque objet et ses *vues*, ce qui permet de faire des appels de délégation pour l'exécution des fonctions qui ne sont pas définies dans sa classe. D'un côté, les appels aux membres de base dans les classes *vues* sont délégués vers leur code défini dans les *classes de base.* D'un autre côté, les appels des fonctions qui ne sont pas définies dans les *classes de base* sont remplacés par des appels de délégation vers leur code défini dans les *vues*. La transformation des appels est effectuée automatiquement par nos outils de support de *vues* qui permettent de reporter les décisions concernant le type des objets et leur état jusqu'au moment de l'exécution. Chaque appel d'une fonction des vues est remplacé par un appel de délégation qui permet de faire passer la compilation sans erreur et de reporter la décision définitive à cet égard jusqu'au moment de l'exécution.

Une implantation des outils qui supportent la programmation par vues

Nous avons choisi d'implanter la programmation par *vues* de façon intuitive et peu intrusive pour le développeur, ce qui a rendu notre tâche d'autant plus complexe. Notre objectif est de faciliter l'utilisation et l'introduction des *vues* pour les nouvelles applications et pour celles qui sont déjà développées.

Pour la réalisation du travail, nous avons intégré un certain nombre de règles à celles de la grammaire du langage populaire C++. La manipulation de l'ensemble de ces règles, qui se chiffrent à plus de 350, fut très complexe et a nécessité de nombreuses lignes de code qui dépassent 10 kilos lignes de code. Par rapport au développement conventionnel, nous avons ajouté de nouvelles instructions pour la manipulation des *vues* et pour le codage des *vues* et des *points de vue*, et nous avons implanté des outils de prétraitement qui facilitent et automatisent l'analyse de ces nouvelles instructions. La programmation par *vues* est ainsi faite au détriment de nouvelles instructions à manipuler et

d'un investissement de temps nécessaire au pré-traitement. Néanmoins, il résulte de cela des avantages marqués.

VIII.2. Directions futures

Notre travail se poursuit dans plusieurs directions que nous pouvons diviser en deux volets, soit l'aspect théorique et l'aspect pratique.

Sur le plan théorique

Nous avons mentionné, dans le chapitre quatre, que la relation d'héritage entre les *points de vue* peut avoir certaines implications sur les *vues*. Comme pour le cas des classes, cette relation peut comporter certains avantages et réduire encore plus les investissements requis pour le développement des applications. Elle peut nous être utile pour développer des *points de vue* à partir de ceux qui existent déjà. Cependant, elle a des effets comme ceux de la relation d'héritage offerte entre les classes de base qui ne sont pas négligeables.

Outre l'étude de la relation d'héritage entre les entités de même type, il sera intéressant de voir certaines relations entre les entités de types différents. Par exemple, nous avons étudié le cas de la relation entre un *point de vue* et une classe à laquelle il est appliqué. Or, dans un cas plus général, nous avons le cas de la relation de plusieurs *points de vue* avec plusieurs classes. Il faut donc voir la possibilité de définir des *vues* sur des collections d'objets ou des collaborations entre objets. Ainsi, un *point de vue* ressemblera à un modèle de collaboration entre plusieurs objets, et la vue générée correspondrait à une instanciation de ce modèle pour des classes spécifiques. Cette approche ressemble un peu aux travaux de Vanhilst et Notkin sur l'implantation de rôles fonctionnels [VanH 96].

Certains autres points restent à explorer pour mieux supporter la programmation par *vues*. Particulièrement, la rétroingénierie qui permet de définir des *vues* et des *points de vue* à partir des applications déjà existantes. Ceci va permettre la migration de toutes les applications vers des applications avec *vues*. Nous avons aussi, dans notre travail,

parlé de la classe *vue de combinaison* dans laquelle sont définies les fonctions des *vues* et de la classe de base. Le code de ces fonctions est créé selon certaines règles de composition qui doivent être analysées plus à fond.

Sur le plan pratique

Nous pouvons apporter certaines améliorations à l'interface de nos outils pour faciliter leur utilisation, et couvrir plus de scénarios de développement pour arriver à une implantation davantage robuste. Ce n'est qu'avec plusieurs scénarios que l'on peut couvrir plus de cas particuliers et de détails. Citons l'exemple des membres de la partie **requires** d'un *point de vue* qui devraient être dans la partie **public** de la *classe de base*. Cette constatation a été faite suite à l'écriture d'un code qui utilise les membres de la *vue générée*. D'autres problèmes reliés aux détails de l'implantation sont aussi à élaborer, notamment celui relié à la duplication des attributs dans la classe et dans les *points de vue*, qui peut être différent de celui de la duplication des méthodes. Ce dernier a été résolu par la création d'un nouveau code dans la *vue de combinaison*.

Finalement, sur le plan pratique, notons qu'il sera intéressant de voir comment le support des *vues* peut être implanté pour le langage Java et quelles sont les particularités des différentes implantations.

Chapitre IX.

Références

[Adib 81] M. Adiba; *Derived relations: a unified mechanism for views, snapshots and distributed data;* Proceedings of Very Large Database Systems Conference (VLDB), Cannes, France, Septembre 1981.

[Ages 97] O. Agesen, S. Freund et J. Mitchell; *Adding Type Parameterization to the Java language;* Proceedings of object-oriented systems, languages, and applications (OOPSLA'97), Atlanta, Octobre 1997.

[Aksi 92] M. Aksit, L. Bergmans, et S. Vural; *An Object-Oriented Language-Database Integration Model: The Composition Filters Approach.* Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92) et Lecture Notes in Computer Science no. 615, 1992.

[Alba 85] A. Albano, L. Cardelli et R. Ornini; *Galileo : A strongly typed inheritance conceptual language;* Transaction database system, pp.230-260, Juin 1985.

[Alba 93] A. Albano, R. Bergamini, G. Ghelli et R. Orsini; *An object data model with roles;* Proceedings of the 19th international conference on Very Large Database (VLDB), pp. 39-51, San Mateo, Ca, 1993.

[Ande 92] E.P. Andersen et T. Reenskaug; *System Design by Composing Structures of Interacting Objects;* Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92), 1992.

[Bani 98] E. Baniassad et G. Murphy; *Conceptual Modules Querying for Software Reengineering;* In Proceedings of the International Conference on Software Engineering (ICSE 20), Avril 1998.

[Bato 00a] D. Batory, G. Chen, E. Robertson, et T. Wang; *Design Wizards and Visual Programming Environments for GenVoca Generators*; IEEE Transactions on Software Engineering, pp. 441-452, Mai 2000.

[Bato 00b] D. Batory, C. Johnson, B. MacDonald et D. v. Heeder; *Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study;* International Conference on Software Reuse, Vienna, Austria, 2000.

[Bell 98] Z. Bellahsène et R. Ducournau; *Vue en base de données et points de vue en représentation des connaissances;* L'objet. Vol. 4 n.3, pp. 307-331, Septembre 1998.

[Bend 00] S. Bendelloul, H. Mili, J. Dargham et H. Mcheick; *A comparaison of view programming, aspect oriented programming and subject oriented programming from a reuse perspective*; ICSSAE, Paris France, Decembre 2000.

[Bert 92] E. Bertino; *A view mechanism for Object-Oriented Database;* Proceedings of the Third international conference on extending database technology (EDBT'92), pp. 136-151, Vienna, Austria, Mars 1992.

[Biel 93] R. Bielak et J. C. McKim; *The many faces of a class : Views and Contracts*; Proceedings of Tools USA' 93, pp. 153-161, 1993 .

[Blak 87] E. Blake et S. Cook; *On including part hierarchies in object-oriented languages, with an implementation in Smalltalk*; Proceedings of European Conference on Object-Oriented Programming (ECOOP'87), pp. 45-54, 1987 .

[Bobr 87] D.G. Bobrow et M.S. Stefik; *The LOOPS manual;* Report of XEROX Palo Alto Research Center, 1987.

[Booc 86] G. Booch; *Object-Oriented Development;* IEEE Transaction on Software Engineering, 1986.

[Bros 88] V. Brosda et G. Vossen; *Update and retrieval in a relational database through a universal schema interface;* ACM transactions on data base systems, Vol. 13, no. 4 , pp. 449-485, Decembre 1988.

[Card 84] L. Cardelli; *A semantics of multiple inheritance, In semantics of data types;* Lecture notes in computer science, vol. 173, pp. 51-67, New-York, 1984.

[Card 85] L. Cardelli et P. Wegner; *On understanding types, data abstraction and polymorphism;* Computing Surveys, vol. 17, no. 4, Decembre 1985.

[Card 88] L. Cardelli; *Types for data-Oriented languages;* Lecture Notes in computer Science, vol. 303, 1988.

[Carr 89] B. Carré; *Méthodologie oo pour la représentation des connaissances, concepts de points de vue, de représentations multiples et évolutive d'objets*; Thèse de doctorat, Université de Lille, France 1989.

[Carr 90] B. Carré et J.M. Geib; *The point of view notion for multiple inheritance;* Proceedings of object-oriented systems, languages, and applications (OOPSLA'90), pp. 312-321, Octobre 1990.

[Case 93] Y. Caseau; *Efficient handling of multiple inheritance hierarchies;* Proceedings of object-oriented systems, languages, and applications (OOPSLA'93), pp. 271-287, Octobre 1993.

[Clar 99] S. Clarke, W. Harrison, H. Ossher et P. Tarr; *Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code*; ACM SIGPLAN Notices; Proceedings of object-oriented systems, languages, and applications (OOPSLA'99), pp. 325-339, Octobre 1999.

[Clay 85] B. Claybrook, A. Claybrook et J. Williams; *Defining Database views as data abstractions;* IEEE transactions on software Engineering, Vol. 11, no.1, Janvier 1985.

[Coul 96] B. Coulange; *Méthodes informatiques et pratiques des systèmes: Réutilisation du logiciel*; Collection coordonnée par A. Champenois, Edition Masson, Paris, 1996.

[Czar 00] K. Czarnecki et U. W. Eisenecker; *Generative Programming: Methods, Tools, and Applications;* Addison-Wesley, Reading, MA, Juin 2000.

[Darg 96] J. Dargham et H. Mili; *Viewpoints: a generic concept for viewing;* Worshop on subject oriented programming of the conference of object-oriented systems, languages, and applications (OOPSLA'96), Octobre 1996.

[Darg 97] J. Dargham et H. Mili; *Feature interaction through object-oriented view integration;* Document interne du projet IGLOO, Juin 1997.

[Date 90] C. J. Date; *An introduction to Database systems*; Volume I, 1990, cinquième édition.

[Davi 87] H. Davis; *VIEWS: Multiple perspectives and structured Objects in a knowledge representation language*; Bachelor and Master of science thesis, MIT 1987.

[Debr 97] L. Debrauwer, G. Vanwormhoudt et B. Carré; *Un cadre de conception par contextes fonctionnels de systèmes d'information à Objets*; Actes du Congrès INFORSID97, Toulouse, 1997

[Dekk 92] L. Dekker et B. Carré; *Multiple and dynamic representation of frames with points of view in FROME;* Proceedings of Representation par Objets, La Grande Motte, pp. 97-111, Juin 1992.

[Delo 92] C. Delobel, C. Lecluse et P. Richard; *Bases de données : des systèmes relationnels aux sytèmes à objets*; Proceedings of Représentation par Objets, La Grande Motte, Juin 1992.

[Duco 89] R. Ducournau et M. Habib; *La muptiplicité de l'héritage dans les langages à objets;* Technique et Science Informatiques, vol. 8 pp. 40-62, 1989.

[Ferb 89] J. Ferber; *Objets et Agents: une étude des structures de représentation et de communications en iIntelligence Artificielle*; Thèse d'Etat, Université Pierre et Marie Curie, Paris 1989.

[Gamm 95] E. Gamma, R. DHelm, R. Johnson et J. Vlissides; *Design Patterns, Elements of Reusable Object Oriented Software;* Addison-Wesley, 1995.

[Garl 87] D. Garlan; *Views tools in integrated environments;* Proceedings of Tools'87, 1987.

[Gold 81] I. Goldstein et D. Bobrow; *An experimental description-based programming environment : Four reports;* Xerox Palo Alto Research center, Rapport technique CSL-81-3, Mars 1981.

[Gott 88] Gottlob, Paolini et Zicari; *Properties and update semantics of consistent views;* ACM transactions on data base systems, Vol. 13, no. 4 pp. 486-524 Decembre 1988.

[Hail 90] B. Hailpern et H. Ossher; *Extending Objects to support multiple interfaces and access control;* IEEE Transaction on software engineering, Vol. 16, no. 11, Novembre 1990.

[Harr 93] W. Harrison et H. Ossher; *Subject-Oriented Programming: A critique of pure objects;* Proceedings of object-oriented systems, languages, and applications (OOPSLA'93), pp. 411-428, Washington D.C. 1993.

[Haut 86] A. Hautamäki; *Points of view and their logical analysis;* Acta Philosophica Fennica 41, 3-126 1986.

[Heil 90] S. Heiler et S. Zdonik; *Object Views : Extending the vision;* IEEE data engineering conference, pp. 86-93, Los Angelos 1990.

[Hoyd 93] G. M. Hoydalsvik et J. H. Holn; *Dynamic modeling in OOram*; Proceedings of object-oriented systems, languages, and applications (OOPSLA'93), Washington D.C. 1993.

[Kais 87] G. Kaiser et D. Garlan; *Melding data flow and Object-Oriented programming*; Proceedings of object-oriented systems, languages, and applications (OOPSLA'87), pp. 254-267, Orlando, FL, October, 1987. Proceedings published as SIGPLAN Notices, 22(12), Decembre 1987.

[Ken 99] E. Kendall; *Role Model Designs and Implementations with Aspect-Oriented Programming;* ACM SIGPLAN Notices; Proceedings of object-oriented systems, languages, and applications (OOPSLA'99), pp. 353-369, Octobre 1999.

[Kicz 97] G. Kiczales et al.; *Aspect-Oriented Programming;* Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, Spring-Verlag LNCS 1241, Juin 1997.

[Kicz 99] G. Kiczales et C. Lopez; *Modularization revisited: aspects in the design and evolution of software systems;* TOOLS'99, USA, 1999. www.tools.com/usa.

[Kris 96] B. B. Kristensen et K. Osterbye; *Roles: Conceptual Abstraction Theory & Practical Language Issues;* Theory and Practice of objects systems, 1996.

[Lieb 96] K. Lieberherr; *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns;* PWS Publishing Company, Boston, 1996.

[Maid 82] A.S. Maida et S.C. Shapiro; *Intensional concepts in propositional semantic networks;* Cognitive science; pp. 291-330, 1982.

[Male 95] J. Malenfant; *On the semantic diversity of Delegation-Based Programming Languages;* Proceedings of object-oriented systems, languages, and applications (OOPSLA'95), pp. 215-230, Octobre 1995.

[Marc 93a] S. Marcaillou, B. Coulette et D. P. Vo; *An approach to view point modelling;* Tools Europe'93 Versailles(F), 08-11 Mars 1993.

[Marc 93b] S. Marcaillou et B. Coulette; *Intégration de la notion de point de vue dans la modélisation par objets*; Revue ICO Québec, Automne 1993.

[Marc 94] S. Marcaillou, A. Kriouile et B. Coulette; *VBOOL, une extension d'Eiffel intégrant le concept de point de vue*; Third Maghrebian Conference on Software Engineering and Artificial Intelligence, MCSEAI'94, pp. 115-125, Rabat 11-14 Avril 1994.

[Mart 84] W.A. Martin; *Descriptions and the specialisation of concepts;* In Artificial Intelligence: An MIT Perspective, Vol. 1, MIT Press, pp. 375-419, 1984.

[Mart 92] J. Martin and J. Odell; *In Object-Oriented Analysis and Design;* Prentice-Hall, 1992.

[McFa 88] F. McFadden et J. Hoffer; *Database management;* Second edition 1988.

[Meye 88] B. Meyer; *Object-Oriented Software Construction;* Prentice-Hall International Edition, 1988.

[Mili 96] H. Mili, W. Harrison et H. Ossher; *Implementing Subject-Oriented Programming in Smalltalk*; Proceedings de la conférence TOOLS USA'96, Santa Barbara, CA, Août 1996.

[Mili 99] H. Mili, J. Dargham et A. Mili; *View programming: toward a Framework for decentralized development and execution of oo programs*; TOOLS'99, Août 1999.

[Mili 99a] H. Mili, J. Dargham et S. Bendelloul; Separation of concerns and typing: A first stab; workshop MDSOC at the conference of object-oriented systems, languages, and applications (OOPSLA'99), Novembre 1999.

[Mili 01] H. Mili, H. Mcheick, J. Dargham et S. Bendelloul; Distribution d'objets avec vues; Actes de la conférence Langages et Modèles à Objets (LMO'01) France, Janvier 2001.

[Myer 97] A. Myers, J. Bank et B. Liskov; *Parameterized Types for Java;* Symposium on Principles of Programming Languages (POPL'97), the 24th ACM SIGPLAN-SIGACT, Paris, Janvier 1997.

[Naja 95] H. Naja et N Mouaddib; *Un modèle pour la représentation multiple dans les bases de données orientées objets*; Actes de la conférence Langage et Modèles Objet (LMO), Nancy, 1995.

[Ngu 91] A. Ngu, L. Wong et S. Widjojo; *On Canonical and non-canonical classification*; Second international conference on deductive and object oriented data bases, Munich, Germany, pp. 371-390, Decembre 1991.

[Nguy 92] G. T. Nguyen et D. RIeu; *Multiple object representations*; 20th. ACM Computer Science Conference; Kansas City (Mo), Mars 1992.

[Nuse 94] B. Nuseibeh, J. Kramer et A. Finkelstein; *A framework for expressing the relationships between multiple views in requirements specification*; IEEE transactions on software engineering, vol, 20, no. 10 Octobre 1994.

[Oder 97] M. Odersky et P. Wadler; *Pizza into Java: Translating theory into practice;* Symposium on Principles of Programming languages (POPL'97), the 24th ACM SIGPLAN-SIGACT, Paris, Janvier 1997.

[Ossh 95] H. Ossher, M. Kaplan, W. Harrison, A. Katz et V. Kruskal; *Subject-oriented composition rules;* Proceedings of object-oriented systems, languages, and applications (OOPSLA'95), pp 235-250, Austin, TX, Octobre 1995.

[Pals 90] J. Palsberg et M. Schwartzbach; *Type substitution for Object-Oriented Programming;* Proceedings of object-oriented systems, languages, and applications (OOPSLA'90) et European Conference on Object-Oriented Programming(ECOOP'90), pp.151-160, Ottawa 1990.

[Pern 90] B. Pernici; *Objects with Roles*; in the Proceedings of the conference on Office Information Systems, SIGOIS Bulletin, Vol. 11, Issues 2,3,pp. 205 215, ACM Press, 1990.

[Reh 90] F. Rechenmann, O. Mrino et P. Uvietta; *Multiples perspectivesand classification mechanism in Object Representation*; Proceedings of ECAI Conference, Juillet 1990.

[Reen 96] T. Reenskaug, A. Wold et A. Lehne; *Working with objects; The OORAM Software Engineering Method;* par Manning Publications Co, 1996.

[Reis 84] S. Reiss; *A Graphical Program Development with PECAN Program Development Systems*; Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (Pittsburgh), pp. 30 -41, Avril, 1984.

[Rica 90] C. Ricardo; *Database systems : Principles, Design, and Implementation*; 1990.

[Rich 91] J. Richardson et P. Schwarz; *Aspects : Extending Objects to support Multiple independent roles*; Proceedings of the ACM SIGMOD conference, pp. 298-307, Denver, Co, Mai 1991.

[Rieh 98] D. Riehle et T. Gross; *Role Model Based Framework Design and integration*, Proceedings of object-oriented systems, languages, and applications (OOPSLA'98), pp. 117-133, 1998.

[Rieu 92] D. Rieu, G. Nguyen, J. Escamila; M*éthodes et représentation de connaissances*; Proceedings de Representation par objets, pp. 17-29, La grande Motte, Juin 1992.

[Scha 86] C. Schaffert, T. Cooper, B. Bullis, M. Kiliane et C. Wilpolt; *An introduction to Trellis/OWL; ACM conference on oo systems, languages and applications*; Proceedings published as SIGPLAN Notices 21(11) nov. 86, Portland, 1986.

[Shil 89] J. Shilling et P. Sweeney; *Three Steps to views : Extending the Object-Oriented Paradigm;* Proceedings of object-oriented systems, languages, and applications (OOPSLA'89), pp. 353-361, 1989

[Snyd 86] A. Snyder; *Encapsulation and inheritance in oo programming languages*; Proceedings of object-oriented systems, languages, and applications (OOPSLA'86), pp. 38-45, Portland Oregon, ACM, Septembre 1986.

[Stef 86] M. Stefik et D.G. Bobrow; *Object-Oriented : Themes and variations;* IA magazine vol. 4, no. 4, 1986.

[Stor 88] V. Storey et R. Goldestein; *A Methodology for Creating User Views in Database Design;* ACM transactions on data base systems, Vol. 13, no. 3, , pp. 305-338, Septembre 1988.

[Stro 97] B. Stroustrup; *The C++ Programming Languages;* Third edition, Addison Wesley 1997.

[Reen 96] T. Reenskaug, P. Wold et O. A. Lehne; *Working with objects*; Greenwich: Manning, 1996.

[Tsic 78] D. Tsichritzis and A. Klug; *The ANSI/X3/SPARC DBMS framework*; Information System, vol.3, 1978.

[Ullm 82] J. Ullman; *Principles of Database Systems*; Second edition 1982.

[VanH 96] M. VanHilst et D. Notkin; *Using Role Components to Implement Collaboration-Based Designs;* Proceedings of object-oriented systems, languages, and applications (OOPSLA'96), ACM Press, San José, Cal., pp. 359-369, Octobre 1996.

[Velh 93] A. Velho et R. Carapuca; *Modelling object behavior with Roles;* Proceedings of workshop on specification of behavioral semantics in object-orinted information modelling, Septembre 1993.

[Way 87] E.C. Way; *The nature and structure of semantic hierarchies, In Dynamic type hierarchies: An approach to knowldge representation through metaphor*; Doctoral Dissertation, Department of Philosophy, SUNY at Binghamton, pp. 103-138, 1987.

ANNEXE

Annexe A

Quelques lignes de code de l'analyseur Yacc

```
%{
/*********/
#define YYDEBUG_LEXER_TEXT (yylval) /* our lexer loads this up each time.
We are telling the graphical debugger where to find the spelling of the tokens.*/
#define YYDEBUG 1       /* get the pretty debugging code to compile*/
#define TABLE_SIZE 1024  /* Number of entries in the symbol table */
#define TEXTMAX 2000

#include <stdio.h>
#include "symtab.h"
#include "parseTree.h"
#include "bloc.h"
#include "viewpoint.h"

int laposition; /* utilise pour repositionner yyin une fois on ecrit la classe*/
FILE * resultat; /* variable pour le copiage direct *****/
int indicee; /* variable pour parcourir le fichier d'entree yyin pour copier au fur et a mesure */

int traite=0; /*flag indiquant qu'on a mis viewable_classe pour la classe qu'on copie dans
                resultat */
int ch;    /* utiliser pour le copiage avec getc et putc, cette variable doit etre de type int */
char ch_c[10];
int nombre=0;

/*************************************************************/
/***************** variables pour les vues ****************/
/********** correspondance de variables **********/

ptListeCorresp listeCorresp = NULL; /* liste contenant les parametres saisis de
                        correspondance de variables de classe et PV.*/

ptHistoryCorresp historyCorresp = NULL; /* Liste de correspondance entre classe et vue.
                        Construite lors de la definiton de la vue */

ptMapHistory maphistory = NULL;  /* liste des correspondances entre classes PV et vues */

/*************************************************************/
/********* variables pour la correspondance de methodes ******/

ptMethodparam param_corresp = NULL;     /* liste des parametres de correspondance */
ptMethodandparam methodcorresporig = NULL;   /* methode origine de correspondance */
ptMethodandparam methodcorrespdestin = NULL;  /* methode destin de correspondance */
ptMethodandparam methodcorrespnew = NULL;   /* la methode new de correspondance */
ptMethodCorresp listmethodeCorresp = NULL; /* liste des methodes de correspondance*/
ptclassMethod historymethodeCorresp = NULL; /*Correspondance de methode construite */
```

```
/***************variables des viewpoints ********/

ptlistOfVP requiredList = NULL;
ptViewpoint ptrListViewpoint = NULL;     /* liste contenant la liste des PV */
ptView  ptrListView =NULL;               /* liste contenant la liste des vues */
ptViewpoint currentViewpoint = NULL;   /* le point de vue courant *****/
myptMethod currentVPMethod = NULL;
ptVar currentVPVariable = NULL;
ptVar      currentVPParameter = NULL;
ptVar currentVar;        /* variable courante du point de vue cree par createVar */
ptVar      currentTestVariable = NULL;


int vpFlag=0;          /* flag pour dire si on etudie un point de vue ou non,
                       pour l'instant currentviewpoint le remplace  ******/
int require=0;        /* variable indiquant qu'on est dans requires ***/
int required=0;       /* variable indiquant qu'on est dans required ***/
int provided=0;       /* variable indiquant qu'on est dans provided ***/

char *currenttype = "";
char *methodtype = "";
int beginingDefinition = 0;         /*** variable indiquant qu'on est a l'interrieur
                                 de la definition d'une methode ********/
char comment[TEXTMAX];
int i = 0;
int lineno;
long charPos = 0;
long startClass;
long startDecl;
int methOrVar;
char *filename;
char * headfile;
int varType = 1;
int currentBloc =0;
ptClass ptrNewClass, ptr;
ptClass ptrBeginOfClass = NULL;
ptClass firstClass = NULL;   /* la super Classe de toutes les classes */
ptClass ptrEndOfClass = NULL;
ptClass ptrCurrentClass = NULL;
ptDataMem nodeOfDataVar = NULL;
ptCall ptStructCall = NULL;
char * varIdent;
int isaDestructor = 0;
char * typeAfterName;
char * ptrView = NULL;
int isDeclarationMethOrVar=2;   /* it is 0 if there is a variable and 1 for method, it is used to
                                   distinguish between variable and method */
int openBloc = 0;             /* used to get the parameters of method as variable
                                                                  of its bloc (potree) */
int isCallMethOrVar = 1;       /* to distinguish between variable call and function call */
char *nameMethod;
char * newInstance = NULL;
```

```
ptMethod temporaryMethod=0;
int yesScope = 1;
char * typeSpecifier;
ptTemplate typeListTemp = NULL;
%}

/*************************************************************************/
%union { struct symtab *symp; char  *value; struct treeNode *nodePt; }

%token BREAK        ELSE    SWITCH                      CONTINUE
%token CASE         ENUM    DO                          IF
%token  RETURN      WHILE   FOR                         DEFAULT
%token  GOTO    SIZEOF

%token DEFINE OPDEFINED ELIF ENDIF ERROR IFDEF IFNDEF INCLUDE
%token LINE PRAGMA UNDEF
%token  NEW DELETE
%token  THIS
%token  OPERATOR

/* Group by basic_type_name */
%token INT      CHAR                        SHORT                       LONG
%token  FLOATDOUBLE                         SIGNED                      UNSIGNED
%token  VOID   CONST                        VOLATILE

/* Group by storage_class */
%token  EXTERNTYPEDEF                       STATIC                      AUTO
%token  REGISTERFRIEND                      OVERLOAD    INLINE
%token  VIRTUAL

/* Group by aggregate_key */
%token  CLASSSTRUCT                         UNION                       TEMPLATE

/********** Group by aggregate_key for viewpoints **************/
%token VIEWPOINT     REQUIRES  PROVIDES  REQUIRED PROVIDED
%token AS VIEWDEF

/*****************************************************************/
/* Group by access_specifier */
%token  PUBLIC        PROTECTED      PRIVATE

/* ANSI C Grammar suggestions */
%token <symp> STRINGliteral
%token <symp> FLOATINGconstant      INTEGERconstant      CHARACTERconstant
%token <symp> OCTALconstant         HEXconstant

/* New Lexical element, whereas ANSI C suggested non-terminal */
%token <symp> TYPEDEFname
%token  <symp> IDENTIFIER
```

```
/* Multi-Character operators */
%token  ARROW           /*   ->                      */
%token  ICR DECR        /*   ++     --               */
%token  LS RS           /*   <<     >>               */
%token  LE GE EQ NE     /*   <=     >=     ==     !=     */
%token  ANDAND OROR     /*   &&     ||               */
%token  ELLIPSIS        /*   ...                     */
%token SEMICOLON COLON
%token LP RP COMMA LC RC LB DOT AND STAR PLUS MINUS NEGATE NOT DIV MOD
%token LT GT XOR PIPE QUESTION ASSIGN
%token ANDAND OROR MULTassign DIVassign MODassign PLUSassign
%token MINUSassign LSassign RSassign ANDassign ERassign ORassign ELLIPSIS
            /* Following are used in C++, not ANSI C       */
%token  CLCL            /*   ::                      */
%token  DOTstar ARROWstar/*   .*      ->*             */

/* modifying assignment operators */
%token  MULTassign  DIVassign   MODassign  /*  *=     /=     %=     */
%token  PLUSassign  MINUSassign            /*  +=     -=           */
%token  LSassign    RSassign               /*  <<=    >>=          */
%token  ANDassign   ERassign    ORassign   /*  &=     ^=     |=     */

/* Type definition of non-terminals that could be insert in the parse tree */

%type <nodePt> abstract_declaratoraccess_specifierrequired_annotation aviewpoint_list
access_specifier_opt
/***********viewpoints*********/
%type <nodePt> aggregate_key_viewpoint aggregate_name_viewpoint
%type <nodePt> additive_expression aggregate_key aggregate_name
%type <nodePt> aggregate_name_elaborationallocation_expression
%type <nodePt> AND_expressionany_operatorargument_expression_list
%type <nodePt> array_abstract_declaratorassignment_expression
%type <nodePt> assignment_operator              asterisk_or_ampersand
%type <nodePt> basic_declaration_specifierbasic_type_name
%type <nodePt> basic_type_specifier             bit_field_declarator
%type <nodePt> bit_field_identifier_declaratorcast_expression
%type <nodePt> clean_postfix_typedef_declaratorclean_typedef_declarator
%type <nodePt> comma_expressioncomma_expression_opt
%type <nodePt>  comma_opt_ellipsiscomplex_namecompound_statement
%type <nodePt>  conditional_expressionconstant
%type <nodePt>  constant_expression             constructed_declarator
%type <nodePt>  constructed_identifier_declarator
constructed_parameter_typedef_declarator
%type <nodePt>  constructed_paren_typedef_declarator
%type <nodePt>  constructor_conflicting_parameter_list_and_body
%type <nodePt>  constructor_conflicting_typedef_declarator
%type <nodePt>  constructor_function_declaration
constructor_function_definition
%type <nodePt>  constructor_function_in_classconstructor_init
%type <nodePt>  constructor_init_list
constructor_init_list_opt
%type <nodePt>  constructor_parameter_list_and_bodydeallocation_expression
```

```
/*****************view***********/
%type <nodePt>  view_declaration
%type <nodePt>  view_correspondance variable_correspondance method_correspondance
%type <nodePt>   parameter_view1 parameter_view2 parameter_view3
%type <nodePt>   method_para_v1 method_para_v2 method_para_v3
%type <nodePt> parameter_corresp_list
%type <nodePt>  declaration     declaration_list declaration_qualifier
%type <nodePt> declaration_qualifier_listdeclaration_specifier
%type <nodePt> declarator       declaring_list     default_declaring_list
%type <nodePt> derivation_listderivation_optelaborated_type_name
%type <nodePt> elaborated_type_name_elaborationenum_name
%type <nodePt> enum_name_elaborationenumerator_list
%type <nodePt> enumerator_list_no_trailing_commaenumerator_name
%type <nodePt> enumerator_value_opt
%type <nodePt> equality_expression              exclusive_OR_expression
%type <nodePt> expression_statementexternal_definitionfunction_declaration
%type <nodePt> function_definitionglobal_opt_scope_opt_complex_name
%type <nodePt> global_opt_scope_opt_deleteglobal_opt_scope_opt_enum_key
%type <nodePt> global_opt_scope_opt_identifierglobal_opt_scope_opt_operator_new
%type <nodePt> global_opt_scope_opt_typedefnameglobal_or_scope
%type <nodePt> global_or_scoped_typedefnameglobal_scope
%type <nodePt> identifier_declarator            inclusive_OR_expression
%type <nodePt> initializer        initializer_group initializer_list
%type <nodePt> initializer_optiteration_statement
%type <nodePt> jump_statementlabel              labeled_statement
%type <nodePt> linkage_specifier                logical_AND_expression
%type <nodePt> logical_OR_expression
%type <nodePt> member_conflict_declaring_item
%type <nodePt> member_conflict_paren_declaring_item
member_conflict_paren_postfix_declaring_item
%type <nodePt> member_declaration
%type <nodePt> member_declaration_list_optmember_declaring_list
%type <nodePt> member_default_declaring_list member_name
%type <nodePt> member_pure_opt                  multiplicative_expression
%type <nodePt> named_parameter_type_listnew_function_definition
%type <nodePt> non_casting_parameter_declaration non_elaborating_type_specifier
%type <nodePt> nonunary_constructed_identifier_declarator
%type <nodePt> old_function_declaratorold_function_body
%type <nodePt> old_function_definitionold_parameter_type_list
%type <nodePt> old_postfixing_abstract_declaratoroperator_function_name
%type <nodePt> operator_function_ptr_opt
%type <nodePt> operator_new_array_declaratoroperator_new_declarator_opt
%type <nodePt> operator_new_initializer_optoperator_new_type
%type <nodePt> parameter_declaration parameter_list
%type <nodePt> parameter_type_list              parameter_typedef_declarator
%type <nodePt> paren_identifier_declaratorparen_typedef_declarator
%type <nodePt> parent_class                                   point_member_expression
%type <nodePt> postfix_abstract_declarator
%type <nodePt> postfix_expression               program
%type <nodePt> postfix_identifier_declaratorpostfix_old_function_declarator
%type <nodePt> postfix_paren_typedef_declaratorpostfixing_abstract_declarator
```

```
%type <nodePt> primary_expression               relational_expression
%type <nodePt> scope            scope_opt_complex_name
%type <nodePt> scope_opt_identifierscoped_typedefname
%type <nodePt> scoping_nameselection_statement
%type <nodePt> shift_expression
%type <nodePt> simple_paren_typedef_declarator
%type <nodePt> statement        statement_list_opt
%type <nodePt> storage_classstring_literal_listsue_declaration_specifier
%type <nodePt> sue_type_specifiersue_type_specifier_elaboration
/*********viewpoints************/
%type <nodePt> tag_name_viewpoint tag_name_view
%type <nodePt> tag_name        translation_unit  type_name
%type <nodePt> type_qualifiertype_qualifier_listtype_qualifier_list_opt
%type <nodePt> type_specifiertypedef_declaration_specifier
%type <nodePt> typedef_declaratortypedef_type_specifier
%type <nodePt> template_argumenttemplate_argument_expression_list
%type <nodePt> template_headertemplate_parameter template_parameter_list
%type <nodePt> unary_abstract_declaratorunary_expression
%type <nodePt> unary_identifier_declaratorunary_modifier
%type <nodePt> virtual_opt
%type <nodePt> typeComment  template_name

/*************************************************************************/
%start program
/*************************************************************************/
%%

/********************** CONSTANTS ********************************/
constant:
      INTEGERconstant
        {$$ = insertElement("", "", NULL, $1, "",0,0,0,0,0,0);}
      | FLOATINGconstant
       {$$ = insertElement("", "", NULL, $1, "",0,0,0,0,0,0);}
       | OCTALconstant
       {$$ = insertElement("", "", NULL, $1, "",0,0,0,0,0,0);}
      | HEXconstant
       {$$ = insertElement("", "", NULL, $1, "",0,0,0,0,0,0);}
      | CHARACTERconstant
       {$$ = insertElement("", "", NULL, $1, "",0,0,0,0,0,0);}
      ;

string_literal_list:
        STRINGliteral
        {$$ = insertElement("", "", NULL, $1, "",0,0,0,0,0,0);}
        | string_literal_list STRINGliteral
          {createChildTab(); childTab[0] = $1;
           $$ = insertElement("","", childTab, $2,"", 0,0,0,0,0,0);}
         ;
/************************ EXPRESSIONS ********************************/
paren_identifier_declarator:
      scope_opt_identifier
        {createChildTab(); childTab[0]=$1;
```

```
        $$ = insertElement("","",childTab,0,"",$1->nameNode,0,$1->start,0,2,0);}

      | scope_opt_complex_name
        { createChildTab(); childTab[0]=$1;
       $$ = insertElement("","",childTab,0,"",$1->nameNode,0,0,0,0,0);}

      | '(' paren_identifier_declarator ')'
        {createChildTab();
         childTab[0] = insertElement("", ")",NULL,0,"",0,0,0,0,0,0);
         childTab[1] = $2;
       $$ = insertElement("", "(", childTab, 0, "",0,0,0,0,1,0);}
      ;

primary_expression:
      global_opt_scope_opt_identifier
        {createChildTab(); childTab[0]= $1;
         $$=insertElement("","",childTab,0,"",$1->nameNode,0,$1->start,$1->instance, $1-
>parite, $1->isViewpoint);}
      | global_opt_scope_opt_complex_name
        { $$ = $1;}
      | THIS
       {$$ = insertElement("", "this", NULL, 0, "",0,0,0,0,0,0);}
          /* C++, not ANSI C */
      | constant
                  {printf(" PRIMARY_EXPRESSION: 4 \n");
       $$ = $1;}
      | string_literal_list
        { $$ = $1;}
      | '(' comma_expression ')'
        {createChildTab(); childTab[0] = $2;
         childTab[1] = insertElement("", ")", NULL, 0, "",0,0,0,0,0,0);
        $$ = insertElement("", "(", childTab, 0, "",0,0,0,$2->instance,0,0);}
      ;

operator_function_name:
      OPERATOR any_operator
        {createChildTab(); childTab[0]=$2;
       $$ = insertElement("","",childTab,0,"",$2->tokenId,0,0,0,0,0);}

      | OPERATOR type_qualifier_list  operator_function_ptr_opt
        {createChildTab();
         childTab[0] = $2;
       childTab[1] = $3;
         $$ = insertElement("", "operator", childTab, 0, "",0,0,0,0,0,0);}

      | OPERATOR non_elaborating_type_specifier operator_function_ptr_opt
        {createChildTab();
       childTab[0] = $2;
       childTab[1] = $3;
         $$ = insertElement("", "operator", childTab, 0, "",0,0,0,0,0,0);}
      ;
```

```
operator_function_ptr_opt:
      /* nothing */ {$$ = NULL;}
      | unary_modifier       operator_function_ptr_opt
        {createChildTab(); childTab[0] = $1; childTab[1] = $2;
         $$ = insertElement("", "",childTab, 0, "",0,0,0,0);}
      | asterisk_or_ampersand operator_function_ptr_opt
        {createChildTab(); childTab[0] = $1; childTab[1] = $2;
         $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}
      ;
type_qualifier_list_opt:
      /* Nothing */
      { if (ptrCurrentClass != NULL)   /* Pour le cas ou on a juste des viewpoints */
        temporaryMethod = createInsertMethod("","","",0,0,required,provided, requiredList,isNull);
       $$ = NULL;
       }
      | type_qualifier_list
         {$$ = $1;
          temporaryMethod = createInsertMethod("","","",0,0,required,provided,
            requiredList,isNull);
       }
      ;

postfix_expression:
      primary_expression
      {printf("POSTFIX_EXPRESSION:1 \n");
      createChildTab(); childTab[0]= $1;
      $$ = insertElement("","",childTab,0,"",$1->nameNode,0,$1->start,$1->instance,$1-
>parite,$1->isViewpoint); }

      | postfix_expression '[' comma_expression ']'
        {isCallMethOrVar = 0;
        if(ptStructCall == 0)
          { ptStructCall = createCall($1->nameNode,"","",isNull);
          ptStructCall=createCall("","",ptrCurrentClass->nameClass,getInstance);
            ptStructCall=createCall("","",ptrCurrentClass->nameClass,listParameter);
          ptStructCall = createCall("","", "" , isCallVariable);
          isCallMethOrVar=2;
           }
         createChildTab(); childTab[0] = $1;
         childTab[1] = insertElement("","[",NULL,0,"",0,0,0,0,0,0);
         childTab[2] = $3;
         childTab[3] =insertElement("","]",NULL,0,"",0,0,0,0,0,0);
         $$ = insertElement("", "", childTab, 0, "",$1->nameNode,0,0,0,2,0);}

      | postfix_expression '(' ')'
        {if (currentVPMethod != NULL)
            ajout_appel(currentVPMethod,varIdent,$1->start,charPos);
        isCallMethOrVar = 1;
        if (ptrCurrentClass !=0)
            { if (ptStructCall == 0)
                {ptStructCall = createCall(varIdent,"","",isNull);
                ptStructCall=createCall("","",ptrCurrentClass->nameClass,getInstance);
```

```
                ptStructCall=createCall("","",ptrCurrentClass->nameClass,listParameter);
                ptStructCall = createCall("","", "" , isCallMethod);
                }
            else    ptStructCall = createCall("","", "" , isCallMethod);
            newInstance = ptStructCall->typeMember;}
          ptStructCall = 0;
          createChildTab(); childTab[0] = $1;
          childTab[1]=insertElement("parametresTyped:NULL","",NULL,0,"",0,0,0,0,0,0);
          $$ = insertElement("","",childTab,0,"",0,0,$1->start,0,1,0);}

        | postfix_expression '(' argument_expression_list ')'
          {printf("POSTFIX_EXPRESSION 4 \n");
          if (currentVPMethod != NULL)
            ajout_appel(currentVPMethod,varIdent,$1->start,charPos);
          isCallMethOrVar = 1;
          if (ptrCurrentClass !=0)
           {if(ptStructCall == 0)
            { printf("90 varIdent %s ,ptrCurrentClass->nameClass %s,  \n",varIdent,
ptrCurrentClass->nameClass );
              ptStructCall = createCall(varIdent,"","",isNull);
              ptStructCall=createCall("","",ptrCurrentClass->nameClass,getInstance);
              ptStructCall=createCall("","",ptrCurrentClass->nameClass,listParameter);
              ptStructCall = createCall("","", "" , isCallMethod);
            else      ptStructCall = createCall("","", "" , isCallMethod);
          newInstance = ptStructCall->typeMember; }
          ptStructCall = 0;
          createChildTab();
          childTab[0] = $1;
          childTab[1]=insertElement("parametresTyped:","",NULL,0,"",0,0,0,0,0,0);
          childTab[2] = $3;
          $$ = insertElement("", "", childTab, 0, "",0,$1->typeNode,0,0,1,0);}

        | postfix_expression DOT member_name
          {if(newInstance == NULL)
             { if (currentVPMethod != NULL)   /* IIIIappelIIII */
               ajout_appel(currentVPMethod,varIdent,$1->start,charPos);
              else
               { ptStructCall = createCall("","", varIdent, getInstance);
                  ptStructCall=createCall("","",varIdent,listParameter);}
              }
            else
            { if (currentVPMethod != NULL)   /* */
              ajout_appel(currentVPMethod,newInstance,$1->start,charPos);
              else
                {ptStructCall = createCall("","", newInstance, getInstance);
                  ptStructCall=createCall("","",newInstance,listParameter);}
             }
         createChildTab();
         childTab[0] = $3;
         childTab[1] = insertElement("fromClass:","",NULL,0,"",0,0,0,0,0,0);
          childTab[2] = insertElement("",".", NULL, 0, "",0,0,0,0,0,0);
          childTab[3] = $1;
```

```
        $$=insertElement("postType:","",childTab,0,"",$3->nameNode,0,0, $1->instance,2,0);
       }

      | postfix_expression ARROW member_name
        {if(newInstance == NULL)
            {if (currentVPMethod != NULL)   /* */
             ajout_appel(currentVPMethod,varIdent,$1->start,charPos);
             else
              {printf("appel de CREATE CALL varIdent = %s \n", varIdent);
               ptStructCall = createCall("","", varIdent, getInstance);
                ptStructCall=createCall("","",varIdent,listParameter);}
                }
          else
          {if (currentVPMethod != NULL)   /* */
            ajout_appel(currentVPMethod,newInstance,$1->start,charPos);
            else
             {
              ptStructCall = createCall("","", newInstance, getInstance);
                ptStructCall=createCall("","",newInstance,listParameter);
                }
          }
         createChildTab();
        childTab[0] = $3;
        childTab[1] = insertElement("fromClass:","",NULL,0,"",0,0,0,0,0,0);
         childTab[2] = $1;
        $$ = insertElement("","",childTab,0,"",0,0,0,$1->instance,2,0);}

      | postfix_expression ICR
        {printf("POSTFIX_EXPRESSION:7 \n");
         createChildTab(); childTab[0] = $1;
         $$ = insertElement("", "++", childTab, 0, "",0,0,0,0,2,0);}

      | postfix_expression DECR
        {createChildTab(); childTab[0] = $1;
         $$ = insertElement("", "--", childTab, 0, "",0,0,0,0,0,0);}

            /* The next 4 rules are the source of cast ambiguity */
      | TYPEDEFname                '(' ')'
        { $$ = insertElement("", "", NULL, $1, "",0,0,0,0,0,0);}

      | basic_type_name '(' assignment_expression ')'
        {createChildTab(); childTab[0] = $1;
         childTab[1] = insertElement("", "(", NULL, 0,"",0,0,0,0,0,0);
         childTab[2]=$3;
         childTab[3] = insertElement("", ")", NULL, 0, "",0,0,0,0,0,0);
         $$ = insertElement("", "", childTab, 0, "",0,0,0,0,1,0);}
;

member_name:
      scope_opt_identifier
        {createChildTab();
       childTab[0]=insertElement("","", NULL,0, "", 0,0,0,0,0,0);
```

```
        childTab[1] = $1;
        ptStructCall = createCall($1->nameNode,"","", isNull);
        $$=insertElement("","",childTab,0,"",$1->symbValue->name,0,0,0,0,0);}

      | scope_opt_complex_name
        {createChildTab(); childTab[0] = $1;
         $$ = insertElement("", "", childTab, 0, "",0,0,0,0,0,0);}

      | basic_type_name CLCL '~' basic_type_name  /* C++, not ANSI C */
        {createChildTab();
        childTab[0] = $1;
        childTab[1] = insertElement("::", "~", NULL, 0, "",0,0,0,0,0,0);
        childTab[2] = $4;
        $$=insertElement("","",childTab,0,"",0,0,0,0,0,0);}

      | declaration_qualifier_list  CLCL '~'  declaration_qualifier_list
        {createChildTab(); childTab[0] = $1;
         childTab[1] = insertElement("::", "~", NULL, 0, "",0,0,0,0,0,0);
         childTab[2] = $4;
        $$=insertElement("","",childTab,0,"",0,0,0,0,0,0);}

      | type_qualifier_list       CLCL '~'  type_qualifier_list
        {createChildTab(); childTab[0] = $1;
         childTab[1] = insertElement("::", "~", NULL, 0, "",0,0,0,0,0,0);
         childTab[2] = $4;
        $$=insertElement("","",childTab,0,"",0,0,0,0,0,0);}
      ;
allocation_expression:
        global_opt_scope_opt_operator_new  '(' type_name ')'
                                        operator_new_initializer_opt
        {createChildTab(); childTab[0] = $1;
         childTab[1] = insertElement("", "(", NULL, 0, "",0,0,0,0,0,0);
         childTab[2] = $3;
         childTab[3] = insertElement("", ")", NULL, 0,"",0,0,0,0,0,0);
         childTab[4] = $5;
         $$ = insertElement("", "", childTab, 0, "",0,0,0,0,0,0);}
;


     /* Following are C++, not ANSI C */
global_opt_scope_opt_operator_new:
      NEW
      {$$ = insertElement("", "new", NULL, 0, "",0,0,0,0,0,0);}
      | global_or_scope NEW
        {createChildTab(); childTab[0] = $1;
        childTab[1] = insertElement("", "new", NULL, 0, "",0,0,0,0,0,0);
         $$ = insertElement("", "", childTab, 0, "",0,0,0,0,0,0);}
      ;


operator_new_type:
      type_qualifier_list operator_new_declarator_opt operator_new_initializer_opt
```

```
        {createChildTab();
        childTab[0] = $1;
        childTab[1] = $2;
         childTab[2] = $3;
        $$=insertElement("","",childTab,0,"",0,0,0,0,0,0);}

      | non_elaborating_type_specifier operator_new_declarator_opt
                  operator_new_initializer_opt
        {createChildTab(); childTab[0] = $1; childTab[1] = $2;
          childTab[2] = $3; $$=insertElement("","",childTab,0,"",0,0,0,0,0,0);}
      ;

/****************************** DECLARATIONS ********************************/
declaration:
      declaring_list ';'
      {nodeOfDataVar = NULL;
       temporaryMethod = NULL;
       createChildTab(); childTab[0] = $1;
       childTab[1] = insertElement("","\n",NULL, 0,"",0,0,0,0,0,0);
      $$ = insertElement("","",childTab,0,"",0,0,$1->start,0,0,$1->isViewpoint);}

      | default_declaring_list ';'
      {createChildTab(); childTab[0] = $1;
       childTab[1] = insertElement("","\n",NULL, 0, "",0,0,0,0,0,0);
        $$ = insertElement("","",childTab,0,"",0,0,$1->start,0,0,$1->isViewpoint);}

view_declaration:
   VIEWDEF tag_name_view ':' tag_name_view AS tag_name_view '(' view_correspondance ')'
';'
       {
       /**** definition de la vue *****/
       /* nous avons deja, listeCorresp liste contenant la saisie des variables,
          et listmethodeCorresp contenant la saisie des methodes et leurs parametres*/
        defineView( $2->nameNode ,$4->nameNode, $6->nameNode, $2->start);

       /* ajouter la vue a la liste des vues possibles de la classe */
        ajout_a_class($2->nameNode,$4->nameNode);
       createChildTab(); childTab[0] = $2;  childTab[1] = $4;
       childTab[2] = $6;
        $$ = insertElement("", "",childTab, 0, "",0,0, $2->start,0,0,1);}
;

view_correspondance:

     /* nothing */
     {$$ = insertElement("", "", NULL, 0, "",0,0,0,0,0,0);}

     | method_correspondance view_correspondance
     { $$ = insertElement("", "",NULL, 0, "",0,0, 0,0,0,1);}

     |variable_correspondance  view_correspondance
     { $$ = insertElement("", "",NULL, 0, "",0,0, 0,0,0,1);}
```

```
        ;

method_correspondance:

      /* nothing */
       {$$ = insertElement("", "", NULL, 0, "",0,0,0,0,0,0);}

      | '<' method_para_v1 ',' method_para_v2 ',' method_para_v3 '>'
        {printf("METHOD_CORRESPONDANCE 2 \n");

         /* ajout a la liste de correspondance des methodes avec verification */
         /* definition et ajout des trois methodes a la liste de correspondance */
         defineMethodCorresp();

        /* reinitialisation des methodes deja ajoutees a la liste de correspondance */
         methodcorresporig = NULL;
         methodcorrespdestin = NULL;
         methodcorrespnew = NULL;
         createChildTab(); childTab[0] = $2;  childTab[1] = $4;
         childTab[2] = $6;
          $$ = insertElement("", "",childTab, 0, "",0,0, $2->start,0,0,1);}
;


method_para_v1:
        scope_opt_identifier  '(' parameter_corresp_list ')'
       { /****** creation de la methode origine avec ses parametres ******/
        methodcorresporig = createCorrespMethod($1->nameNode, param_corresp,
methodcorresporig);
        param_corresp = NULL;
        createChildTab();
        childTab[0] = $1;
        childTab[1] = $3;
        $$=insertElement("","",childTab,0,"",$1->nameNode,0,0,0,0,1);
       };

method_para_v2:
        scope_opt_identifier      '(' parameter_corresp_list ')'
       { /****** creation de la methode destination avec ses parametres ******/
        methodcorrespdestin = createCorrespMethod($1->nameNode,
param_corresp,methodcorrespdestin);
        param_corresp = NULL;
        createChildTab();
        childTab[0] = $1;
        childTab[1] = $3;
        $$=insertElement("","",childTab,0,"",$1->nameNode,0,0,0,0,1);
       };



method_para_v3:
      /* nothing */
```

```
        {methodcorrespnew = methodcorrespdestin;
          $$ = insertElement("", "", NULL, 0, "",0,0,0,0,0,0);}

        | scope_opt_identifier        '(' parameter_corresp_list ')'
        {/****** creation de la methode nouveau avec ses parametres ******/
        methodcorrespnew = createCorrespMethod($1->nameNode, param_corresp,
methodcorrespnew);
         param_corresp = NULL;
         createChildTab();
         childTab[0] = $1;
         childTab[1] = $3;
         $$=insertElement("","",childTab,0,"",$1->nameNode,0,0,0,0,1);
        };

parameter_corresp_list:
        /* nothing */
        {$$ = insertElement("", "", NULL, 0, "",0,0,0,0,0,0);}

        | tag_name_view
        { /****** ajout de la variable aux autres parametres *****/
         addparamcorresp($1->nameNode);
         printf("les paramtrs sont %s \n", param_corresp->name);
         createChildTab();
         childTab[0] = $1;
         $$=insertElement("","",childTab,$1,"",$1->nameNode,0,0,0,0,1);
        }

        | tag_name_view   ','   parameter_corresp_list
        { /****** ajout de la variable aux autres parametres *****/
         addparamcorresp($1->nameNode);
         createChildTab();
         childTab[0] = $1;
         childTab[1] = insertElement("nextParameter ", "",NULL, 0, "",0,0,0,0,0,0);
           childTab[2] = $3;
         $$=insertElement("","",childTab,$1,"",$1->nameNode,0,0,0,0,1);
        };

variable_correspondance:

       /* nothing */
        {$$ = insertElement("", "", NULL, 0, "",0,0,0,0,0,0);}

     |  '<' parameter_view1 ',' parameter_view2 ',' parameter_view3 '>'
         {/****** ajout a liste de correspondance ********/
          defineVariableCorresp($2->nameNode, $4->nameNode, $6->nameNode);
          createChildTab(); childTab[0] = $2;  childTab[1] = $4;
          childTab[2] = $6;
           $$ = insertElement("", "",childTab, 0, "",0,0, $2->start,0,0,1);}
;

parameter_view1:
        tag_name_view
```

```
        { printf("PARAMETER_VIEW1 1 \n");
         createChildTab();
         childTab[0] = $1;
         $$=insertElement("","",childTab,$1,"",$1->nameNode,0,0,0,0,1);
        };

parameter_view2:
        tag_name_view
        { printf("PARAMETER_VIEW2 2 \n");
         createChildTab();
         childTab[0] = $1;
         $$=insertElement("","",childTab,$1,"",$1->nameNode,0,0,0,0,1);
        };

parameter_view3:
       /* nothing */
        {$$ = insertElement("", "", NULL, 0, "",0,0,0,0,0,0);}
                        |
        tag_name_view
        { createChildTab();
         childTab[0] = $1;
         $$=insertElement("","",childTab,$1,"",$1->nameNode,0,0,0,0,1);
        };

declaring_list:
        declaration_specifier          declarator {} initializer_opt
          {createChildTab(); childTab[0] = $2; childTab[1] = $1; childTab[2] = $3;
           $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}
        | type_specifier               declarator {} initializer_opt
          {createChildTab(); childTab[0] = $2; childTab[1] = $1; childTab[2] = $3;
           $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}

        | basic_type_name              declarator {} initializer_opt
          { createChildTab(); childTab[0] = $2; childTab[1] = $1;
         childTab[2] = $3;
         /* ajout aux variables de la methode */
         if ((vpFlag ==1) && (currentVPMethod != NULL)&& (currentVar != NULL))
         { currentVar->nextVar = NULL;
         if (currentVPMethod->variableMethod == NULL)
                        currentVPMethod->variableMethod= currentVar;
         else
                        currentVPMethod->variableMethod->nextVar = currentVar; }
         if (nodeOfDataVar !=NULL )    /****** viewpoints ********/
           nodeOfDataVar=createInsertDataMem("",$1->tokenId,NULL,0,0,"",0,
0,NULL,isVarMethod);
         $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}

        | TYPEDEFname    declarator {} initializer_opt
          {createChildTab(); childTab[0] = $2;
         childTab[1] = insertElement("xType:", "", NULL, $1,"",0,0,0,0,0,0);
          childTab[2] = $3;
         if (nodeOfDataVar !=NULL )    /****** viewpoints ********/
```

```
      nodeOfDataVar = createInsertDataMem("",$1->name,NULL,0,0,"",0,0,NULL, isVarMethod);
       $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}

      | TYPEDEFname '<' template_argument_expression_list '>'declarator {}
      initializer_opt
      {if (nodeOfDataVar !=NULL )    /****** viewpoints ********/
       nodeOfDataVar = createInsertDataMem("",$1->name,NULL,0,0,"",0, 0,
NULL,isVarMethod);
        createChildTab(); childTab[0] =$3; childTab[1] = $5;
       childTab[2] = insertElement("xType:", "", NULL, $1, "",0,0,0,0,0,0);
        childTab[3] = $6;
         $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}
      | basic_type_name            constructed_declarator
        { if (nodeOfDataVar !=NULL )    /****** viewpoints ********/
         nodeOfDataVar=createInsertDataMem("",$1->tokenId,NULL,0,0,"",0, 0,
NULL,isVarMethod);
         createChildTab(); childTab[0] = $2; childTab[1] = $1;
         $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}

      | TYPEDEFname                constructed_declarator
        {if (nodeOfDataVar !=NULL )    /****** viewpoints ********/
         nodeOfDataVar=createInsertDataMem("",$1->name,NULL,0,0,"",0, 0,
NULL,isVarMethod);
         createChildTab(); childTab[0] = $2;
        childTab[1] = insertElement("xType:", "", NULL, $1,"",0,0,0,0,0,0);
         $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}

      | TYPEDEFname '<' template_argument_expression_list '>' constructed_declarator
        {if (nodeOfDataVar !=NULL )    /****** viewpoints ********/
         nodeOfDataVar=createInsertDataMem("",$1->name,NULL,0,0,"",0, 0,
NULL,isVarMethod);
         createChildTab(); childTab[0] = $5; childTab[1]= $3;
         childTab[2] = insertElement("xType:", "", NULL, $1, "",0,0,0,0,0,0);
         $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}

      | global_or_scoped_typedefname constructed_declarator
        { createChildTab(); childTab[0] = $2; childTab[1] = $1;
         $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}
      | declaring_list ','          constructed_declarator
        {createChildTab(); childTab[0] = $1; childTab[1] = $3;
         $$ = insertElement("", "\n",childTab, 0, "",0,0,0,0,0,0);}
      ;

constructed_parameter_typedef_declarator:
      TYPEDEFname    '(' argument_expression_list ')'
        { createChildTab(); childTab[0] = $3;
         $$ = insertElement("xType:", "", childTab, $1, "",0,0,0,0,0,0);}

      | TYPEDEFname '<' template_argument_expression_list '>' '('
                                                  argument_expression_list ')'
        {createChildTab();
       childTab[0] = insertElement("xType:", "", NULL, $1,"",0,0,0,0,0,0);
```

```
        childTab[1] = $6;
        $$ = insertElement("xType:", "", childTab, $1, "",0,0,0,0,0,0);}

      | TYPEDEFname  postfixing_abstract_declarator '(' argument_expression_list ')'
        { createChildTab(); childTab[0] = $2;
       childTab[1] = insertElement("xType:", "", NULL, $1, "",0,0,0,0,0,0);
         childTab[2] = $4;
         $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}
;


declaration_specifier:
      basic_declaration_specifier         /* Arithmetic or void */
        {createChildTab(); childTab[0] = $1;
         $$ = insertElement("","",childTab,0,"",0,$1->typeNode, $1->start,0,0,0);}
      | sue_declaration_specifier         /* struct/union/enum/class */
        {createChildTab(); childTab[0] = $1;
         $$ = insertElement("","",childTab,0,"",0,$1->typeNode, $1->start,0,0,0);}
      | typedef_declaration_specifier     /* typedef*/
        {createChildTab(); childTab[0] = $1;
       $$ = insertElement("","",childTab,0,"",0,$1->typeNode, $1->start,0,0,0);}
      ;

type_specifier:
      basic_type_specifier                /* Arithmetic or void */
        {createChildTab(); childTab[0]=$1;
       $$ = insertElement("","",childTab, 0,"",0,$1->typeNode, $1->start,0,0,0);}
      | sue_type_specifier                /* Struct/Union/Enum/Class */
        {createChildTab(); childTab[0] = $1;
        $$ = insertElement("","",childTab,0,"",0,0, $1->start,0,0,0);}
      | sue_type_specifier_elaboration    /* elaborated Struct/Union/Enum/Class */
        {createChildTab(); childTab[0] = $1;
        $$ = insertElement("","",childTab,0,"",0,0, $1->start,0,0,0);}
      | typedef_type_specifier            /* Typedef */
        {createChildTab(); childTab[0]=$1;
       $$=insertElement("","",childTab,0,"",0,$1->typeNode,$1->start,0,0,0);}
      ;

declaration_qualifier_list:  /* storage class and optional const/volatile */
      storage_class
        {$$ = NULL;}
      | type_qualifier_list storage_class
        { $$ = NULL;}
      | declaration_qualifier_list declaration_qualifier
        {$$ = NULL;}
      ;

type_qualifier_list:
      type_qualifier
        {createChildTab(); childTab[0] = $1;
         $$ = insertElement("","",childTab, 0,"",0,0, $1->start,0,0,0);}
      | type_qualifier_list type_qualifier
```

```
        {createChildTab(); childTab[0] = $1; childTab[1] = $2;
         $$ = insertElement("", "",childTab, 0, "",0,0, $1->start,0,0,0);}
      ;
aggregate_name_viewpoint:
       aggregate_key_viewpoint tag_name_viewpoint
        {createChildTab(); childTab[0] = $1; childTab[1] = $2;
       $$ = insertElement("", "",childTab, 0, "",0,0, $1->start,0,0,1); }
     ;
aggregate_name:
       aggregate_key tag_name
        {createChildTab(); childTab[0] = $1; childTab[1] = $2;
       $$ = insertElement("", "",childTab, 0, "",0,0, $1->start,0,0,0);}

      | global_scope scope aggregate_key tag_name
        {childTab[0] = $1; childTab[1] = $2; childTab[2] = $3;
         $$ = insertElement("", "",childTab, 0, "",0,0, $1->start,0,0,0);}

      | global_scope      aggregate_key tag_name
        {createChildTab();
       childTab[0] = $1; childTab[1] = $2; childTab[2] = $3;
         $$ = insertElement("", "",childTab, 0, "",0,0, $1->start,0,0,0);}

      | scope             aggregate_key tag_name
        {createChildTab();
       childTab[0] = $1; childTab[1] = $2; childTab[2] = $3;
         $$ = insertElement("", "",childTab, 0, "",0,0, $1->start,0,0,0);}

      | template_header aggregate_key tag_name
        {createChildTab();
       childTab[0] = $1; childTab[1] = $2; childTab[2] = $3;
         $$ = insertElement("", "",childTab, 0, "",0,0, $1->start,0,0,0);}
      ;

derivation_opt:
      /* nothing */
      {if (ptrEndOfClass != NULL)
          { i=0;
            while(i < Array)
              {ptrEndOfClass->superClass[i] = NULL;
               i++;}
            ptrView = "Private";}
       if (currentViewpoint == NULL) /******viewpoints*******/
             insertField("","",comment, 0, 0, isaComment);
      $$ = insertElement("", "", NULL, 0, comment,0,0,0,0,0,0); }

      | ':' derivation_list
        { ptrView = "Private";
         insertField("","",comment, 0, 0, isaComment);
       createChildTab();
       childTab[0] = insertElement("beginningOfSuperClass","\n",NULL,0,comment,0,0,0,0,0,0);
         childTab[1] = $2;
       childTab[2] = insertElement("endOfSuperClass","\n",NULL,0,comment,0,0,0,0,0,0);
```

```
        $$ = insertElement("", "",childTab, 0, comment,0,0,0,0,0,0); }
      ;

derivation_list:
      parent_class
      {insertField($1->nameNode, ptrView, "", 0, 0, isaSuperClass);
       createChildTab(); childTab[0] = $1;
       $$ = insertElement("","",childTab, 0,"",$1->nameNode,0,0,0,0,0);}
      | derivation_list ',' parent_class
        { insertField($3->nameNode,ptrView,"", 0, 0, isaSuperClass);
        createChildTab(); childTab[0] = $1; childTab[1] = $3;
      $$ = insertElement("", "\n",childTab, 0, comment,$3->nameNode,0,0,0,0,0);}
      ;

parent_class:
       global_opt_scope_opt_typedefname
       { ptrView = "Private";
       createChildTab(); childTab[0] = $1;
       $$ = insertElement("","",childTab,0,"",$1->nameNode,0,0,0,0,0);}

      | access_specifier virtual_opt global_opt_scope_opt_typedefname
        {createChildTab(); childTab[0] = $1; childTab[1] = $2; childTab[2] = $3;
         $$ = insertElement("", "",childTab, 0, "",$3->nameNode,0,0,0,0,0);}
      ;
required_annotation:
       REQUIRED '(' aviewpoint_list ')'
       {createChildTab();
        required = 1;
       childTab[0] = insertElement("","\n",NULL,0,"",0,0,0,0,0,0);
       $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}

      | PROVIDED '(' aviewpoint_list ')'
       {createChildTab();
        provided = 1;
       childTab[0] = insertElement("","\n",NULL,0,"",0,0,0,0,0,0);
       $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}
       ;

aviewpoint_list:
       IDENTIFIER
       {addViewpoint($1->name);
        $$ = insertElement("les points de vue a creer", "\n",childTab, 0, "",0,0,0,0,0,0);}
       | IDENTIFIER ',' aviewpoint_list
        {createChildTab();
       childTab[1] = $3;
       childTab[0] = insertElement("list de PV a creer","\n",NULL,0,"",0,0,0,0,0,0);
       addViewpoint($1->name);
         $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);
         }
       | TYPEDEFname
         {createChildTab();
         childTab[0] = insertElement("list de PV a creer","\n",NULL,0,"",0,0,0,0,0,0);
```

```
          addViewpoint($1->name);
          $$ = insertElement("it's a typedefname", "\n",childTab, 0, "",0,0,0,0,0,0);}

        | TYPEDEFname ',' aviewpoint_list
         {createChildTab();
        childTab[1] = $3;
        childTab[0] = insertElement("","\n",NULL,0,"",0,0,0,0,0,0);
        addViewpoint($1->name);
          $$ = insertElement("", "",childTab, $1, "",0,0,0,0,0,0);
          };
access_specifier:
      PUBLIC
        { require = 0;
        ptrView = "Public";
        createChildTab();
        printf(" ACCESS SPECIFIER 1 \n");
        childTab[0] = insertElement("","\n",NULL,0,"",0,0,0,0,0,0);
        $$ = insertElement("xView:", "Public",childTab, 0, "",0,0,0,0,0,0);}
      | PRIVATE
        { require = 0;
        ptrView ="Private";
        createChildTab();
        childTab[0] = insertElement("","\n",NULL,0,"",0,0,0,0,0,0);
        $$ = insertElement("xView:", "Private",childTab, 0, "",0,0,0,0,0,0);}
      | PROTECTED
        {  require = 0;
         ptrView = "Protected";
        createChildTab();
        childTab[0] = insertElement("","\n",NULL,0,"",0,0,0,0,0,0);
        $$ = insertElement("xView:", "Protected",childTab, 0, "",0,0,0,0,0,0);}

      | REQUIRES
        {  require = 1;
        createChildTab();
        childTab[0] = insertElement("","\n",NULL,0,"",0,0,0,0,0,0);
       $$ = insertElement("xView:", "REQUIRES",childTab, 0, "",0,0,0,0,0,1);}

      | PROVIDES
        {  require = 0;
        createChildTab();
        childTab[0] = insertElement("","\n",NULL,0,"",0,0,0,0,0,0);
       $$ = insertElement("xView:", "PROVIDES",childTab, 0, "",0,0,0,0,0,1);}
       ;
aggregate_key_viewpoint:
      VIEWPOINT
        { vpFlag = 1;
        $$ = insertElement("viewpoint", "",NULL, 0, "",0,0, charPos-9,0,0,1);
       }

member_declaration_list_opt:
      /* nothing */
     {currentVPParameter = NULL;        /********** viewpoints *****************/
```

```
        currentVPVariable = NULL;        /********** viewpoints *****************/
        $$ = insertElement("","",NULL,0,"",0,0,0,0,0,0); }

      | member_declaration_list_opt typeComment member_declaration
        {printf("MEMBER_DECLARATION_LIST_OPT 2 \n");
         createChildTab(); childTab[0] = $1;
        currentVPMethod = NULL;
       currentVPParameter = NULL;
       currentVPVariable = NULL;
         childTab[1] = insertElement("", "", NULL, 0, comment,0,0,0,0,0,0);
         childTab[2] = insertElement("", "xEnd\n", NULL, 0, "",0,0,0,0,0,0);
         childTab[3] = $2;
         childTab[4] = $3;
         $$ = insertElement("", "",childTab, 0, "",0,0,0,0,$3->parite,0);}
      ;

typeComment:
        { $$ = NULL;
         if ((comment != 0) && (temporaryMethod != 0))
            {temporaryMethod = createInsertMethod("","",comment,0,0,0,0,NULL, isaComment);
           }
          else
             if ((methOrVar ==1) && (nodeOfDataVar  !=NULL))   /* c.a.d une variable */
               nodeOfDataVar=createInsertDataMem("","",NULL,0,0,comment,0, 0,
NULL,isCommentVar);
        temporaryMethod = 0;
        }
;

member_declaration:
     member_declaring_list ';'
    {if (currentVPMethod != NULL) {currentVPMethod->typeMethod = methodtype;
                                   currentVPMethod->startVPMethod = $1->start;}
      methodtype ="";
      if (openBloc == 1) {setCurrentBloc(close); openBloc=0;}
      if (ptrCurrentClass != NULL)       /**** cas ou on a juste des points de vue****/
          { if ((int)($1->parite) == 2)   /* (methOrVar == 1) */
              {if (nodeOfDataVar != NULL)
                  nodeOfDataVar=createInsertDataMem("","",NULL,$1->start,charPos,"",0, 0,
NULL,isaCharPos);}
    else if((int)($1->parite) == 1)  /* it is a method */
     { if (temporaryMethod != NULL)
        temporaryMethod=createInsertMethod("","","",$1->start,charPos,
0,0,NULL,isaCharPosDec); }
       }
       createChildTab(); childTab[0]= $1;
       $$ = insertElement("","",childTab, 0,"",0,0,$1->start,0,$1->parite,0);
       currentVPMethod = NULL;
       currentTestVariable =NULL;
       currentVar = NULL;
        requiredList = NULL;
        required = 0;
```

```
        provided =0;
}
    | member_default_declaring_list ';'
      {required = 0; provided = 0;
       requiredList = NULL;
       $$ = $1;}

     | new_function_definition       ';'    /* C++, not ANSI C */
     {printf("MEMBER_DECLARATION 3: \n");
      if(currentVPMethod != NULL)
         {
          currentVPMethod->startVPMethod = $1->start;
          currentVPMethod->endVPMethod = charPos;

         }
       methodtype ="";
       currentTestVariable = NULL;
       if (temporaryMethod != NULL)
       temporaryMethod=createInsertMethod("","","",$1->start,charPos,
0,0,NULL,isaCharPosDec);
        createChildTab(); childTab[0] = $1;
       required = 0; provided =0;
        requiredList = 0;
       $$=insertElement("","",childTab,0,"",0,0,$1->start,0,1,0); }

     | access_specifier ':'
       { printf("MEMBER_DECLARATION 4: \n");
        $$ = $1;}

     | constructor_function_in_class      /* C++, not ANSI C */
      {printf("MEMBER_DECLARATION 5: \n");
       if ((vpFlag !=NULL) && (currentVPVariable != NULL))
             {currentVPVariable->startVPVar=$1->start;
              currentVPVariable->endVPVar=charPos;}
       if ((vpFlag !=NULL) && (currentVPMethod != NULL))
              {currentVPMethod->startVPMethod =$1->start;
               currentVPMethod->endVPMethod=charPos;}
       if (((int)($1->parite) == 2) && (nodeOfDataVar !=NULL)) /* (methOrVar == 1) i.g variable*/
     nodeOfDataVar=createInsertDataMem("","",NULL,$1->start,charPos,"",0,0,NULL,
isaCharPos);
          else if (((int)($1->parite) == 1) && (temporaryMethod !=NULL)) /* it is a method */
                {temporaryMethod=createInsertMethod("","","",$1->start,charPos,
0,0,NULL,isaCharPosDec);
                 }
        createChildTab(); childTab[0] = $1;
        childTab[1] = insertElement("xType: Constructor","", NULL,0,"", 0,0,0,0,0,0);
        $$ = insertElement("", "",childTab,0, "",0,0,$1->start,0,$1->parite,0);}

     | identifier_declarator         ';'
       {if (methOrVar == 1)
          nodeOfDataVar=createInsertDataMem("","",NULL,$1->start,charPos,"",0, 0,
NULL,isaCharPos);
```

```
          else
            temporaryMethod=createInsertMethod("","","",$1->start,charPos,0,0, NULL,
isaCharPosDec);
        $$ = $1;}

;

/* Ordre d'insertion dans l'arbre est inverse pour que declarator apparaisse
avant son type */

member_declaring_list:       /* Can possibly redeclare typedefs */
      type_specifier     declarator     member_pure_opt
        { if (isDeclarationMethOrVar == 0)
        {isDeclarationMethOrVar =1;
         methOrVar = 1;
         if (vpFlag != 1)   /* avant de faire la modification dans la variable */
                    nodeOfDataVar=createInsertDataMem("",$1->typeNode,NULL,0,0,"",0, 0,
NULL,isAttribut);
         }
         else
          if (isDeclarationMethOrVar == 1)
            { methOrVar = 0;
            temporaryMethod=createInsertMethod("",$1->typeNode,"",0,0,0,0,NULL, isaType);
            }
       createChildTab();
       childTab[0] = $2;
       childTab[1] = $1;
       childTab[2] = $3;
   $$=insertElement("","",childTab,0,"",0,$1->typeNode,$1->start,0,$2->parite,0);}

      | basic_type_name    declarator member_pure_opt
        {printf(" MEMBER_DECLARING_LIST 2 \n");
       methodtype = $1->tokenId;
       if (currentVar != NULL)
          currentVar->startVPVar = $1->start; /* debut de la variable */
        if ((currentVPParameter != NULL  ) && (currentVPMethod != NULL ))
          currentVPMethod->parameterMethod = currentVPParameter;

      /**** association des variables a la methode ****/
      if ((currentVPVariable != NULL  ) && (currentVPMethod != NULL ))
        currentVPMethod->variableMethod = currentVPVariable;
        createChildTab();  childTab[0] = $2;
       childTab[1] = $1; i = 1;
       if (isDeclarationMethOrVar == 0)
        {isDeclarationMethOrVar =1; methOrVar = 1;
         if (vpFlag != 1)    /* avant de faire la modification dans la variable */
            nodeOfDataVar=createInsertDataMem("",$1->tokenId,NULL,0,0,"",0, 0,
NULL,isAttribut);
       }
         else if (isDeclarationMethOrVar == 1)
           {methOrVar = 0;
            if (vpFlag != 1)
```

```
        temporaryMethod=createInsertMethod("",$1->tokenId,"",0,0,0,0,NULL, isaType);
       }
childTab[2] = $3;
$$ = insertElement("", "",childTab, 0, "",0,0, $1->start,0,$2->parite,0);
}
| required_annotation ':' basic_type_name    declarator member_pure_opt
  {methodtype = $3->tokenId;
 if (currentVar != NULL)
    currentVar->startVPVar = $3->start; /* debut de la variable */
  if ((currentVPParameter != NULL  ) && (currentVPMethod != NULL ))
     currentVPMethod->parameterMethod = currentVPParameter;

/**** association des variables a la methode ****/
if ((currentVPVariable != NULL  ) && (currentVPMethod != NULL ))
   currentVPMethod->variableMethod = currentVPVariable;
 createChildTab();  childTab[0] = $4;
 childTab[1] = $3; i = 1;
 if (isDeclarationMethOrVar == 0)
  {isDeclarationMethOrVar =1; methOrVar = 1;
   if (vpFlag != 1)    /* avant de faire la modification dans la variable */
   nodeOfDataVar=createInsertDataMem("",$3->tokenId,NULL,0,0,"",0, 0, NULL,isAttribut);
  }
   else if (isDeclarationMethOrVar == 1)
     { methOrVar = 0;
        if (vpFlag != 1)
        temporaryMethod=createInsertMethod("",$3->tokenId,"",0,0,0,0,NULL, isaType);
       }
childTab[2] = $5;
$$ = insertElement("", "",childTab, 0, "",0,0, $3->start,0,$4->parite,0);
}

| global_or_scoped_typedefname declarator member_pure_opt
  {printf(" MEMBER_DECLARING_LIST 3 \n");
  createChildTab(); childTab[0] = $2; childTab[1] = $1; childTab[2] = $3;
  $$ = insertElement("", "",childTab, 0, "",0,0, $1->start,0,$2->parite,0);}

| member_conflict_declaring_item
  {createChildTab(); childTab[0] = $1;
 if (currentVPMethod != NULL)  methodtype = currentVPMethod->typeMethod;
 if (currentVar != NULL)
    currentVar->startVPVar = $1->start;  /* debut de la variable */
  $$ = insertElement("", "",childTab, 0, "",0,0, $1->start,0,$1->parite,0);}

| required_annotation ':' member_conflict_declaring_item
  {createChildTab(); childTab[0] = $1;
  childTab[1] = $3;
  $$ = insertElement("", "",childTab, 0, "",0,0, $3->start,0,$3->parite,0);}

| member_declaring_list ','    declarator member_pure_opt
  {createChildTab(); childTab[0] = $1; childTab[1] = $3;
   $$ = insertElement("", "\n",childTab, 0, "",0,0, $1->start,0,0,0);}
;
```

```
/* Ordre d'insertion dans l'arbre est inverse pour que declarator apparaisse
avant son type */

member_conflict_declaring_item:
        TYPEDEFname      identifier_declarator   member_pure_opt
         {if (openBloc==0) {setCurrentBloc(open); openBloc = 1;}
        if (isDeclarationMethOrVar == 0)
        {  methOrVar = 1;
          if (vpFlag != 1)  /* avant de faire la modification dans la variable */
              nodeOfDataVar=createInsertDataMem("",$1->name,NULL,0,0,"",0, 0,
NULL,isAttribut);
          else     /* vpFlag ==1 */
           if ((currentTestVariable != NULL) &&
              (currentVPMethod == NULL))
             {currentTestVariable->typeVar = $1->name;
              currentVar = createVar(currentTestVariable->nameVar, $1->name, currentViewpoint-
>nameViewpoint,NULL,charPos - strlen(currentTestVariable->nameVar), require);       }

        }     /* isDeclarationMethodOrVar == 0 */
        else
        { methOrVar = 0;
          if (vpFlag != 1) /* on n'est pas dans un point de vue */
            temporaryMethod = createInsertMethod("",$1->name,"",0,0,0,0,NULL, isaType);
          else
          if (currentVPMethod != NULL)
           currentVPMethod->typeMethod=$1->name;
         } /* end isDeclarationMethodOrVar ==1 */

      if ((currentVPParameter != NULL  ) && (currentVPMethod != NULL ))
          currentVPMethod->parameterMethod = currentVPParameter;

      /**** association des variables a la methode ****/
      if ((currentVPVariable != NULL  ) && (currentVPMethod != NULL ))
        currentVPMethod->variableMethod = currentVPVariable;
       createChildTab(); childTab[0] = $2;
      childTab[1] = insertElement("xType:", "", NULL, $1, "",0,0,0,0);
        childTab[2] = $3;
        $$ = insertElement("","",childTab,0,"",$1->name,0,$1->startPos,0,$2->parite,0);}

      |  TYPEDEFname '<' template_argument_expression_list '>'
                  identifier_declarator          member_pure_opt
       {if (vpFlag != 1)    /* avant de faire la modification dans la variable */
       nodeOfDataVar = createInsertDataMem("",$1->name,NULL,0,0,"",0, 0, NULL,isAttribut);
        createChildTab(); childTab[0] = $3; childTab[1] = $5;
      childTab[2] = insertElement("xType:", "", NULL, $1, "",0,0,0,0,0,0);
        childTab[3] = $6;
       $$=insertElement("","",childTab,0,"",$5->nameNode,0,$1->startPos,0,$5->parite,0);}

       | TYPEDEFname  parameter_typedef_declarator member_pure_opt
          {createChildTab(); childTab[0] = $2;
      childTab[1] = insertElement("xType:", "", NULL, $1, "",0,0,0,0,0,0);
       childTab[2] = $3; $$=insertElement("","",childTab,0,"",0,0,0,0,0,0);}
```

```
        | declaration_specifier identifier_declarator  member_pure_opt
         {if ((vpFlag ==1) && (currentTestVariable != NULL))
          {currentTestVariable->typeVar = currenttype;
           currentVar = createVar(currentTestVariable->nameVar, currenttype, currentViewpoint-
>nameViewpoint,NULL,charPos - strlen(currentTestVariable->nameVar), require);}  /*pour
mettre a jour le type de la variable */
         if (currentVPMethod != NULL) currentVPMethod->typeMethod= methodtype;
         createChildTab(); childTab[0] = $2; childTab[1] = $1; childTab[2] = $3;

       if ( (nodeOfDataVar != NULL) && (temporaryMethod == NULL))
        nodeOfDataVar=
              createInsertDataMem("",$1->typeNode,NULL,0,0,"",0,0, NULL,isAttribut);
       if (temporaryMethod != NULL)
        temporaryMethod = createInsertMethod("",$1->typeNode,"",0,0,0,0,NULL,isaType);
        $$ = insertElement("", "",childTab, 0, "",0,$1->typeNode,$1->start,0,$2->parite,0);}

       | declaration_specifier parameter_typedef_declarator  member_pure_opt
         {createChildTab(); childTab[0] = $2; childTab[1] = $1; childTab[2] = $3;
          $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}

       | declaration_specifier simple_paren_typedef_declarator  member_pure_opt
         {createChildTab(); childTab[0] = $2; childTab[1] = $1; childTab[2] = $3;
          $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}
;

parameter_type_list:
       '(' ')'     type_qualifier_list_opt
         { if ( temporaryMethod != NULL)
           {if(strcmp(temporaryMethod->nameMethod,"")!=0)
                 temporaryMethod = createInsertMethod("","","",0,0,0,0,NULL, identifyMethod);}
        createChildTab(); childTab[0]=$3;
         $$ = insertElement("","",childTab,0,"",0,0,0,0,1,0);}

       | '(' type_name ')'                type_qualifier_list_opt
         {if(strcmp(temporaryMethod->nameMethod,"")!=0)
          temporaryMethod = createInsertMethod("","","",0,0,0,0,NULL,identifyMethod);
         createChildTab();
          childTab[0] = $2;
        childTab[1] = insertElement("endOfParameter","\n",NULL,0,"",0,0,0,0,1,0);
        childTab[2]=$4;
          $$=insertElement("beginningOfParameter","\n",childTab,0,"",0,0,0,0,1,0);}

       | '(' type_name initializer ')' type_qualifier_list_opt /*C++, not ANSI C*/
         {if (strcmp(temporaryMethod->nameMethod,"")!=0)
          temporaryMethod = createInsertMethod("","","",0,0,0,0,NULL,identifyMethod);
         createChildTab(); childTab[0] = $2;
       childTab[1] = insertElement("endOfParameter","\n",NULL, 0, "",0,0,0,0,0,0);
        childTab[2]=$3; childTab[3]=$5;
          $$=insertElement("beginningOfParameter","\n",childTab,0,"",0,0,0,0,1,0);}

       | '(' named_parameter_type_list ')' type_qualifier_list_opt
```

```
        { /***** ici current methode = NULL **********/
        if (ptrCurrentClass != NULL )    /** cas ou on a juste des viewpoints ***/
       { if (temporaryMethod !=NULL)
        if(strcmp(temporaryMethod->nameMethod,"")!=0)
         temporaryMethod=createInsertMethod("","","",0,0,0,0,NULL, identifyMethod);}
          createChildTab(); childTab[0] = $2;
        childTab[1]=insertElement("endOfParameter","\n",NULL,0,"",0,0,0,0,0,0);
        childTab[2]=$4;
          $$=insertElement("beginningOfParameter","\n",childTab,0,"",0,0,0,0,1,0);}
        ;

/*************************** STATEMENTS *******************************/

statement:
        labeled_statement { $$ = $1;}
        | compound_statement   { $$ = $1;}
        | expression_statement   { $$ = $1;}
        | selection_statement { $$ = $1;}
        | iteration_statement{ $$ = $1;}
        | jump_statement{ $$ = $1;}
        | declaration {$$ = $1;}
        ;

labeled_statement:
        label                   ':' statement
          {createChildTab(); childTab[0] = $1;
           childTab[1] = insertElement("", ":",NULL, 0, "",0,0,0,0,0,0);
           childTab[2] = $3;
           $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}
;

declaration_list:
        declaration
          {$$ = $1;}
        | declaration_list declaration
          {createChildTab(); childTab[0] = $1; childTab[1] = $2;
           $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}
        ;

statement_list_opt:
        /* nothing */
        {printf("STATEMENT_LIST_OPT 1 \n");
         beginingDefinition = 1;       /** viewpoints on est dans la definition d'une methode */
         $$ = NULL;}

        | statement_list_opt statement
                        { printf("STATEMENT_LIST_OPT 2 \n");
          createChildTab(); childTab[0] = $1; childTab[1] = $2;
                         $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);
         }
        ;
```

```
expression_statement:
      comma_expression_opt ';'
                  { newInstance = NULL;
       createChildTab(); childTab[0] = $1;
                   childTab[1] = insertElement("", ";\n",NULL, 0, "",0,0,0,0,0,0);
                  $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}
      ;

selection_statement:
.....
/************************* EXTERNAL DEFINITIONS ************************/
program:
      translation_unit
      {parseTree = $1;} ;

translation_unit:
      /* nothing */  { $$ = NULL;}
      | translation_unit external_definition
        { createChildTab(); childTab[0] = $1; childTab[1] = $2;
       ptrCurrentClass = NULL;
        $$ = insertElement("", "",childTab,0, "",0,0,0,0,0,0);
         yyerrok; }
      | translation_unit error {printf(" TRANSLATION UNIT 3 \n");};

external_definition:
      function_declaration                    /* C++, not ANSI C*/
        {createChildTab(); childTab[0] = $1;
         childTab[1] = insertElement("", "", NULL, 0, comment,0,0,0,0,0,0);
         childTab[2] = insertElement("", "xEnd\n",NULL, 0, "",0,0,0,0,0,0);
      /* ici on est a la fin de la definition d'une fonction globale, Il
         faut la copier depuis son debut jusqu'a charPos */
        resultat=fopen(filename, "a");
          laposition = ftell(yyin);
          fseek(yyin,$1->start,0);
          indicee = $1->start;
           while (indicee < charPos)
             {ch = getc(yyin);
              indicee++;
              putc(ch,resultat);}
          fprintf(resultat,"\n");
          fclose(resultat);
          fseek(yyin,laposition,0);
        $$ = insertElement("", "", childTab, 0, "",0,0,0,0,0,0);}

      | function_definition
        { temporaryMethod = 0;
         createChildTab(); childTab[0] = $1;
         childTab[1] = insertElement("", "", NULL, 0, comment,0,0,0,0,0);
         childTab[2] = insertElement("", "xEnd\n",NULL, 0, "",0,0,0,0,0,0);

       /* ici on est a la fin de la definition d'une fonction globale, Il
          faut la copier depuis son debut jusqu'a charPos */
```

```
  copy_function($1->start, charPos,yyin,filename);
   $$ = insertElement("", "", childTab, 0, "",$1->nameNode,0,0,0,0,0);}

| declaration
  { createChildTab(); childTab[0] = $1;
 if ((ptrCurrentClass != NULL) && (vpFlag ==0))
    {insertField("","","", $1->start, charPos, isaCharPos);
      insertField("","",comment, 0, 0, isaComment);
      /* apres la creation de la classe on transforme en points de vue */
      class_a_Viewpoints(ptrCurrentClass->nameClass);
     laposition = ftell(yyin);
    fseek(yyin,ptrCurrentClass->startClass,0);
    resultat=fopen(filename, "a");
    indicee = ptrCurrentClass->startClass; /* parcours le long de la class */
    nombre =0;
    traite =0; /* pour ecrire viewableClass */
     while(indicee < laposition)
      {ch = getc(yyin);
       indicee++;
        if (((ch == ':') || (ch =='{')) && (traite == 0))
              { traite =1;
              if (ch == ':')
                fprintf(resultat, " : virtual public Viewable_classe, \n \t");
              else fprintf(resultat," : virtual public Viewable_classe {\n \t");
              ch = getc(yyin);
              indicee++;
              }
              /* cas ou on a provided ou required */
      if (((ch == 'p') || (ch  == 'r')) && (traite == 1))
              {ch_c[0] = getc(yyin);
               ch_c[1] = getc(yyin);
               ch_c[2] = getc(yyin);
               ch_c[3] = getc(yyin);
               ch_c[4] = getc(yyin);
               ch_c[5] = getc(yyin);
               ch_c[6] = getc(yyin);
       if((strcmp(ch_c,"rovided" ) == 0) || (strcmp (ch_c,"equired") ==0))
              {nombre =7;
               while (ch != ':')
              {ch=getc(yyin);
               nombre++;}
       ch=getc(yyin);
       nombre++;
       indicee =indicee + nombre; }
      else
          {putc(ch,resultat);
           fputs(ch_c,resultat);
           ch = getc(yyin);
           indicee +=8;}
          }/* end if */

      putc(ch,resultat);} /* end while */
```

```
        fprintf(resultat,"\n");  /* fin de la classe */
        fclose(resultat);
        traite =0;
        fseek(yyin,laposition,0); }
      vpFlag = 0;
      childTab[1] = insertElement("", "", NULL, 0, comment,0,0,0,0,0,0);
      $$ = insertElement("", "", childTab, 0, "",0,0,$1->start,0,0,0);}


    /*********************view**********/
    | view_declaration
      {printf(" EXTERNAL DEFINITION 6  \n");
       createChildTab(); childTab[0] = $1;
     $$ = insertElement("", "", childTab, 0, "",0,0,$1->start,0,0,1);}

    | linkage_specifier function_declaration     /* C++, not ANSI C*/
      {createChildTab(); childTab[0] = $1;  childTab[1] = $2;
       childTab[2] = insertElement("", "", NULL, 0, comment,0,0,0,0,0,0);
       childTab[3] = insertElement("", "xEnd\n",NULL, 0, "",0,0,0,0,0,0);
       $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}

    | linkage_specifier function_definition      /* C++, not ANSI C*/
      {createChildTab(); childTab[0] = $1; childTab[1] = $2;
       childTab[2] = insertElement("", "", NULL, 0, comment,0,0,0,0,0,0);
       childTab[3] = insertElement("", "xEnd\n",NULL, 0, "",0,0,0,0,0,0);
       $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}

    | linkage_specifier declaration              /* C++, not ANSI C*/
      {createChildTab(); childTab[0] = $1; childTab[1] = $2;
     if (currentViewpoint == NULL)       /*******viewpoints*******/
         insertField("","","", $2->start, charPos, isaCharPos);
      childTab[2] = insertElement("", "", NULL, 0, comment,0,0,0,0,0,0);
       $$ = insertElement("", "",childTab, 0, "",0,0,$2->start,0,0,0);}

    | linkage_specifier '{' translation_unit '}' /* C++, not ANSI C*/
      {createChildTab(); childTab[0] = $1; childTab[1] = $3;
       $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}
    ;


new_function_definition:
       identifier_declarator   compound_statement
       {printf ("NEW_FUNCTION_DEFINITION: 1 \n");
      createChildTab(); childTab[0] = $1; childTab[1] = $2;
        $$ = insertElement("", "",childTab, 0, "",$1->nameNode,0,$1->start,0,0,0);}
    | declaration_specifier declarator compound_statement /* partially C++ only */
      {createChildTab(); childTab[0] = $2; childTab[1] = $1; childTab[2] = $3;
     temporaryMethod = createInsertMethod("",$1->typeNode,"",0,0,0,0,NULL,isaType);
     $$ = insertElement("", "",childTab, 0, "",0,$1->typeNode, $1->start,0,0,0);
     }

    | type_specifier  declarator  compound_statement /* partially C++ only */
```

```
        { temporaryMethod = createInsertMethod("", $1->typeNode,"",0,0,0,0, NULL,isaType);
        createChildTab(); childTab[0] = $2; childTab[1] = $1; childTab[2] = $3;
          $$ = insertElement("", "",childTab, 0, "",0,0, $1->start,0,0,0);}

      | basic_type_name declarator compound_statement /* partially C++ only */
        {printf ("NEW_FUNCTION_DEFINITION: 4 \n");
        methodtype = $1->tokenId;
        if (currentVPMethod != NULL)
          currentVPMethod->typeMethod = $1->tokenId;

        if ((currentVPParameter != NULL  ) && (currentVPMethod != NULL ))
          currentVPMethod->parameterMethod = currentVPParameter;
        /****** association des variables a la methode *****/
      if ((currentVPVariable != NULL  ) && (currentVPMethod != NULL ))
        currentVPMethod->variableMethod = currentVPVariable;

       createChildTab(); childTab[0] = $2; childTab[1] = $1; childTab[2] = $3;
       if (temporaryMethod != NULL)
       temporaryMethod = createInsertMethod("", $1->tokenId,"",0,0,0,0,NULL,isaType);
       $$ = insertElement("", "",childTab, 0, "",0,0, $1->start,0,0,0);}
;
constructor_function_in_class:
      declaration_specifier   constructor_parameter_list_and_body
        {methOrVar = 0;
         createChildTab(); childTab[0] = $1; childTab[1] = $2;
         $$ = insertElement("", "",childTab, 0, "",0,0, $1->start,0,0,0);}

      | TYPEDEFname   constructor_parameter_list_and_body
        {methOrVar = 0;
         createChildTab(); childTab[0] = $2;
         if (currentViewpoint !=NULL)
          {printf("on acree une methode 3 \n");
           currentVPMethod = createMethod($1->name,"Constructor",currentViewpoint-
>nameViewpoint,charPos - strlen ($1->name), require);
            if (currentVPParameter  != NULL )
              currentVPMethod->parameterMethod = currentVPParameter;
            }

        else {
       temporaryMethod = createInsertMethod($1->name, "","",0,0,required, provided,
requiredList,isNull);
       temporaryMethod = createInsertMethod("", "Constructor","",0,0,0,0, NULL,isaType);
       temporaryMethod = createInsertMethod("","","",0,0,0,0,NULL,identifyMethod);}
        $$ = insertElement("xMethod:","",childTab,0,"",0,0,$1->startPos,0,$2->parite,0);
      }
      ;

declarator:
      identifier_declarator
        { /* association de la variable currentTestVariable a la  methode
          ou son association a la liste de variable */
        if ((currentViewpoint != NULL) &&
```

```
          (currentVPMethod != NULL) &&
          ( currentTestVariable != NULL)  &&
          (beginingDefinition ==1))
         createVariable(currentTestVariable->nameVar,
                                    currentTestVariable->typeVar,
                                    currentViewpoint->nameViewpoint,
                                    charPos - strlen(currentTestVariable->nameVar));
         /**** creation d'une variable dans un point de vue ******/
        if ((currentViewpoint != NULL) &&
           (currentVPMethod == NULL) &&
           ( currentTestVariable != NULL) /*&& (beginingDefinition !=1) */
           )

      currentVar = createVar(currentTestVariable->nameVar,
                        currentTestVariable->typeVar,
                        currentViewpoint->nameViewpoint,
                        NULL,
                        charPos - strlen(currentTestVariable->nameVar),
                        require);
       currentTestVariable = NULL;
       createChildTab(); childTab[0] = $1;
       $$ = insertElement("", "",childTab, 0, "",0,0,0,0,$1->parite,0);
      }

      | typedef_declarator
       {printf("DECLARATOR 2 \n");
      createChildTab(); childTab[0] = $1;
        $$ = insertElement("", "",childTab, 0, "",0,0,0,0,0,0);}
      ;

identifier_declarator:
      unary_identifier_declarator
       {createChildTab(); childTab[0] = $1;
        $$ = insertElement("", "",childTab,0, "",$1->nameNode,0,$1->start,0,$1->parite,0);}

      | paren_identifier_declarator
      {createChildTab(); childTab[0] = $1;
       isDeclarationMethOrVar = 0; methOrVar = 1;
        if (currentViewpoint != NULL)
         createtest($1->nameNode,
                        currenttype,
                        currentViewpoint->nameViewpoint,
                        charPos - strlen($1->nameNode));

      nodeOfDataVar=createInsertDataMem($1->nameNode,"",NULL,0,0,"",required, provided,
requiredList,isNull);
      $$=insertElement("xVariable:","",childTab,0,"",$1->nameNode, 0,0,0,$1->parite,$1-
>isViewpoint);
      };

postfix_identifier_declarator:
    paren_identifier_declarator  postfixing_abstract_declarator
```

```
    { if (isDeclarationMethOrVar == 1) /* if it is a method */
      if ((vpFlag ==1) &&
         (currentViewpoint != NULL) &&
         (currentVPMethod == NULL)) {
          currentVPMethod = createMethod($1->nameNode, methodtype,
                                currentViewpoint->nameViewpoint,$1->start,require);
                    }

         else
        if (isaDestructor != 1)  /* not a destructeur & not related to a VP*/
          {
           if (yesScope ==0)
             {temporaryMethod=createInsertMethod($1->nameNode,"","",0,0,required,
                                      provided,requiredList, isNull);
              yesScope = 1; }
           else
              temporaryMethod=createInsertMethod($1->nameNode,"","",0,0,required,
                                        provided,requiredList,isNull);
           temporaryMethod = createInsertMethod("","","",0,0,0,0, NULL,identifyMethod);
           }

         else  /* it is a Destructor */
          {
           temporaryMethod=createInsertMethod($1->nameNode,"","",0,0,required,
provided,requiredList,isNull);
           temporaryMethod=createInsertMethod("","Destructor","",0,0,0,0,NULL,isaType);
           temporaryMethod = createInsertMethod("","","",0,0,0,0,NULL,identifyMethod);
           isaDestructor = 0;}
            else /* not a method, but it is variable */

        if ((vpFlag ==1 ) && (currentVPMethod == NULL))   /**********viewpoints ************/

           currentVar=createVar($1->nameNode,currenttype,
                        currentViewpoint->nameViewpoint,NULL,
                                charPos-strlen($1->nameNode),require);
         nodeOfDataVar=createInsertDataMem($1->nameNode,"","",0,0,"",required,
provided,requiredList,isNull);
          createChildTab(); childTab[0] = $1; childTab[1] = $2;
         $$=insertElement("xMethod:","",childTab,0,"",$1->nameNode,0,$1->start,0,$2->parite,0);

       }
;
tag_name_viewpoint:
       IDENTIFIER
       {$$=insertElement("","",NULL,$1,"",$1->name,0,charPos-strlen($1->name), 0,$1-
>name,0,0);}
       | TYPEDEFname
         /* ici on rencontre une declaration Si c'est un point de vue creer le
        si c'est une classe aussi elle est creee la definition d'un nouveau
        point de vue se fait au niveau currentBloc =0 */

         {if (vpFlag ==1) /*&& (currentBloc ==0))  */
```

```
          {require =0;
          currentViewpoint =  createViewpoint($1->name, charPos-strlen($1->name));
           if (currentViewpoint == NULL)
               printf("ERREUR, le point de vue n'est pas cree ou existe deja \n");
           currentViewpoint = lookupVp($1->name); }
       createChildTab();
       childTab[0] = insertElement( "", "", NULL, 0, "", $1->type,0,0,0,0,0);
       $$=insertElement("","",childTab,$1,"",$1->name,0,
        charPos-strlen($1->name),$1->name,0,1);
}
      ;

tag_name_view:
      IDENTIFIER
      {printf("TAG_NAME_VIEW 1 \n");
       $$=insertElement("","",NULL,$1,"",$1->name,0,charPos-strlen($1->name), 0,$1-
>name,0,0);}
      | TYPEDEFname

       /* ici on rencontre une declaration de vue la definition d'un nouveau
       point de vue se fait au niveau currentBloc =0 */
       {printf("TAG_NAME_VIEW 2 \n");
       createChildTab();
       childTab[0] = insertElement( "", "", NULL, 0, "", $1->type,0,0,0,0,0);
       $$=insertElement("","",childTab,$1,"",$1->name,0,
charPos-strlen($1->name),$1->name,0,1);
}
      ;


tag_name:
      IDENTIFIER
       {$$=insertElement("","",NULL,$1,"",$1->name,0,charPos-strlen($1->name) ,0,$1-
>name,0,0);}
      | TYPEDEFname
       { if (currentBloc == 0)
         { ptrView = "Private";
          insertField($1->name,"","", 0, 0, isaClass);
         }
       createChildTab();
       childTab[0] = insertElement( "", "", NULL, 0, "", $1->type,0,0,0,0,0);
        $$=insertElement("","",childTab,$1,"",$1->name,0, charPos-strlen($1->name),$1-
>name,0,0);

}
      ;

scope_opt_identifier:
       IDENTIFIER
       {$$ = insertElement("", "", NULL, $1, "",$1->name,0,ftell(yyin) - (strlen($1->name) -1),$1-
>name,0,0); }
      | scope IDENTIFIER  /* C++ not ANSI C */
```

```
        {nameMethod = $2->name; yesScope =0;
       if(currentBloc != 0)      /* !=0 when We have a call of method X::f()*/
      { ptStructCall = createCall($2->name,"","",isNull);
       ptStructCall = createCall("","", $1->nameNode,getInstance);
        ptStructCall = createCall("","",$1->nameNode,listParameter);
      }
       createChildTab();
       childTab[0]=insertElement("callMethod:","",NULL,0,"",0,0,0,0,0,0);
       childTab[1]=insertElement("xtype:","",NULL,0,"",0,0,0,0,0,0);
       childTab[2]= $1;
         $$ = insertElement("", "",childTab, $1, "",$2->name,0,0,0,0,0);}
      ;

%%
extern FILE *yyin;

yyerror(string)
char*string;

{printf("Erreur a la ligne: %d value: %s  token: %s\n",lineno,
yylval.value,yyname[yychar]);

}

main(argc, argv)
int argc;
char **argv;
{                       /* FILE *yyin; */
      yyin = fopen(argv[1],"r");
      filename = malloc(strlen(argv[1]));
      headfile = malloc(strlen(argv[1]));
      strncpy(headfile, argv[1],strlen(argv[1]) -3);
      strcat(headfile,".h");
      strncpy(filename, argv[1],strlen(argv[1]) -3);
      strcat(filename,".c");
      resultat = fopen(filename, "w");
      fprintf(resultat,"/*****************************************************/ \n");
      fprintf(resultat,"/* fichier contenant le resultat de notre analyse */ \n");
      fprintf(resultat,"/*****************************************************/ \n");
      fprintf(resultat,"#include \"entete.h\" \n");
      fprintf(resultat,"#include \"%s\" \n", headfile);
      fclose(resultat);
                  while(!feof(yyin))
        { printf("YYIN ouvert!\n");
          start_file_bloc();
                     if (yyparse() != 0) {
                        printf("parse non reusssi\n");
                         free_hash_table();
                         freeChildTab();
                         deleteTree(parseTree);}

                      else
```

```
                    {
                printf("\n------------------ Begin ------------------ \n");
                printViewpoint(ptrListViewpoint);
              printView(ptrListView);
              printf("\n------------------ End ------------------ \n");} }
              viewDefinition(yyin, headfile);

              if (ptrBeginOfClass != NULL)
               {  freeStructClass();
                   free_hash_tab();
                  free_hash_table();
                   freeChildTab();
                   deleteTree(parseTree);}
               fclose(yyin);

}
```