Université Joseph Fourier

U.F.R Informatique &
Mathématiques Appliquées

Institut National Polytechnique
de Grenoble

ENSIMAG

I . M . A . G .

ECOLE DOCTORALE
MATHEMATIQUES ET INFORMATIQUE

DEA D'INFORMATIQUE :
SYSTEMES ET COMMUNICATIONS

Projet présenté par :

Humberto CERVANTES

Analyse de dépendances et Découpe dans un Logiciel de Grande Taille

Effectué au laboratoire : L.S.R , équipe Adèle

Date : 19 Juin 2000

Jury :
Joëlle Coutaz
Christine Collet
Florence Maraninchi
Jean-Marie Favre

R E S U M E

La connaissance des dépendances qui existent entre les éléments qui composent un logiciel est une information indispensable pour aider à sa compréhension et qui peut permettre de réaliser entre autres des modifications de façon contrôlée, surtout lorsque la taille de ce logiciel est très grande.

Le travail que nous présentons est réalisé dans le cadre d'une collaboration entre le laboratoire LSR, équipe Adèle, et le partenaire industriel Dassault Systèmes (DS). Ce dernier a développé une extension du langage C++ appelée l'Object Modeler pour construire des logiciels complexes, sur lesquels nous cherchons à proposer des moyens pour étudier les dépendances. L'Object Modeler possède certaines caractéristiques qui rendent difficile cette étude, en particulier le fait que les dépendances sont souvent créées de façon dynamique.

L'analyse des dépendances dans les logiciels peut être réalisée au moyen de graphes de dépendance et d'une technique appelée « Découpe » (*slicing*) qui permet d'isoler des parties de ces graphes à partir de critères spécifiques de manière statique ou dynamique. Cependant, les caractéristiques introduites par l'Object Modeler ne peuvent pas être représentées avec les types de graphes de dépendance qui ont été proposés jusqu'à présent.

Pour combler cette difficulté, notre travail présente un type de graphe de dépendances orienté à la représentation de programmes construits avec l'Object Modeler. Nous discutons aussi l'application de la technique de découpe dans ce types de graphe et finalement nous proposons certaines applications qui peuvent être réalisées à partir de cette information.

CHAPITRE 1 : Introduction

1.1 Problématique

Le Génie Logiciel est devenu un des aspects de l'informatique auquel on consacre un nombre croissant de recherches. Ceci est dû en partie au fait que depuis un certain nombre d'années, la complexité des logiciels augmente. Les logiciels atteignent actuellement des tailles très importantes, et leur croissance se poursuit avec des contraintes de temps toujours plus strictes.

Cette situation à provoqué, chez la plupart des compagnies de développement de logiciels, de grandes difficultés à réaliser une planification formelle du processus de création du logiciel. Ces compagnies ont été poussées à suivre plutôt une tendance de résolution des problèmes au fur et a mesure de leur apparition. Le développement des logiciels est réalisé tout en ayant perdu la vision globale du produit, qui d'un autre côté continue de croître de manière désorganisée.

Malheureusement il s'avère souvent impossible d'arrêter le processus d'évolution pour essayer de reconstruire un logiciel problématique d'une manière plus efficace, et il devient plutôt nécessaire de choisir une stratégie différente qui permette de mieux orienter la poursuite du développement du logiciel.

Le premier pas a suivre pour envisager ceci est inévitablement celui de comprendre l'existant. Cependant ceci peut être une tache compliquée, particulièrement lorsque la liaison entre la spécification et l'implémentation n'est pas bien établie (ce qui a lieu quand la spécification et l'implémentation ne sont pas synchronisés).

1.2 Contexte de l'étude

L'étude que nous allons effectuer est réalisée autour d'un logiciel réel de CFAO (Conception et Fabrication Assistée par Ordinateur) de nom CATIA, construit par Dassault Systèmes (DS).

Ce logiciel fournit une étude de cas intéressante du fait que son degré de complexité est très élevé et qu'il est développé suivant un modèle qui présente un certain nombre de caractéristiques particulières.

Des contraintes au niveau de l'accès à l'information ont contribué fortement à réaliser certains choix sur le niveau d'abstraction dans lequel nous avons décidé de faire notre étude, car le travail a du être effectué sans le code source car celui ci n'était pas disponible. La démarche suivie dans ce DEA à donc été de faire des propositions générales et extensibles pouvant être complétées postérieurement par des informations complémentaires.

Dans leur ensemble, les propositions restent à être validées au moyen de tests pratiques.

1.3 Objectifs

Ce travail a pour but de présenter un modèle de représentation des dépendances à l'intérieur des logiciels construits suivant le modèle de DS et de proposer un certain nombre d'applications pouvant être réalisées à partir de l'application de la technique de découpe sur cette représentation.

1.4 Plan du document

Cette étude est divisée en trois parties principales :

- La première partie consiste en un état de l'art qui présente l'évolution des graphes de dépendance et de la technique de découpe.

  Nous allons décrire premièrement divers types de graphes de dépendance suivant leur évolution selon à partir du type de programme qu'ils représentent.

  Nous présenterons ensuite la technique de découpe, appliquée aux graphes de dépendance présentés auparavant et nous décrirons aussi d'autres types de découpes ainsi que leurs applications.

- La deuxième partie introduit l'Object Modeler de Dassault Systèmes.

  Nous justifions le besoin d'étudier les dépendances dans les logiciels construits avec l'OM, à partir des applications qu'il serait utile de réaliser pour permettre de résoudre les problèmes mentionnés auparavant. Ces applications nécessitent d'une représentation des dépendances sous forme de graphes et de la réalisation des coupes sur celle ci.

  Nous introduirons ensuite l'ensemble d'éléments qui caractérisent l'Object Modeler et qui seront ensuite utilisés dans la réalisation des graphes de dépendances.

- La troisième partie présent le graphe de dépendances de l'Object Modeler et discute sur divers aspects relatifs à celui-ci, comme son obtention ou la possibilité de le visualiser.

  Nous présenterons ensuite l'application de la technique de découpe dans ce type de graphe en particulier et les diverses applications qui peuvent être obtenues à partir de cette technique.

Nous donnerons pour terminer des conclusions sur ce travail et nous discuterons des perspectives qui se présentent à l'issue de celui ci.

CHAPITRE 2 : Graphes de dépendance et Découpe de programmes

Les Graphes de dépendance permettent de réaliser la représentation des programmes sous forme de graphe et la technique de découpe de programmes permet d'isoler des parties de ces graphes suivant différents critères particuliers. Dans ce chapitre nous allons présenter ces deux concepts ainsi que leurs applications.

2.1 Graphes de dépendance

Il existe plusieurs définitions sur ce que sont les graphe de dépendances. Ces définitions dépendent en partie de l'utilisation qu'on cherche à leur donner, mais il s'agit pour la plupart d'entre elles de variations autour d'un thème introduit initialement en 1972 [1].

Un des objectifs initiaux des graphes de dépendance était de construire une représentation intermédiaire des programmes en vue de réaliser des optimisations sur ceux-ci ou de trouver des moyens de les rendre parallèles [1]. On retrouve dans l'ensemble des définitions apparues postérieurement la caractéristique commune d'introduire des représentations explicites pour les dépendances de contrôle ainsi que pour celles de données.

Les graphes de dépendance étant une représentation des programmes, ils ont évolué en suivant les modèles de programmation existants. Commençant par la représentation des instructions à l'intérieur de programmes consistant d'une unique fonction (programmes monolithiques), ils ont été étendus postérieurement pour permettre de modéliser d'autres caractéristiques plus complexes, telles que les appels de fonctions, les passages de paramètres, les liens dynamiques et plus récemment les objets et leurs diverses caractéristiques [2].

Ce chapitre présente plus en détail cette évolution pour donner les bases permettant de proposer postérieurement un modèle particulier au contexte de notre étude.

2.1.1 Le Graphe de Dépendance de Programmes

Le Graphe de Dépendance de Programmes (ou PDG de ses initiales en anglais) est une représentation sous forme de graphe dirigé d'un programme monolithique. Ce graphe est formé par des nœuds et des liens. Les nœuds correspondent à des instructions ou à des expressions [1] [2] et les liens relient les nœuds et représentent deux types de dépendances, celles de données et celles de contrôle. Le PDG est une représentation adéquate des programmes [12]

Les dépendances *de données* ont lieu entre deux instructions dans le cas où une variable qui est définie dans la première d'entre elles est référencée dans la deuxième et ceci sans qu'il y ait entre ces deux instructions d'autres redéfinitions de la variable.

Par exemple :

```
X=1+2;     (i1)
Y=X+1;     (i2)
```

Dans cet exemple, la variable *X* est définie dans l'instruction i1 puis référencée dans l'instruction i2, il existe donc une dépendance de données entre les deux instructions. Ceci implique que l'ordre d'exécution de i1 et i2 ne peut pas être changé.

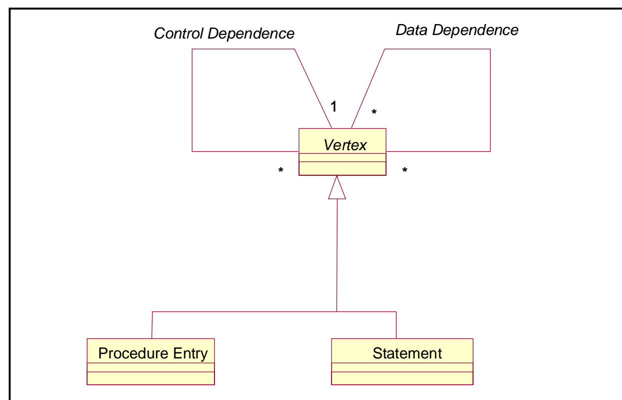
Une dépendance *de contrôle* apparaît entre une instruction et un prédicat dont la valeur contrôle de manière directe l'exécution de cette instruction, par exemple :

```
if(X==1)  (i1)
     Y=2; (i2)
```

Dans ce cas l'instruction i2 dépend du prédicat `X==1` car le résultat de l'évaluation de celui ci déterminera si elle est exécutée ou non.

Pour représenter des programmes "monolithiques", c'est à dire formés par une fonction unique, deux types de nœuds sont nécessaires : ceux qui représentent les instructions et ceux qui représentent le début de la fonction.

Le modèle correspondant à ceci exprimé dans la notation UML est le suivant :



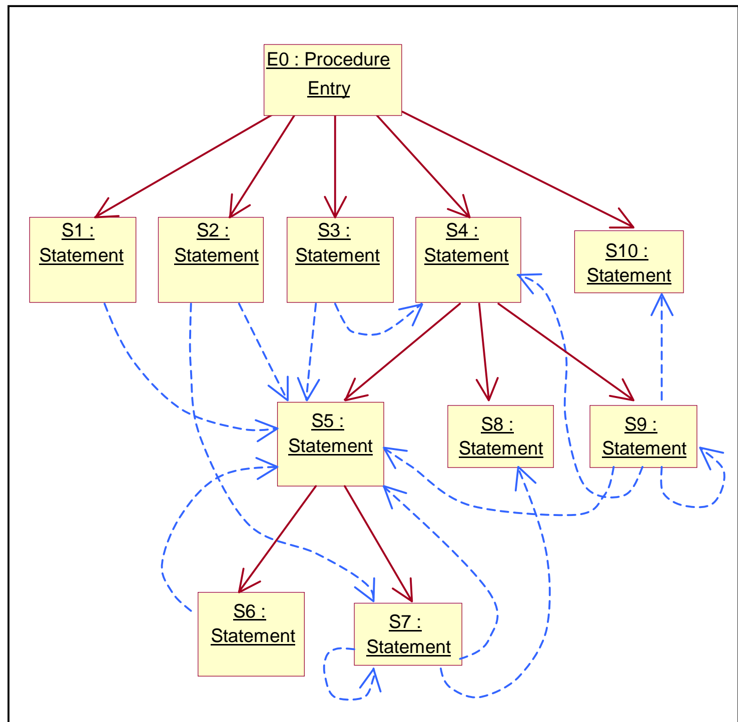
Ce diagramme présente un méta modèle des graphes. Le graphe de dépendance de programmes sera un diagramme d'instances dans lequel seulement les nœuds instruction (*statement*) et entrée (*procedure entry*) apparaîtront car la classe nœud (*vertex*) est une classe abstraite.

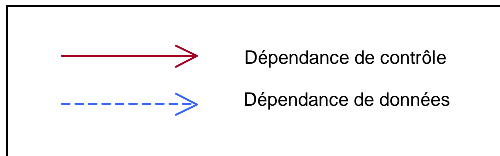
Les cardinalités dans les liens de contrôle sont de 1 vers plusieurs du fait qu'une instruction peut contrôler l'exécution d'un ensemble d'instructions mais qu'inversement une instruction n'est contrôlée que par une seule instruction. En fait, cette relation forme un arbre correspondant à l'emboîtement des instructions de contrôle.

Les cardinalités des liens de données sont multiples des deux cotés car une seule instruction peut référencer plusieurs variables et car une plusieurs instructions peuvent dépendre de l'instruction où est définie une variable.

La figure ci-dessous présente le graphe de dépendance de programmes correspondant au programme suivant :

```
E0:     main()
        {
S1:         int a=0;
S2:         int b=0;
S3:         int x=0;
S4:         while(x<3)
            {
S5:             if((a+b)<x)
S6:                 a=3;
                else
S7:                 b=b+1;
S8:             printf("%d",b);
S9:             x=x+1;
            }
S10:        printf("%d",x);
        }
```





Le schéma ci dessus correspond à un diagramme d'instance obtenu à partir du modèle UML. Par souci de clarté, des flèches ont été utilisées pour représenter la nature des liens entre instances, et les instances de nœuds ont un nom qui correspond aux instructions du programme.

2.1.2 Le Graphe de Dépendance de Système

Le PDG que nous avons décrit antérieurement introduit les principes concernant l'étude des dépendances dans un programme formé d'une fonction unique. Cependant ce cas est extrêmement limité et éloigné des problèmes réels. Une extension de ce type de graphe, le Graphe de Dépendance de Système (ou SDG), permet la représentation de programmes constitués de procédures multiples, ce qui permet de représenter des programmes plus complexes et plus rapprochés de la réalité [4].

Le graphe de dépendance de système reprend les caractéristiques du PDG et rajoute en plus certains nœuds et liens permettant de représenter l'appel de procédures, ainsi que le passage de paramètres.

Les nœuds rajoutés sont:

- Nœud d'appel de procédure (*procedure call*) : A tous les points où un appel de procédure est effectué. Un appel de procédure est un cas particulier d'instruction.

- Nœuds de paramètres *formels* : Ces nœuds représentent les paramètres de la procédure, soit ceux d'entrée (correspondant aux valeurs reçues) soit ceux de sortie (correspondant aux valeurs de retour).

- Nœuds de paramètres *effectifs* : Les nœuds de ce type sont reliés aux nœuds d'appel de procédure et représentent les valeurs particulières que prennent les paramètres (soit à l'entrée soit à la sortie) qui sont envoyés à une procédure. Pour chaque paramètre *formel* d'une procédure il doit exister un paramètre *effectif* relié au nœud d'appel de cette procédure.

Les liens rajoutés sont :

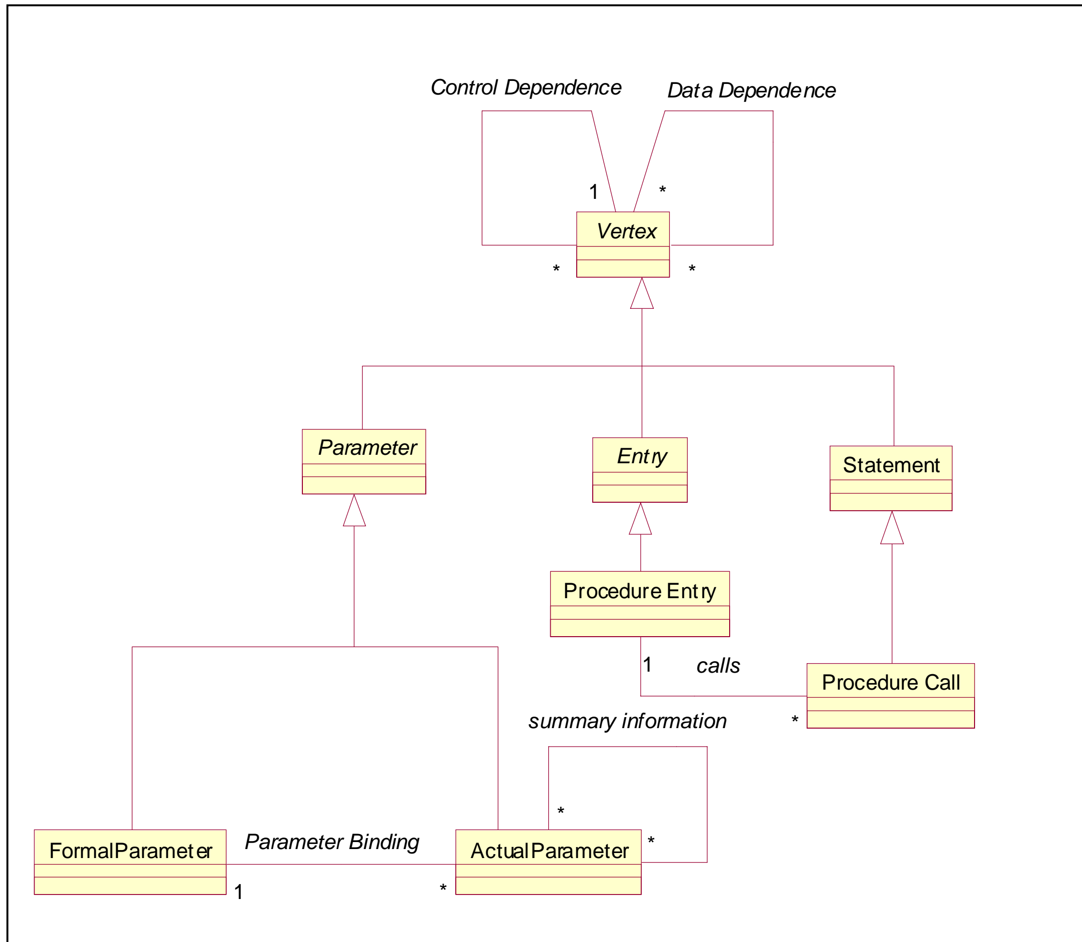
- Lien de liaison de paramètres (*parameter binding*) : Ces liens relient les nœuds de paramètres formels et actuels.

- Liens de résumé (*summary edges*) : Ces liens relient les paramètres actuels de sortie et d'entrée et représentent les dépendances transitives dues à l'appel de la procédure. Ceci signifie qu'il y aura un lien entre un paramètre actuel d'entrée et un paramètre de sortie si la valeur du paramètre de sortie dépend de celle du paramètre d'entrée.

- Liens d'appel (*call edge*) : Ces liens relient les nœuds d'appel de procédure avec les nœuds d'entrée des procédures.

Les éventuelles variables globales sont représentées sous forme de paramètres envoyés aux méthodes qui y accèdent.

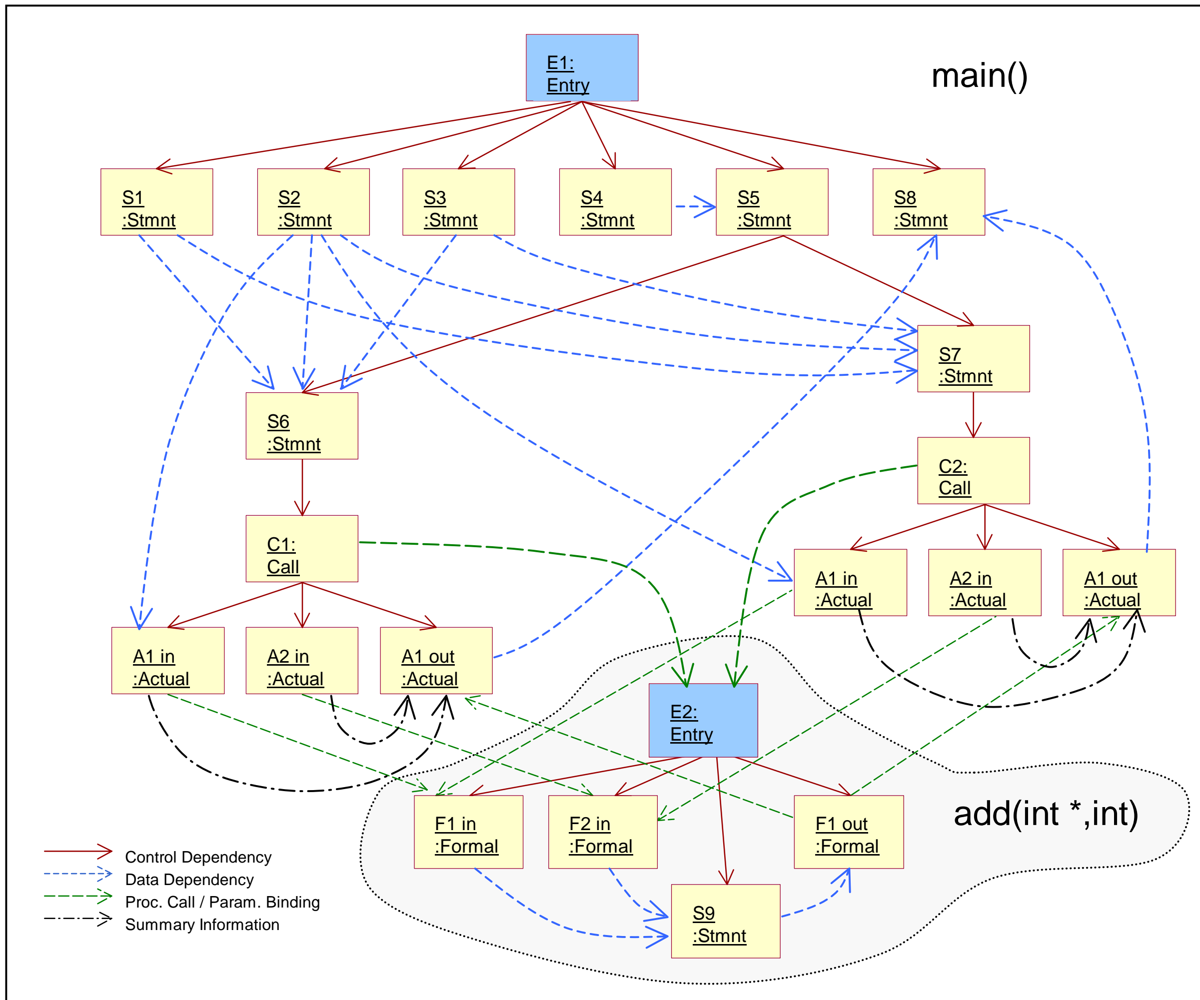
Le modèle UML qui représente le SDG est le suivant :



Nous montrons à la suite un exemple de programme ainsi que son SDG correspondant (dans celui ci les noms des types ont été abbreviés pour améliorer la lisibilité) :

```
E1:     main()
        {
S1:        int floor=5;
S2:        int cur_floor=1;
S3:        int top_floor=10;
S4:        int cur_dir=UP;
S5:        if(cur_dir==UP)
S6:           while((cur_floor!=floor)&&(cur_floor<=top_floor))
C1:              add(&cur_floor,1);
           else
S7:           while((cur_floor!=floor)&&(cur_floor>0))
C2:              add(&cur_floor,-1);
S8:        printf("%d",cur_floor);
        }

E2:     add(int *a,int b)
        {
S9:        *a=*a+b;
        }
```



Il est intéressant de noter que même pour un programme simple, le nombre de liens nécessaires à sa représentation est assez important.

2.1.3 Le Graphe de Dépendance Orienté Objet

Les graphes de dépendance pour des programmes construits selon le paradigme Orienté Objet sont récents et il existe peu de travaux relatifs à ce domaine. Cependant ceux qui ont traité ce sujet [2] [3] [15] coïncident en particulier sur le besoin de suivre un des principes importants de la philosophie qui existe dans le modèle Objet, c'est à dire celui de la réutilisation de l'information.

Les méthodes sont traitées comme des procédures et donc représentées au moyen de graphes de dépendance de système. Les attributs de la classe sont traités comme des variables globales à celle ci et donc passés aux méthodes comme s'il s'agissait de paramètres.

Ci dessous, nous décrivons d'abord un modèle simple permettant de représenter les classes et nous discuterons ensuite la manière de représenter les autres caractéristiques des programmes orientés objet, telles que l'héritage et le polymorphisme.

2.1.3.1 Représentation des classes

Les nœuds qui sont ajoutés au modèle du SDG est le suivant :

- Nœud d'entrée des classes (*class entry*) : Ces nœuds représentent la classe et ils sont reliés aux nœuds d'entrée des méthodes (qui sont des nœuds d'entrée de procédure conventionnels).

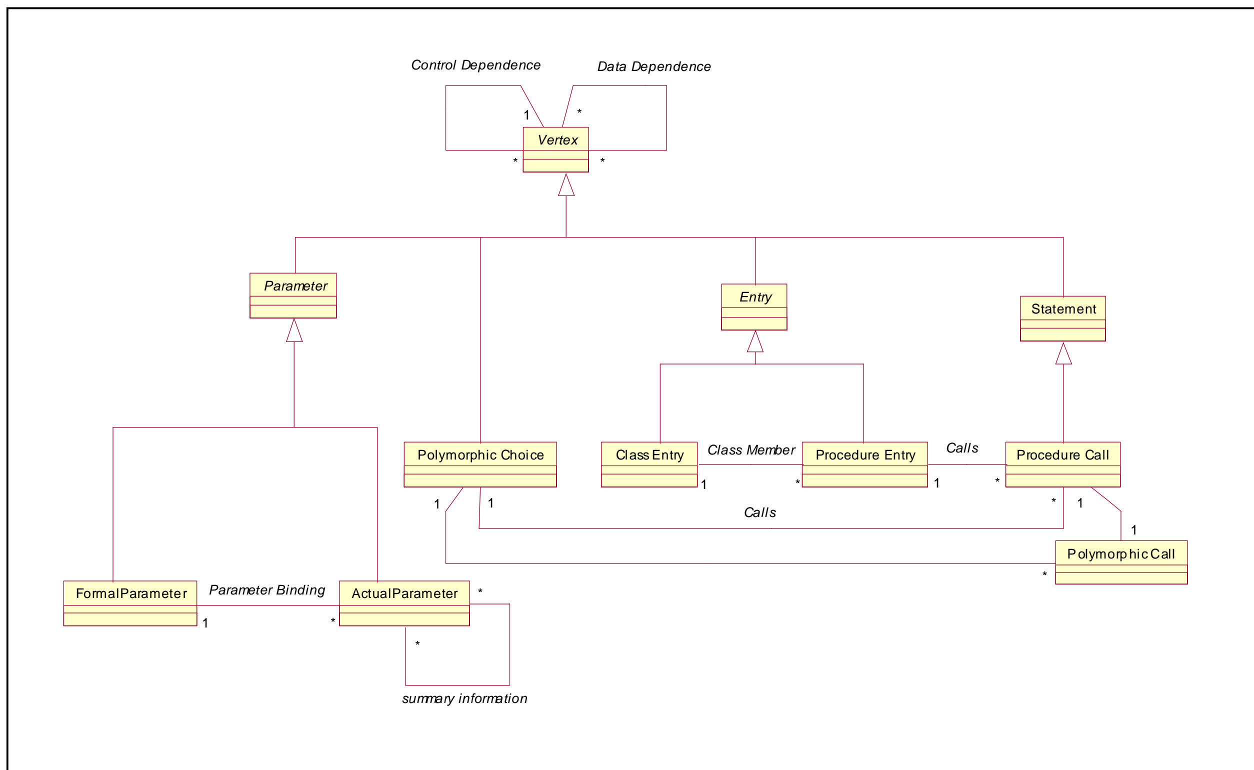
- Nœud de choix polymorphique (*polymorphic choice*) : Ce nœud est introduit lors d'un appel résolu dynamiquement, il représente la sélection d'un appel particulier à partir d'un choix de possibilités multiples (ceci est décrit en détail plus loin dans la section "polymorphisme").

- Nœud d'appel polymorphique(*polymorphic call*) : Ce nœud est relié au celui de choix polymorphique et permet de créer un appel vers les destinations potentielles.

Le lien nouveau par rapport au SDG est :

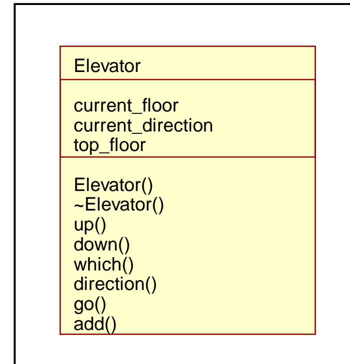
- Liens de membre de classe (*class member*) : Ces liens relient les nœuds d'entrée de classe avec les nœuds d'entrée des méthodes appartenant à la classe.

Le modèle UML représentant ceci est le suivant :



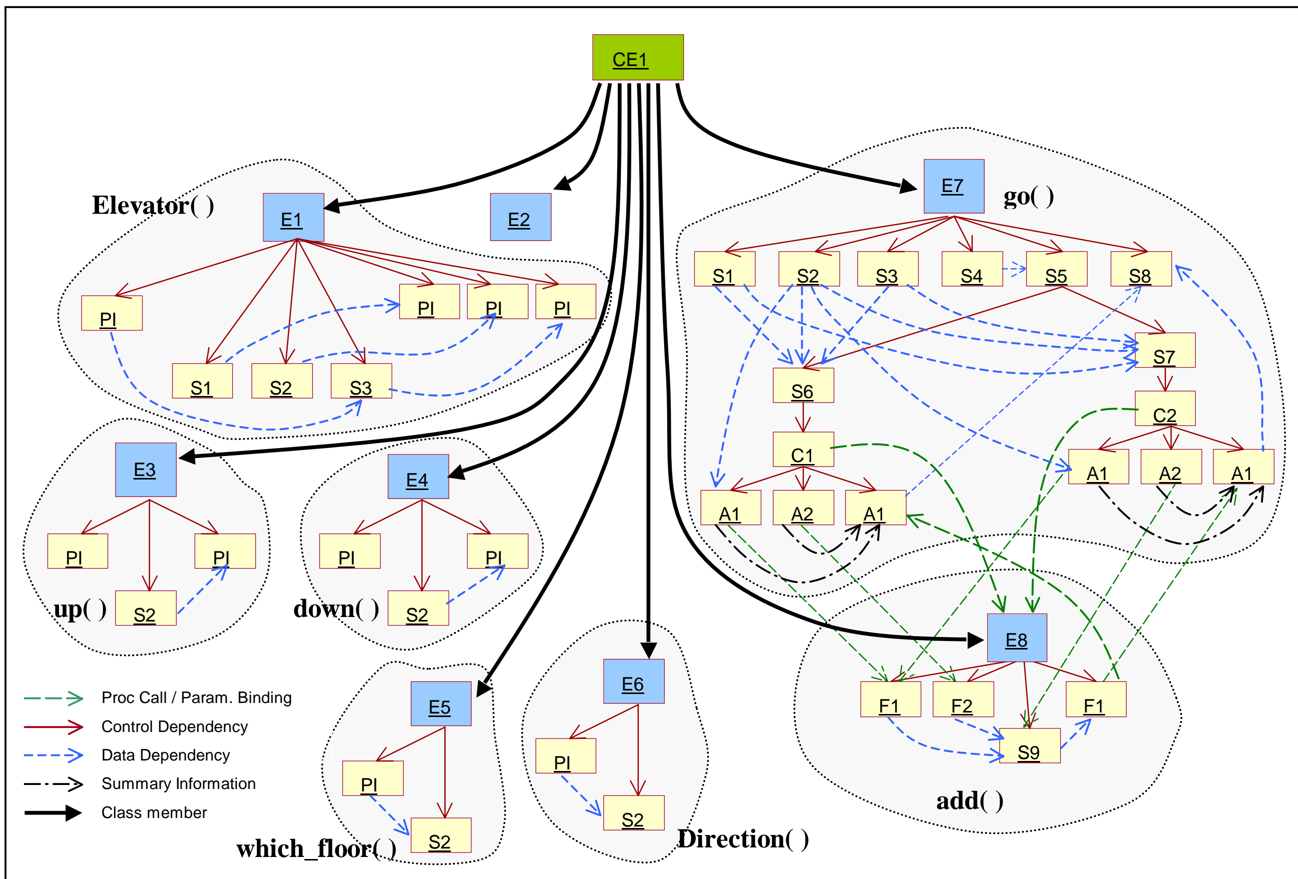
Exemple :

```
   class Elevator
   {
      private:
E8 :     add(int &a,const int &b);
      protected:
         int current_floor;
         int current_direction;
         int top_floor;
      public:
E1 :     Elevator(int);
E2 :     virtual ~Elevator() {}
E3 :     void up();
E4 :     void down();
E5 :     int which_floor();
E6 :     int Direction();
E7 :     virtual void go();
   };
```




```
Elevator ::Elevator(int l_top_floor)
{     current_floor=1;
      current_direction = UP;
      top_floor=l_top_floor;
}
void Elevator ::up()
{     current_direction = UP;
}
void Elevator ::down()
{     current_direction = DOWN;
}
int Elevator ::which_floor()
{     return current_floor;
}
int Elevator::Direction()
{     return current_direction;
}
virtual void go(int floor)
{     if(current_direction == UP)
      { while(current_floor != floor)&&(current_floor <= top_floor))
            add(current_floor,1); }
      else
      { while(current_floor != floor)&&(current_floor > 0))
            add(current_floor,-1); }
}
Elevator ::add(int &a,const int &b)
{     a = a + b ;
}
```

Graphe de dépendance correspondant :



2.1.3.2 Création d'instances

Pour représenter la création d'instances des classes, on utilisera un lien d'appel de procédure vers le constructeur de la classe.

2.1.3.3 Héritage

Pour représenter l'héritage, on crée un graphe pour la nouvelle classe et on relie le nœud d'entrée de cette classe vers les méthodes dont qui sont héritées de la classe de base.

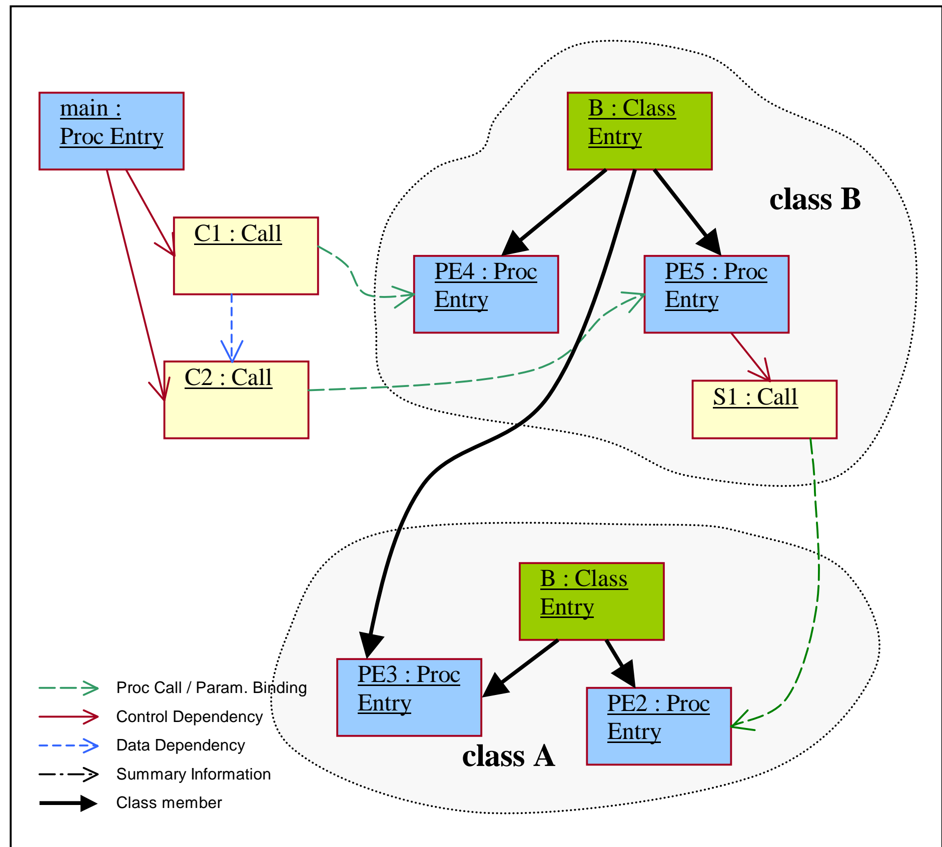
Lorsqu'une méthode de la classe dérivée appelle une des méthodes de la classe de base, on emploie la même représentation que s'il s'agissait d'un appel vers une des méthodes de la classe dérivée (c'est à dire, on ajoute un nœud d'appel ainsi qu'un lien d'appel de procédure).

Exemple :

```
Class A
{
PE1 :   A( );
PE2 :   virtual Method( ) ;
PE3 :   BaseMethod( ) ;
};

class B : public A
{
PE4 :   B( ) ;
PE5 :   Method( )  {
S1 :            A :: Method( ) ;}
};

void main( )
{
C1 :    B object ;
C2 :    object.Method( ) ;
}
```



Dans cet exemple, nous pouvons voir comment un objet de type B est instancé dans la méthode *main*. Ceci est représenté par l'appel en C1 de son constructeur (PE4), et ensuite une de ses méthodes est appelée en C2. Cette méthode, virtuelle, à son tour appelle la méthode définie par la classe de base (en S1).

Nous pouvons aussi remarquer que le nœud d'entrée de la classe B est relié au nœud d'entrée de la méthode BaseMethod() (PE3) car elle est héritée de la classe A. Il n'y a pas de lien vers A::Method() car elle est redéfinie. Les frontières montrent à quelles classes appartiennent les nœuds.

2.1.3.4 Polymorphisme

Une caractéristique importante des langages orientés objet est le polymorphisme qui permet, en temps d'exécution d'appeler une méthode particulière parmi un ensemble de destinations possibles pour l'appel. Cette caractéristique correspond à la liaison dite "dynamique".

Le graphe de dépendance orienté objet permet de représenter les appels polymorphiques vers des méthodes. Ces appels étant résolus dynamiquement, la destination choisie lors de l'appel ne peut être connue de manière statique [14][16].

L'approche choisie pour représenter ceci est de prendre en compte toutes les destinations possibles de l'appel, et ceci à l'aide d'un nœud dit de "choix polymorphique", à partir duquel sont réalisés des appels vers toutes les destinations possibles.

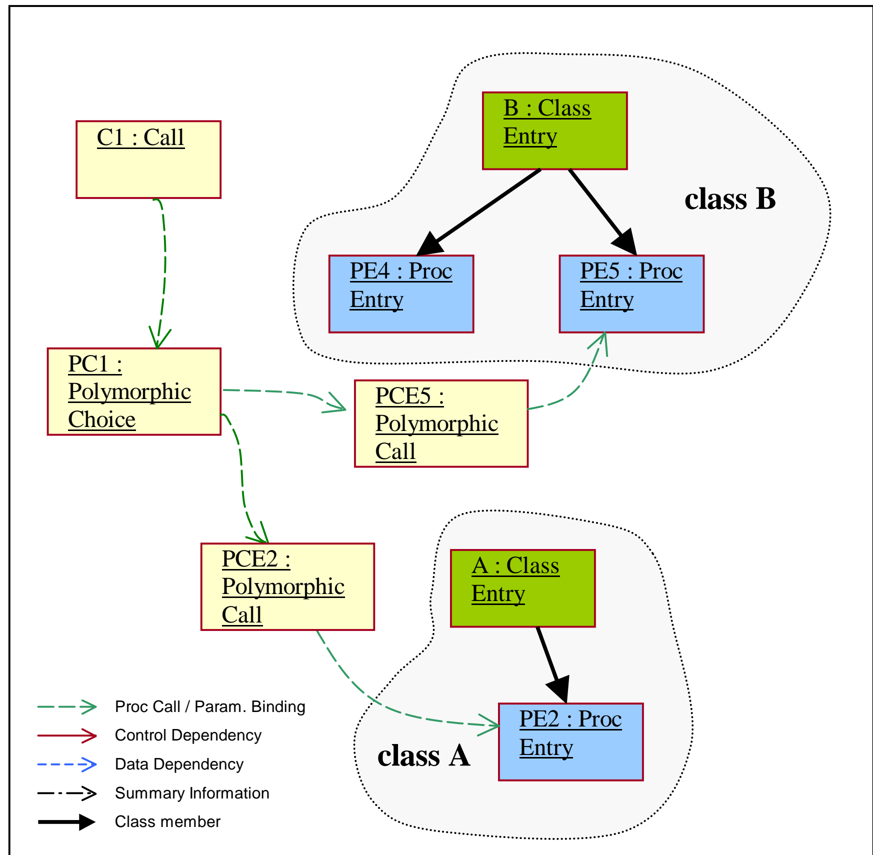
Exemple :

```
Class A
{
PE1 :   A( );
PE2 :   virtual Method( ) ;

};

class B : public A
{
PE4 :   B( ) ;
PE5 :   Method( ) ;
};

void main( )
{
        A *object_ptr ;
S1 :    if(…)
          object_ptr =new A( ) ;
        else
          object_ptr =new B( ) ;
C1 :    object_ptr ->Method( ) ;
}
```



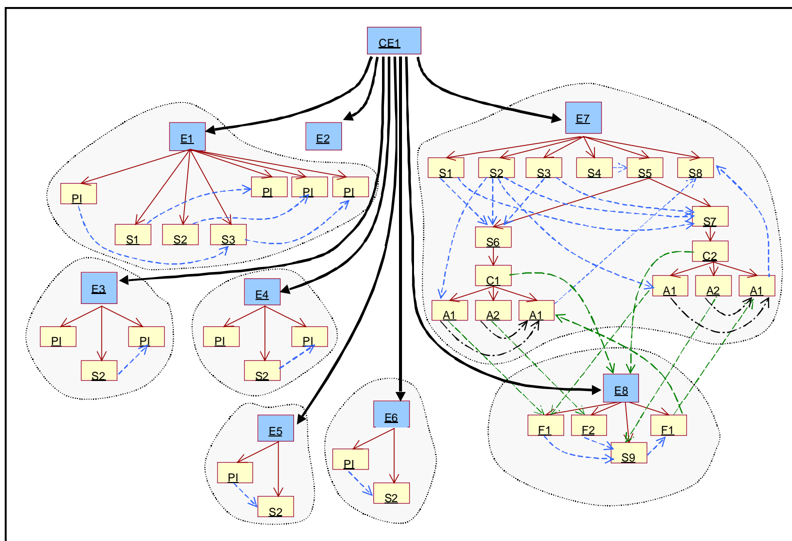
Dans cet exemple, très proche de celui que nous avons montré pour l'héritage, nous pouvons observer que l'appel vers la méthode en C1 sera résolu en temps d'exécution, selon le résultat de la condition S1. Le nœud PC1 de choix polymorphique est donc relié aux deux nœuds d'appel polymorphique qui 'simulent' un appel à la méthode (si nécessaire, ces nœuds seront reliés à des nœuds de paramètres *effectifs*)

2.1.4 Niveaux de granularité

Un point qui est important de noter sur les graphes de dépendance est le fait que le nombre de liens croît très rapidement avec le rajout de caractéristiques nouvelles ainsi qu'avec la taille du logiciel. Ceci est particulièrement vrai par exemple dans le cas des graphes orientés objet lors des appels polymorphiques vers des méthodes.

Il est possible de travailler à des niveaux différents de détail, aussi appelés de *granularité*, pour diminuer la complexité des graphes, par exemple [11]. Ainsi, si nous ne sommes pas intéressés par les dépendances à l'intérieur des méthodes, il est possible de travailler seulement avec les nœuds d'entrée vers celles ci, car ils représentent la méthode dans l'ensemble, c'est à dire à un niveau de granularité plus faible.

Si nous reprenons l'exemple vu auparavant :



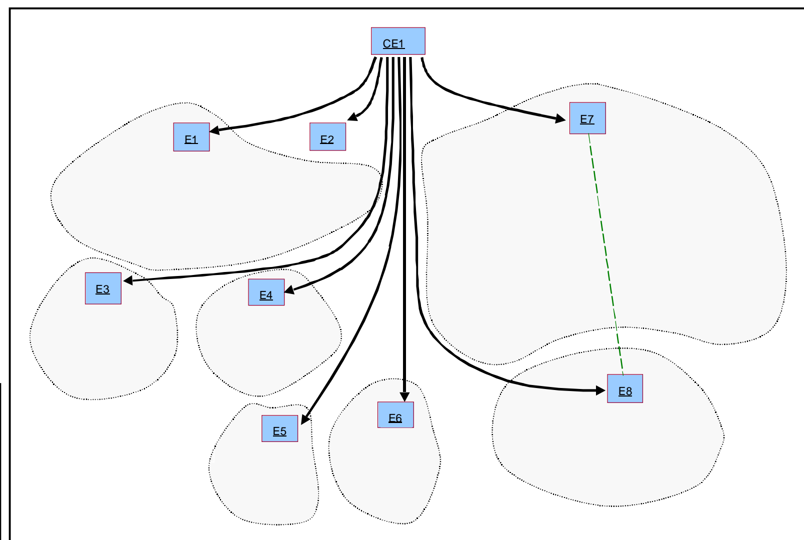
Granularité Faible:

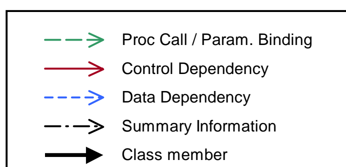
Ce graphe représente une classe avec plusieurs méthodes.

Deux des méthodes dépendent entre elles (E7 et E8), car un appel est réalisé à l'intérieur d'une d'entre elles vers l'autre.

Granularité Forte :

Cette autre vue du même graphe ne prend pas en compte les instructions à l'intérieur des méthodes, cependant la dépendance entre les méthodes doit être conservée.





2.1.5 Synthèse

Nous avons détaillé plusieurs types de graphes de dépendance, qui permettent de créer des représentations de divers types de programmes, depuis les plus 'simples', qui consistent d'une procédure unique, jusqu'aux programmes réalisés selon le paradigme Orienté Objet. Ces représentations ont suivi l'évolution des programmes, et chaque type de graphe reprend les caractéristiques du modèle plus simple qui le précède, en lui rajoutant un certain nombre de liens et de nœuds nouveaux.

Lors de la description sur la manière de représenter les caractéristiques des programmes construits selon le paradigme Orienté Objet, nous avons décrit entre autres la représentation des appels polymorphiques. Ces appels ne peuvent être résolus de manière statique, et doivent être représentés en suivant toutes les destinations possibles où ils peuvent aboutir. Ceci pourrait sembler peu précis, cependant, le graphe de dépendances ne cherche pas à résoudre le problème des choix multiples lors d'un appel.

Nous avons décrit finalement la possibilité de travailler à divers niveaux de granularité, ceci est utile pour réduire le degré de complexité d'un graphe ou simplement parce que nous ne sommes pas intéressés par une analyse à un niveau très fin de granularité.

Le tableau suivant fait une synthèse des graphes décrits dans cette section et montre les ajouts réalises successivement :

Type de graphe :	*Représente :*	*Nœuds :*	*Liens*
Graphe de dépendance de programmes	Programmes Monolithiques (une seule procédure)	Instructions Entrée de procédure	Dep. Contrôle Dep. Données
Graphe de dépendance de Système	Programmes consistant de plusieurs fonctions	Paramètres effectifs et actuels	Lien d'appel de proc. Liaison de Param. Lien de résumé
Graphe de dépendance Orienté Objet	Programmes Orientés Objet.	Entrée de classe Choix polymorphique Lien polymorphique	Membre de classe

2.2 Découpe de programmes

Dans cette section nous allons présenter la technique de découpe de programmes, qui est réalisée à partir des graphes de dépendance.

Nous ferons d'abord une introduction qui présente les origines de cette technique. Nous expliquerons ensuite, de façon plus détaillée, la manière d'obtenir des coupes *statiques arrière* à partir des exemples donnés dans la section correspondante aux graphes de dépendance. Nous présentons uniquement des exemples détaillés sur ce type de coupe car ils permettent de comprendre le principe des algorithmes de découpe, qui sont similaires pour les divers types de coupes.

Finalement nous discuterons brièvement sur la manière d'obtenir d'autres variétés de coupes ainsi que leurs particularités.

2.2.1 Origines de la technique de Découpe

Le concept original de la "découpe de programmes" (ou *slicing* en anglais), fut trouvé en 1979 puis publié en 1984 par Mark Weiser [6]. Ce dernier découvrit que les programmeurs font certaines abstractions mentales lors du processus de corrections d'erreurs (débogage) de leurs programmes. En effet, lorsqu'un programmeur essaie de corriger une erreur qui a lieu dans un point particulier d'un programme, il examine des instructions à d'autres endroits du programme qui peuvent être responsables de l'erreur. L'abstraction mentale qu'il réalise comprend l'ensemble de ces instructions.

Observons ceci pour un exemple de programme qui produit une erreur lors de son exécution:

```
1:   main()
     {
2:    int a=3,b=3,c=2;
3:    a=a-b;
4:    c=5*b;
5:    c++;
6:    b=b+2;
7:    c=c/a;            // Erreur de division par zéro
8:    printf("%d",c);
     }
```

Dans cet exemple simple, si nous voulons chercher la raison pour laquelle il y'a une erreur de division par zéro dans la ligne 7, nous devons trouver quelles sont les instructions qui modifient la variable *a* (qui est la cause de l'erreur), et nous pouvons ignorer le reste. Ceci serait équivalent à "voir" le programme de manière simplifiée comme suit:

```
main()
{
  int a=3,b=3,c=2;
  a=a-b;                // cette ligne est à l'origine de l'erreur.
  c=c/a;
}
```

Ce programme réduit possède le même comportement relatif à la valeur de *a* que celui à partir duquel il est extrait, ce qui nous permet de trouver rapidement l'erreur. Nous dirons que ce programme est une *coupe* du programme original par rapport à la variable *a* dans l'instruction `c=c/a`.

Dans la définition originale, une coupe *S* est un programme réduit et exécutable obtenu à partir d'un programme *P* en supprimant des instructions selon un critère donné de sorte que *S* réplique une partie du comportement de *P*. Le critère $< s, v >$ spécifie un emplacement (instruction *s*) et une variable *v*.

Au début, Weiser utilisa les graphes de flot de contrôle [6] comme des représentations intermédiaires pour réaliser la découpe. D'autres auteurs notèrent postérieurement que les coupes pouvaient être calculées de manière efficace en utilisant plutôt les PDG. La découpe devînt alors un problème d'accessibilité des graphes, c'est à dire de suivi des dépendances à l'intérieur de ceux ci [13].

2.2.2 Caractéristiques des coupes

Dans la définition originale, les coupes sont des programmes exécutables obtenus en suivant, à partir d'un point donné, les dépendances qui aboutissent à ce point. En raison de cela, elles sont appelées *coupes exécutables arrière.*

D'autres variétés de coupes sont apparues postérieurement à la proposition originale. Elles se différencient par diverses caractéristiques que nous décrivons à la suite:

- *Exécutables* ou *non exécutables*: Lorsque la découpe donne lieu à un programme complet qui peut être exécuté, nous dirons qu'il s'agit d'une coupe *exécutable*. Dans le cas contraire, les coupes sont dites *non exécutables.*

- S*tatiques* ou *dynamiques*: Les coupes sont souvent réalisées seulement à partir de l'information que fournit le code, il s'agit alors de coupes *statiques.* Cependant, il est possible aussi de réaliser une coupe pour une exécution particulière d'un programme, nous parlerons alors d'une coupe *dynamique*.

- A*rrière* ou *avant* : On parle de coupe *arrière* lorsque les liens du graphe de dépendance sont traversés en sens inverse, une coupe de ce type consiste de toutes les instructions et prédicats d'un programme qui peuvent affecter la valeur de *v* au point *s*. Il existe aussi la possibilité de suivre les liens dans leur sens, et dans ce cas la coupe est appelée coupe *avant* et elle est formée par toutes les instructions et prédicats qui sont affectés par la valeur de *v* au point *s*.

Une coupe combine les diverses caractéristiques mentionnées. Par exemple, une coupe peut être *exécutable statique arrière*, comme dans le cas de la définition originale [10].

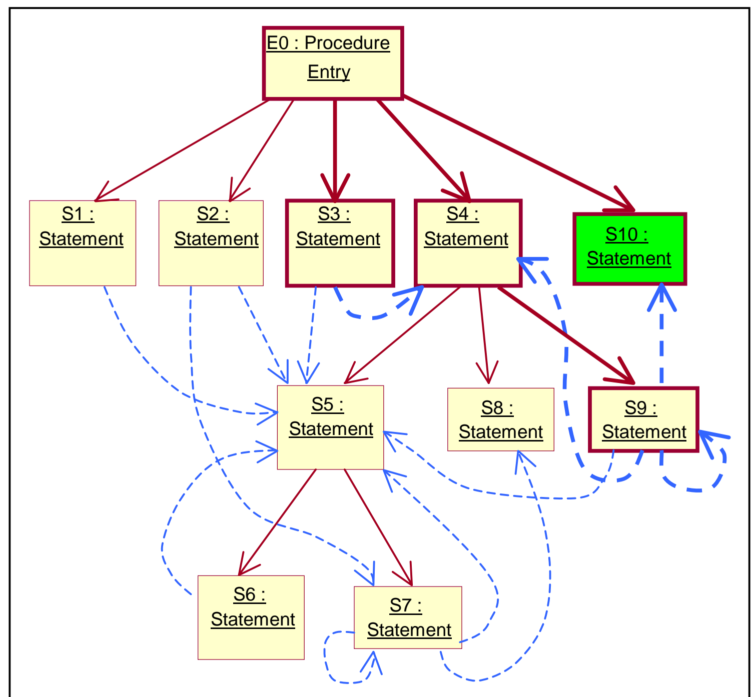
2.2.3 Découpe du Graphe de Dépendance de Programmes

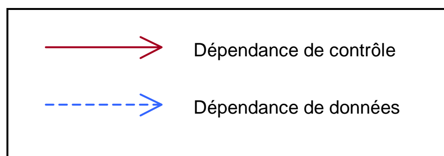
Pour un nœud *s* du PDG *G*, la coupe de *G* par rapport a *s*, est un graphe qui contient tous les nœuds qui possèdent une dépendance transitive de données ou de contrôle vers *s*, c'est à dire, tous les nœuds qui peuvent 'atteindre' *s* à travers une dépendance de l'un de ces types. Ces coupes sont appelées aussi *intra-procédurales* du fait qu'elles sont réalisées à l'intérieur d'une procédure.

L'algorithme qui réalise la découpe effectue un "marquage" ou sélection des nœuds, ceci signifie que les nœuds qui sont rencontrés lors de la traversée des liens sont marqués comme faisant partie de la coupe.

En reprenant l'exemple donne pour le PDG, la coupe arrière par rapport au nœud S10 du code montré serait le suivant :

```
E0:     main()
        {
S1:         int a=0;
S2:         int b=0;
S3:         int x=0;
S4:         while(x<3)
            {
S5:             if((a+b)<x)
S6:                 a=3;
                else
S7:                 b=b+1;
S8:             printf("%d",b);
S9:             x=x+1;
            }
S10:        printf("%d",x);
        }
```





Dans le graphe, les nœuds et les liens qui sont traversés pendant l'exécution d'un algorithme qui réalise ce type de coupe sont montrés en gras. En éliminant les nœuds qui ne sont pas impliqués dans la coupe, le programme résultant devient le suivant (il ne contient plus que des instructions concernant la variable *x*):

```
E0:         main()
            {
S3:                 int x=0 ;
S4:                 while(x<3) ;
                    {
S9:                         x=x+1 ;
                    }
S10:                printf("%d",x) ;
            }
```

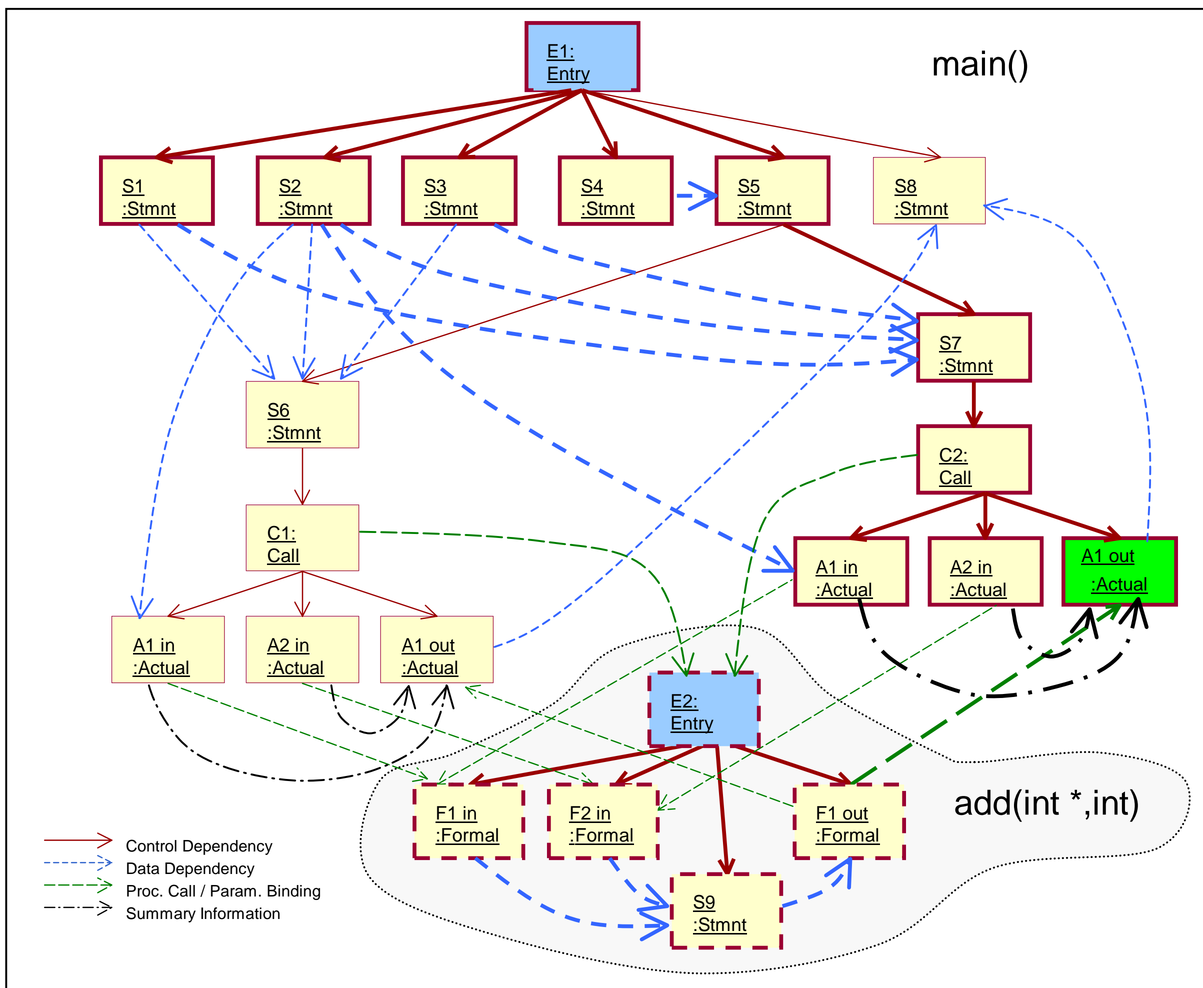
2.2.4 Découpe du Graphe de Dépendance de Système

La découpe dans le SDG est appelée *inter-procédurale* car elle est réalisée sur plusieurs procédures. Horwitz, Reps et Binkley [6] proposent un algorithme en deux étapes pour obtenir les coupes dans ces types de graphes.

Pour un graphe G et un nœud s qui se trouve dans une procédure P , la première étape permet de trouver les nœuds qui peuvent atteindre s et qui se trouvent dans P ou toute autre procédure qui appelle P. La deuxième étape identifie les nœuds qui atteignent s à partir de procédures qui sont appelées par P de manière transitive ou à partir de procédures qui appellent des procédures qui appellent à leur tour P [4].

Dans chacune des deux étapes on réalise un procédé similaire à celui qui permet de faire la découpe du SDG, c'est à dire la sélection (ou marquage) de nœuds à partir du suivi de certains liens. Dans la première étape, tous les liens sont suivis (en arrière) à l'exception des liens de paramètres de sortie. La deuxième étape traverse en sens inverse à partir des nœuds marqués dans la première étape tous les liens sauf ceux d'appel (*call*) et ceux de paramètres d'entrée, dans cette étape, la traversée "entre" dans les procédures.

Exemple de coupe pour le nœud A1_out:

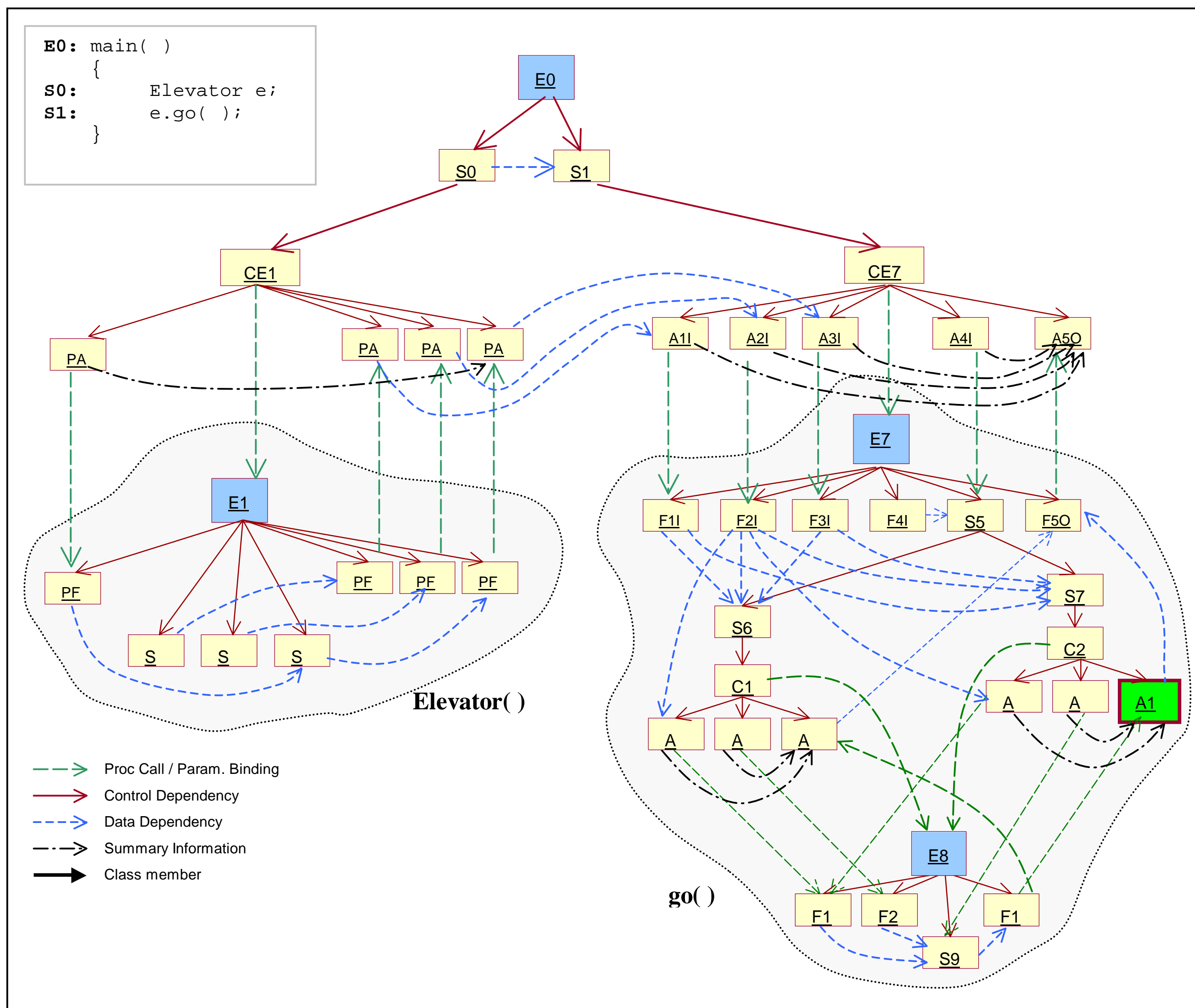


Les nœuds qui sont "marqués" sont montrés avec leur bord en ligne rehaussée, ceux de la première étape ont un bord continu, et ceux de la deuxième ont un bord pointillé. Les liens suivis sont aussi rehaussés.

2.2.5 Découpe du Graphe de Dépendance Orienté Objet

Plusieurs auteurs ont décrit la découpe des graphes de dépendance Orientés Objet ([2],[3],[11]). Nous reprenons ici la proposition de Larsen et Harrold [3], qui est réalisée avec un algorithme identique à celui employé pour le SDG, c'est à dire en 2 étapes.

Exemple : Si nous reprenons la classe Elevator définie dans la section 2.1.2, avec un programme simple qui crée un objet de ce type et appelle une de ses fonctions, nous aurons le graphe suivant :



Si nous voulons réaliser une coupe pour le nœud indique (A1), dans une première étape, tous les nœuds à l'exception de ceux de la méthode E8 seront marqués. Lors de la deuxième étape la traversée entrera dans la méthode E8 et ses nœuds seront marqués à leur tour.

Il est intéressant de noter que les liens de dépendance de données qui existent entre les paramètres des appels de méthodes CE1 et CE7 représentent le fait que le constructeur (E1) modifie des attributs de la classe qui sont utilisés par la méthode (E7), ceci est responsable du fait que les nœuds du constructeur soient inclus dans la coupe.

En ce qui concerne les graphes contenant les appels polymorphiques, l'algorithme de découpe inclura tous les nœuds qui constituent l'ensemble des méthodes pouvant être des candidats à un appel particulier.

2.2.6 Autres variétés de Découpes

Nous avons présenté la manière de réaliser la découpe *statique arrière* pour les divers types graphes de dépendance. Les autres types de coupes que nous avons mentionné auparavant sont obtenus d'une manière similaire, nous décrivons à la suite quelques points particuliers relatifs à chacune des variétés de coupes.

2.2.6.1 Découpe Avant

La différence entre la découpe avant et la découpe arrière est que la découpe arrière cherche à trouver tous les éléments d'un programme qui *peuvent affecter* une valeur, alors que la découpe avant consiste en un ensemble de toutes les instructions et prédicats du programme *pouvant être affectés* par la valeur de *v* en *s*. Un algorithme pour réaliser des coupes inter-procédurales *avant* peut être réalisé sur les SDG, en utilisant les mêmes techniques employées pour la réalisation de la découpe arrière (c'est à dire un algorithme de suivi du graphe en deux étapes). La première étape de l'algorithme ne traverse pas les liens d'appel ni de paramètres d'entrée, et donc elle n'entre pas dans les procédures appelées. La deuxième étape ne suit pas les liens de paramètres de sortie et pour cette raison elle ne "remonte" pas dans les procédures qui réalisent les appels.

2.2.6.2 Découpe Dynamique

La découpe dynamique des programmes est un concept qui est apparu postérieurement à celui de la découpe statique. Une coupe dynamique diffère d'une coupe statique du fait qu'elle emploie de l'information sur une exécution particulière du programme. Ainsi, une coupe dynamique contient toutes les instructions qui *affectent réellement* la valeur d'une variable à un point particulier du programme pour une exécution de celui ci [5][6].

Les coupes dynamiques sont calculées par rapport à un "historique d'exécution" (aussi appelé trajectoire). Cet historique enregistre l'exécution d'instructions pendant que le programme s'exécute. L'exécution d'une instruction produit une occurrence d'une instruction dans l'historique, de cette manière, l'historique est une liste d'occurrences d'instructions.

La découpe dynamique peut être vue aussi comme un problème d'accessibilité des graphes. Il existe plusieurs algorithmes pour réaliser ceci [5]. La plupart réalisent un marquage des éléments

du graphe comme ayant été "exécutés". Les marques peuvent être appliquées soit sur les nœuds soit sur les liens. Après l'exécution, la coupe est calculée en appliquant un algorithme de découpe statique seulement sur les nœuds ayant été marqués (ainsi que sur les liens qui les relient).

2.2.6.3 Découpe d'Interfaces

Une variété particulière de découpe est employée pour obtenir des coupes d'interfaces [7]. A différence des coupes conventionnelles qui cherchent à isoler le comportement d'un ensemble spécifié de variables, une coupe d'interface cherche à isoler des comportements spécifiques qu'un module (package en Ada, classe en C++) exporte au système logiciel qui le contient.

Une coupe d'interface est réalisée dans un module qui contient une spécification d'une interface et le code qui l'implémente. Le critère de découpe consiste en un sous ensemble d'opérations choisies à partir de celles qui sont déclarées dans l'interface.

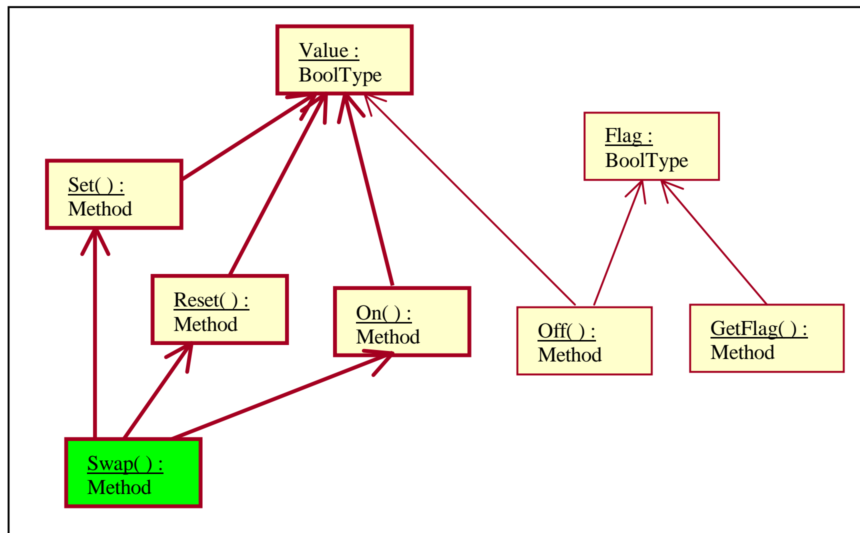
Les coupes sont réalisées à l'aide d'un Graphe de Dépendance d'Interface, dans lequel les nœuds correspondent à des types, des variables globales ou des sous-programmes. Les liens du graphe sont des liens de dépendance par rapport aux références de définition ou d'utilisation des divers éléments.

Les coupes sont calculées en réalisant une fermeture transitive des nœuds du graphe en suivant les dépendances qui existent entre eux. Elles résultent en un nouveau module qui est un sous ensemble de l'original mais qui contient le code nécessaire pour supporter la fonctionnalité spécifiée par le critère des opérations requises. Ces coupes sont donc de type statique et exécutables et n'ont pas un sens particulier.

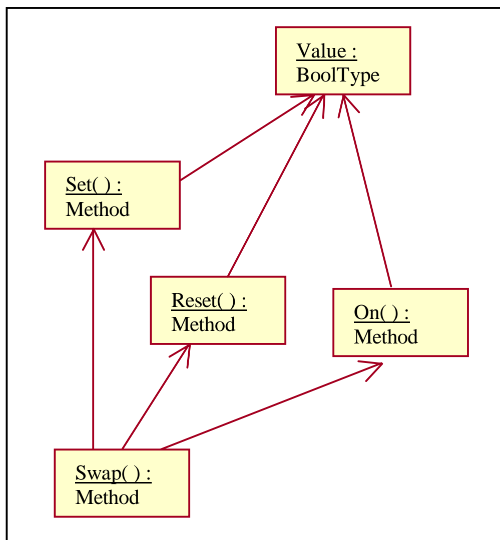
Nous allons montrer à la suite un exemple de découpe d'interface réalisé dans une classe simple.

```
class toggle {
private:
   bool Value;
   bool Flag ;
public:
   bool GetFlag( ) {return Flag;}
   bool On( ) {return Value ==TRUE;}
   bool Off( ) {return Value ==FALSE;Flag=TRUE}
   void Set( ) {Value =TRUE ;}
   void Reset( ) {Value =FALSE ;}
   void Swap( ) {if(On())
                   Reset( );
                 else
                    Set( );}
};
```

Graphe correspondant :



Coupe par rapport a Swap( ) :




```
class toggle {
private:
   bool Value;
public:
   bool On( ) {return Value ==TRUE;}
   void Set( ) {Value =TRUE ;}
   void Reset( ) {Value =FALSE ;}
   void Swap( ) {if(On())
                   Reset( );
                 else
                    Set( );}
};
```

L'intérêt particulier de ce type de découpe est qu'il se place à un niveau de granularité supérieur que l'ensemble de coupes que nous avons décrit précédemment, car l'analyse ne se fait pas au niveau des instructions mais plutôt au niveau des méthodes.

2.2.7 Applications de la découpe

L'utilisation de la découpe a été proposée pour diverses applications. Parmi ces applications, nous pouvons citer :

- Différentiation de programmes. Les coupes permettent d'identifier des différences sémantiques entre deux programmes.

- Intégration de programmes. Cette opération concerne le problème de fusionner des variantes de programmes, les coupes permettent d'étudier "l'interférence" qu'il peut y avoir entre les variantes.

- Maintenance des logiciels. Les applications principales de la découpe dans ce domaine sont la compréhension et la modification des logiciels, à travers l'étude des dépendances vers certains éléments ainsi que l'étude d'impact lors de modifications.

- Réalisation de tests après la modification. La découpe peut permettre de diminuer la quantité de tests qui doivent être réalisés après la réalisation de modifications sur un logiciel.

2.2.8 Synthèse

Nous avons détaillé comment la technique de *découpe arrière* peut être appliquée à divers types de programmes, commençant par ceux qui peuvent être considérés comme étant simples, consistant d'une fonction unique, jusqu'à ceux qui présentent un degré élevé de complexité, comme serait le cas des programmes orientés objet.

Nous avons aussi présenté d'autres types de *slicing*, tels que le *slicing* avant, dynamique et d'interfaces qui sont réalisés suivant des algorithmes différents sur des graphes de dépendance. La découpe d'interface présente un intérêt particulier du fait qu'elle est réalisée à un niveau de granularité supérieur que les autres techniques qui ont été présentées.

2.3 Conclusion

Toutes les variétés de découpe que nous avons décrit ici sont réalisées à partir de graphes de dépendance, et nous pouvons remarquer que la possibilité d'obtenir des coupes dépend directement des possibilités d'obtention des graphes de dépendance pour un programme donné.

La précision des coupes dépend de la complexité du langage choisi, ainsi que du critère choisi pour réaliser la coupe.

Nous pouvons mentionner aussi qu'il est possible de proposer d'autres types de coupes, pour cela il faudra prendre en compte :

- L'obtention d'un graphe de dépendance qui représente le programme sur lequel on veut réaliser la découpe.

- Le choix d'un critère et de règles ou contraintes pour réaliser les coupes.

- L'application de ces règles sur le graphe de dépendance.

CHAPITRE 3 : L'Object Modeler : Un cas d'étude

3.1 Introduction

Dans ce chapitre nous allons donner les raisons pour lesquelles les techniques présentées dans le chapitre précédent peuvent être appliquées pour les logiciels construits avec l'Object Modeler, qui sera lui même introduit dans ce chapitre. La représentation graphique des éléments de l'Object Modeler présentée dans cette étude suit la représentation utilisée dans l'ensemble des travaux menés dans l'équipe Adèle [9], cependant nous utiliserons aussi UML pour décrire de manière plus précise les relations.

3.1.1 Origines de l'Object Modeler

Dassault Systèmes (DS) développe depuis le début des années 80 un logiciel de CFAO (Construction et Fabrication Assistée par Ordinateur) de nom CATIA. Actuellement dans sa 4ème version, l'entreprise se prépare au lancement de la version 5 [8].

CATIA est formé principalement par un noyau sur lequel il est possible de construire des applications partageant des caractéristiques communes. On peut citer par exemple l'interface utilisateur ou des protocoles communs offerts par l'architecture.

Ce besoin d'étendre un noyau pour créer des applications fût une des motivations principales qui orientèrent la construction de son architecture. Le paradigme Orienté Objet se révéla être le plus adéquat pour répondre à ce besoin d'extension et à l'époque où l'on décida d'implémenter ceci (début des années 90), le langage C++ fût choisi par DS pour le développement.

Cependant certaines caractéristiques du langage C++ furent pour DS plutôt des limitations que des avantages. Particulièrement : les dépendances de compilation (besoin recompiler des grandes parties du logiciel lors d'une modification dans un fichier), des contraintes imposées par l'héritage multiple et la difficulté de faire évoluer les abstractions en gardant la compatibilité avec les applications construites sur des versions antérieures du noyau. Une autre contrainte très importante était le besoin de ne pas distribuer le code source du noyau.

Le résultat de ce cahier des charges est l'Object Modeler (OM), qui est vu comme « une extension naturelle du langage de programmation C++ ». L'OM consiste en un ensemble de constructions donnant lieu à des mécanismes qui permettent de résoudre les problèmes posés par le langage de base.

3.1.2 Le besoin d'étudier les dépendances

En raison de la taille du logiciel CATIA et de la vitesse de son développement, il n'existe pas une connaissance précise sur les dépendances entre les éléments qui le composent. Connaître ces relations permettrait de réaliser un nombre important d'études parmi lesquelles nous pouvons citer :

- La compréhension de l'architecture : Une vue générale de l'ensemble des éléments du logiciel et de leurs relations permettrait d'apprécier la structure du logiciel et d'aider à sa compréhension.
- L'analyse d'Impact : Lors de la réalisation de modifications, il serait possible de savoir quelles sont les parties susceptibles d'être affectées en observant les dépendances vers les éléments modifiés.
- L'élimination d'éléments non utilisés : des éléments vers lesquels il n'existe pas de dépendances sont probablement inutilisés et leur élimination pourrait contribuer à la réduction de la taille du code.
- Optimisation de la livraison : Un client qui construit une application sur le noyau peut ne pas avoir besoin de tout l'ensemble de celui ci. Une connaissance des dépendances qui existent entre son développement et le noyau permettrait de livrer de manière plus précise les parties requises par le client.

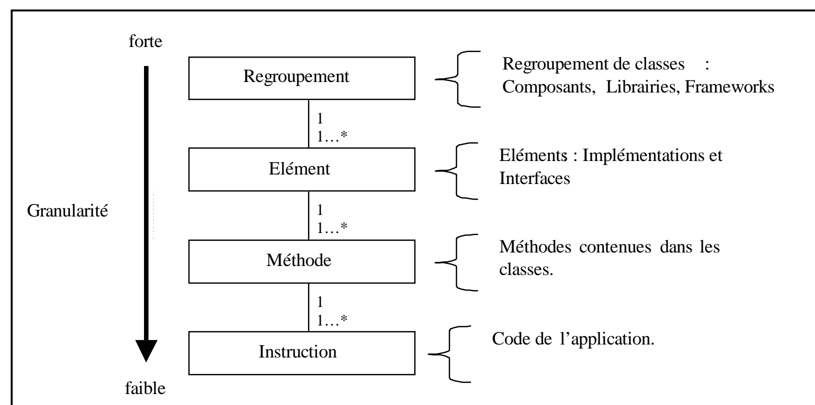
Ces applications nécessitent de manière générale d'être effectuées sur une représentation adéquate du logiciel, car leur réalisation implique le suivi de certaines dépendances entre les éléments de celui ci. Par leurs caractéristiques, les graphes de dépendance et la découpe sont des techniques qui peuvent permettre de résoudre ce type de problème.

3.1.3 Granularité et choix des éléments de l'étude

L'Object Modeler propose un certain nombre de concepts qui n'existent pas de manière directe dans le langage C++ avec lequel il est construit. Certains d'entre eux sont des mécanismes qui se trouvent à un niveau d'abstraction supérieur à celui que l'on peut décrire avec le langage original. Pour cette raison, il est possible d'avoir diverses visions des logiciels réalisés avec l'OM, selon le niveau d'abstraction dans lequel on se place.

Un niveau d'abstraction très bas, comme pourrait être celui du code, est de granularité fine car les éléments qui le composent ne peuvent pas être divisés en d'autres éléments de taille inférieure. Un niveau d'abstraction haut, comme pourrait être celui des composants (concept que nous détaillerons plus loin), est de granularité forte car ses éléments peuvent être divisés en un ensemble d'éléments plus fins (gains gros).

De manière schématique nous pourrions représenter les divers niveaux d'abstraction de l'Object Modeler de la manière suivante :



Il est important de choisir un niveau de granularité correct pour réaliser l'étude du logiciel. Le niveau dépend du type d'analyse que l'on veut réaliser et le choix final dépend d'un compromis. Cependant, un niveau très fort de granularité qui fait apparaître un très grand nombre de détails, entraîne une augmentation de la complexité. D'un autre coté, un niveau très faible de granularité risque de cacher des détails importants.

L'analyse que nous cherchons à réaliser est centrée principalement autour du niveau *élément* qui regroupe deux types de classes : Les interfaces et les implémentations. La justification de ce choix est qu'il permet d'avoir un niveau de détail suffisant pour travailler à l'intérieur des composants et que l'analyse n'est pas trop fine pour devenir difficile à réaliser. Il est nécessaire cependant de présenter certains concepts qui se trouvent à un niveau supérieur d'abstraction car ils permettent de mieux comprendre le modèle.

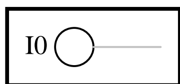
3.2 Concepts de l'Object Modeler

Dans cette section, nous allons présenter les éléments principaux de l'Object Modeler qui se trouvent aux niveaux d'abstraction *élément* et *regroupement* ainsi que les relations qui existent entre eux.

3.2.1 Entités du niveau "élément"

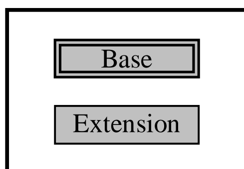
Les entités introduites par l'Object Modeler qui sont construites au dessus de la couche du langage C++ (au niveau *élément* ou classe*)* sont les *interfaces* et les *implémentations*.

3.2.1.1 Interfaces



A différence du modèle objet conventionnel de C++, dans le modèle OM, les messages entre objets sont reçus exclusivement par des interfaces qui les transmettent aux objets clients. Les interfaces, qui sont des classes qui ne sont jamais instanciées, contiennent les signatures des méthodes à travers lesquelles le client obtient des services du serveur. Une interface n'a cependant aucune connaissance sur la manière avec laquelle les méthodes sont implémentées. Une conséquence de ceci est le fait que l'implémentation qui implémente une interface peut changer sans que ceci ait un impact sur le client. Les interfaces sont manipulées par les clients exclusivement au moyen de références.

3.2.1.2 Implémentations



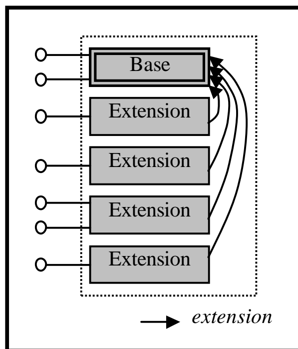
Il s'agit de classes qui donnent lieu aux objets serveurs qui se trouvent "derrière" les interfaces et qui fournissent des services aux objets clients. Elles doivent réaliser deux tâches principales : déclarer que leur code est accessible aux clients à travers un certain nombre d'interfaces et fournir le code qui implémente les méthodes déclarées à l'intérieur de ces interfaces. Les implémentations ne sont jamais accédées de façon directe par les clients.

Il existe deux types différents d'implémentations, les *bases* et les *extensions*. Les *bases* sont des implémentations simples tandis que les *extensions* permettent d'étendre (ou augmenter) une base pour former des regroupements à un niveau supérieur d'abstraction.

3.2.2 Entités du niveau "regroupement"

A un niveau au dessus de la couche *élément*, nous trouvons deux catégories de regroupements : les regroupements conceptuels et physiques. Du côté conceptuel, nous trouvons les *composants*, du côté physique, nous trouvons les *librairies* et les *frameworks.*

3.2.2.1 Regroupements conceptuels



Composants : Ce regroupement est constitué par plusieurs éléments : une base unique[1] reliée à un nombre variable d'extensions, et les interfaces correspondantes à l'ensemble de ces implémentations.

La relation qui permet de regrouper la base et les extensions est la relation d'*extension* (que nous détaillerons plus loin), qui permet entre autres à un client qui à accès à l'une des interfaces, d'avoir accès aux autres, sans se soucier du type d'implémentation qui se trouve "derrière" les interfaces qu'il accède.

Il faut noter que dans la représentation des composants nous montrons les relations d'extension entre la base et ses extensions, cependant nous pourrions les omettre, car la 'frontière' qui les regroupe les rend implicites.

Les concept de base et de composant sont étroitement liés, car un composant peut être formé uniquement d'une base, et de ce fait, on identifie normalement un composant en faisant référence à sa base.

Les bases (et donc les composants) sont instancées de manière indirecte à travers des *fabriques* qui suivent elles aussi les mêmes règles de structuration (elles sont aussi des composants). Leur tache est de créer la nouvelle instance et de faire connaître au client une des interfaces implémentées par celle ci.

Les instances des extensions sont créées de manière automatique lorsqu'un client accède une des interfaces qu'elles implémentent.

3.2.2.1 Regroupements physiques

Librairies : Il s'agit d'un regroupement physique car les librairies contiennent le code correspondant aux implémentations et aux interfaces. Elles se trouvent donc au même niveau que les composants mais ne regroupent pas les mêmes éléments.

Framework : Le *Framework* est une autre type de regroupement physique et il contient un ensemble de librairies. L'intérêt de ce regroupement est qu'il représente l'unité de regroupement pour le travail de collaboration.

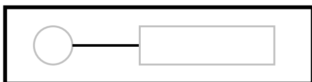
---

[1] Il existe aussi un mécanisme permettant de regrouper des extensions sans que leur base soit connue dès le début, il s'agit du mécanisme des *type late* que nous n'étudierons pas dans ce rapport

3.2.3 Relations de regroupement en composants

Il existe plusieurs types de relations entre les entités de l'Object Modeler. Dans cette section, nous allons décrire plus en détail celles qui sont en relation à la constitution des composants:

- L'implémentation
- L'extension
- La délégation
- L'héritage

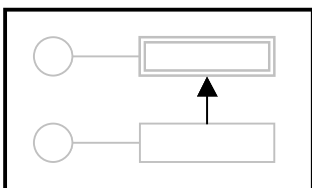


Implémentation. La relation d'*implémentation* est une relation fondamentale dans le modèle. Elle à lieu entre une interface et une implémentation et elle signifie plus concrètement que l'implémentation donnée fournit le code correspondant aux méthodes déclarées dans l'interface.

Le modèle décrivant cette relation est le suivant :

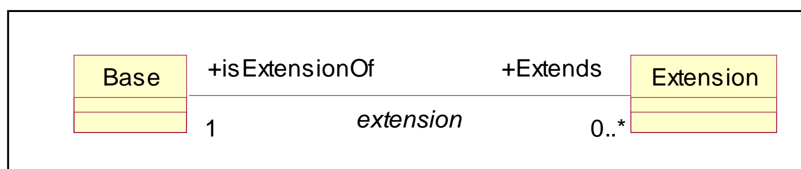
Une interface particulière peut être implémentée par plusieurs implémentations et une implémentation peut implémenter plusieurs interfaces. Une implémentation particulière déclare quelles sont les interfaces dont elle implémente les méthodes. Cependant ceci n'est pas vrai en sens inverse, c'est à dire que les interfaces n'ont pas connaissance des implémentations qui les implémentent. Les raisons de ceci seront décrites plus loin.

Il existe aussi une variété d'implémentation appelée *implémentation conditionnelle*. La différence avec l'implémentation conventionnelle est qu'il est possible de décider si une implémentation implémente ou non une certaine interface à partir d'une condition.

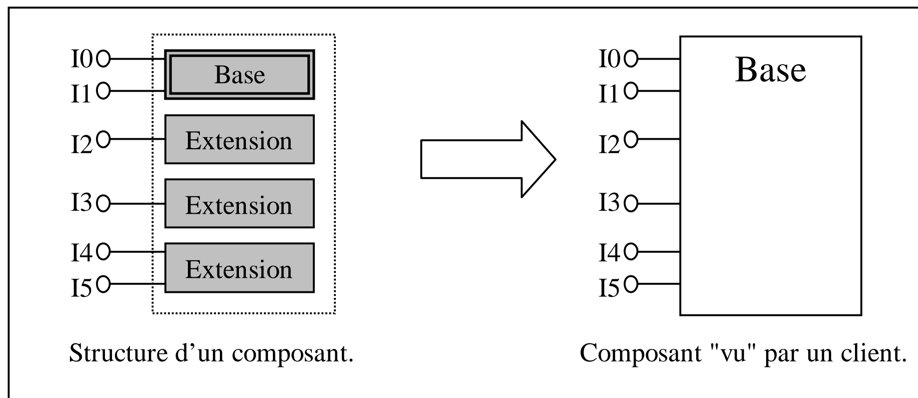


Extension. La relation d'*extension*, qui est aussi fondamentale dans le modèle, à lieu comme nous avons décrit précédemment entre une implémentation de base et une extension.

Le modèle UML correspondant à cette relation est le suivant :



Nous rappelons qu'un *composant* est un regroupement d'un ensemble d'implémentations, dont une base et zéro ou plus extensions ; cependant l'extension fait qu'il soit "vu" par le client de manière similaire à une "boite noire" qui *affiche* (*displays*) un certain nombre d'interfaces.

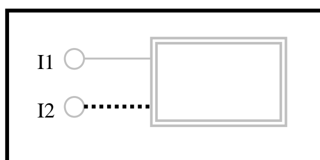


Structure d'un composant. Composant "vu" par un client.

La relation d'extension permet d'ajouter des fonctionnalités à une base. Une fois que la base est étendue par un certain nombre d'extensions, il est possible, pour un objet client d'avoir accès à toutes les interfaces du même composant à travers une méthode commune à toutes les interfaces appelée *QueryInterface.*

QueryInterface retourne une référence vers une interface du composant si elle est implémentée par celui ci, directement ou par héritage. Si l'implémentation d'une interface que est demandée n'est pas instancée lors de la demande de cette interface, elle sera créée de façon transparente pour le client.

L'extension d'une base peut être effectuée postérieurement à sa création, et ceci sans que son code soit modifié. L'information permettant de réaliser ceci est stockée dans un fichier appelé *dictionnaire* qui contient une liste des interfaces qui sont implémentées par chaque composant ainsi que la *librairie* où elles se trouvent (il s'agit communément d'une *DLL*[2]). Ainsi, chaque fois que l'on réalise l'extension d'une base, le dictionnaire est modifié en rajoutant une ligne correspondant à chaque nouvelle interface.



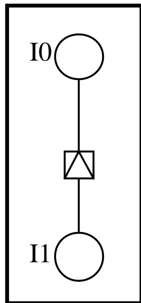
Délégation : a *délégation* est un type particulier d'*implémentation*. Dans ce cas là, un composant (appelé *delegating*) affiche des interfaces qui seront implémentées par un autre composant (le *delegated*). Ceci est réalisé en temps d'exécution par client ayant accès aux interfaces des deux composants impliqués dans la relation. De manière similaire à l'extension, la délégation s'effectue seulement dans le dictionnaire, sans besoin de modifier le code des composants acteurs dans la relation.

La délégation pose un problème de résolution en temps d'exécution, car on ne sait pas, de manière statique, quel composant sera le responsable d'implémenter les interfaces déléguées.

[2] DLL signifie Dynamic Link Library (de Microsoft).

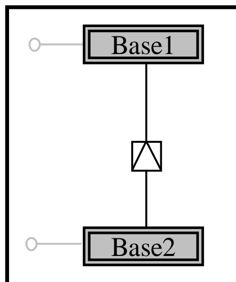
Héritage.

L’Object Modeler propose également le concept d’héritage. Ce type d’héritage peut être effectué seulement sur les interfaces et sur les implémentations de base. Pour ces deux entités il prend une signification différente que nous allons préciser à la suite :



Héritage OM sur Interfaces : Ce type d’héritage implique qu’il y à un héritage C++ entre les interfaces et qu’un client qui à accès à l’une des interfaces du composant peut obtenir à travers *QueryInterface*, une référence à I0. Dans ce cas I0 sera implémentée par la même implémentation que I1.

Dans un même composant il ne peut jamais y avoir deux interfaces qui héritent au sens OM d’une autre interface commune, car il pourrait apparaître des problèmes de détermination de l’implémentation qui implémente l’interface dont on hérite.



Héritage OM sur Implémentations : Ce type d’héritage qui a lieu exclusivement sur les bases, implique aussi un héritage C++ entre celles ci. Le résultat de cet héritage sera que le composant Base2 affichera toutes les interfaces qu’il implémente (base + extensions), ainsi que toutes celles de Base1 (base + extensions). De plus il est possible d’obtenir des références à l’ensemble des interfaces à travers *QueryInterface*. On appelle aussi ce type d’héritage héritage entre composants.

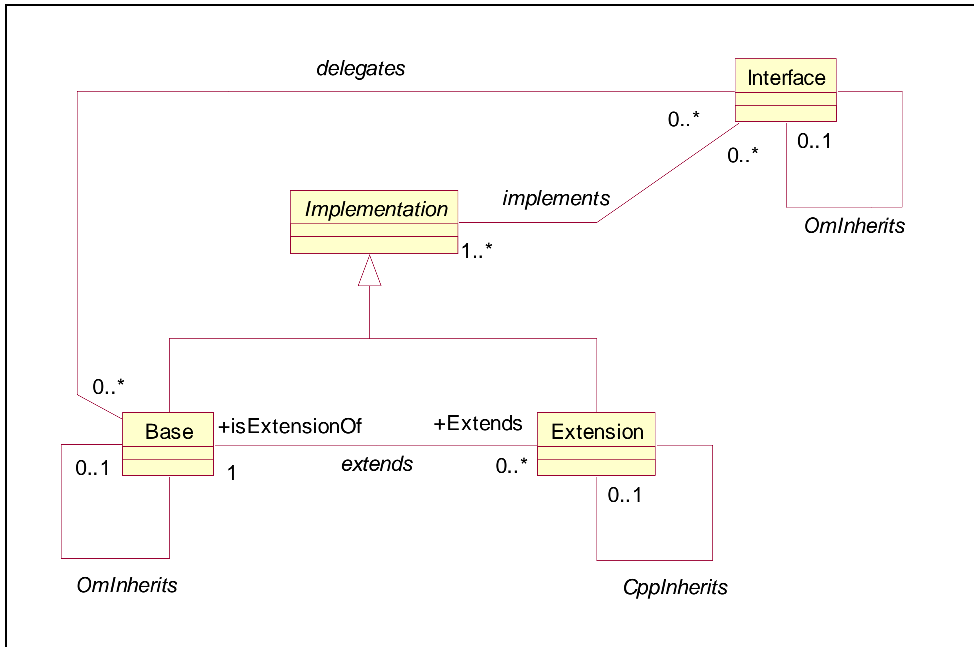
L’héritage au sens OM ne peut avoir lieu qu’entre deux bases ou entre deux interfaces, et dans le cas des bases, les interfaces affichées par la classe dont on dérive ne doivent pas être reimplementées par la classe dérivée.

Deux extensions ne peuvent hériter entre elles que de manière conventionnelle, c’est à dire, au sens C++. L’extension qui hérite rajoute des fonctionnalités à l’extension dont elle hérite.

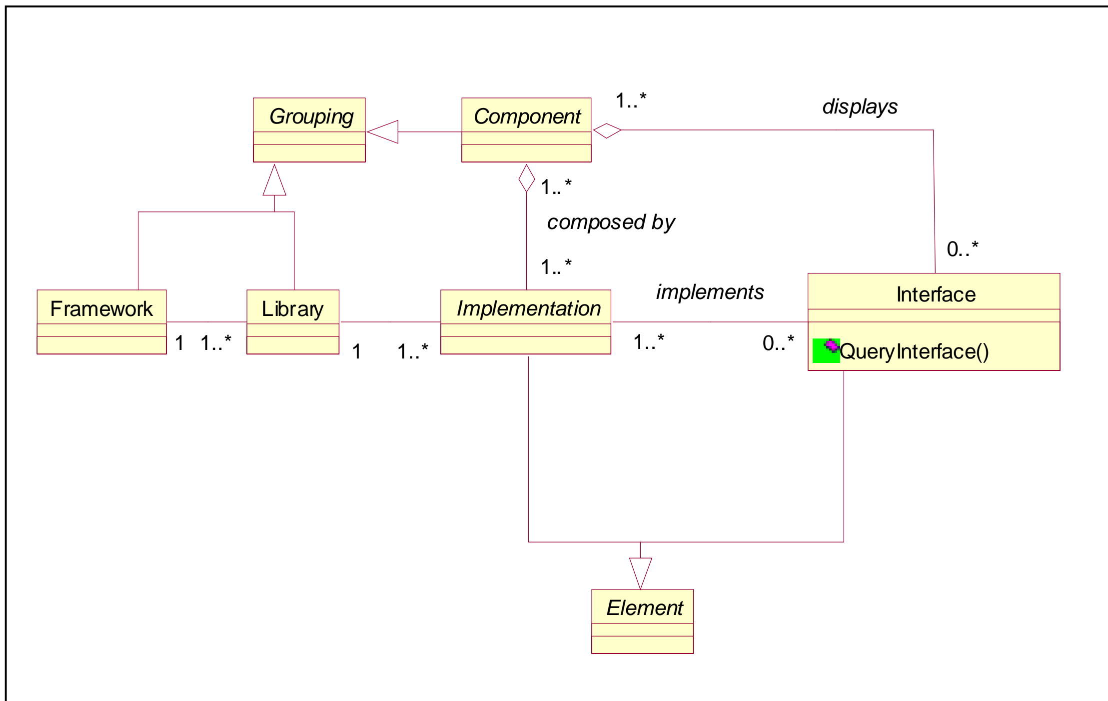
Un dernier point qu’il est important de souligner est que l’héritage multiple C++ est interdit dans le modèle; cependant, ceci n’interdit pas la possibilité d’avoir des multiples niveaux d’héritage.

3.2.4 Synthèse

Nous pouvons modéliser les relations décrites auparavant de la manière suivante :

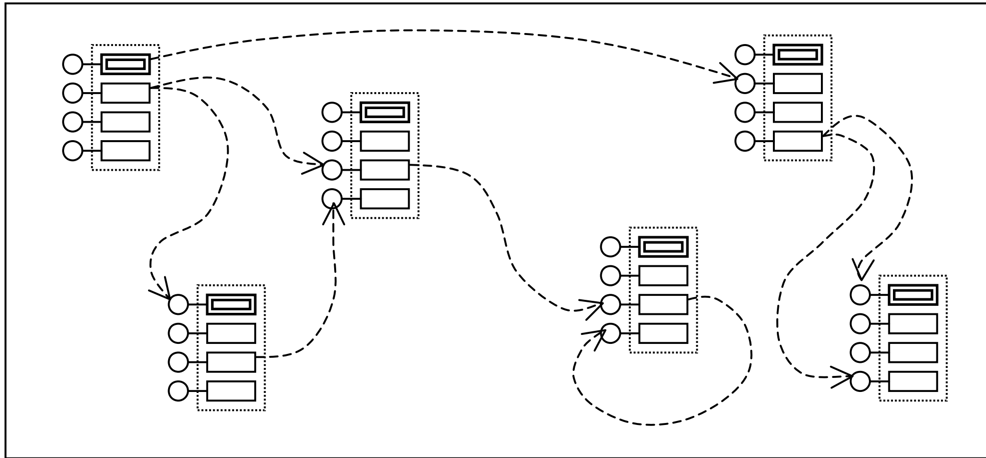


A un niveau de granularité supérieur, le modèle qui décrit les composants ainsi que certaines autres entités dont nous avons fait mention dans la description est le suivant :



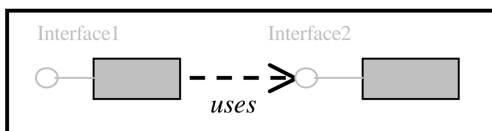
3.3 Relations de dépendance

Dans la section précédente, nous avons déjà introduit certaines relations qui existent entre les éléments définis par l'Object Modeler. Ces relations nous permettent de connaître de manière détaillée la structure des composants qui constitue le logiciel. Nous allons introduire maintenant un autre type de relation : la relation de dépendance, qui peut avoir lieu aussi bien à l'intérieur qu'à l'extérieur des composants, et qui d'une manière schématique pourrait être représenté comme suit :



Les flèches en pointillés représentent des dépendances entre les éléments du logiciel. Ces dépendances ont leur origines à l'intérieur des implémentations et aboutissent dans des interfaces qui se trouvent soit dans le même composant soit dans un autre. Nous allons décrire plus en détail cette relation dans les points suivants.

3.3.1 La relation *uses*



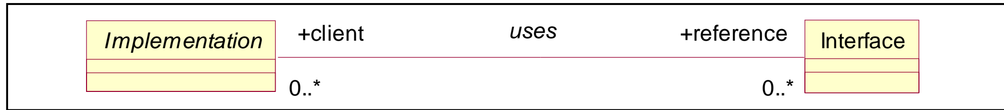
Lorsque nous avons introduit les entités définies par l'Object Modeler, nous avons fait mention du fait que les interfaces sont le seul moyen qu'il existe pour avoir accès aux implémentations. Les clients ne possèdent que des références aux interfaces et celles ci sont employées principalement pour appeler les méthodes que ces interfaces affichent ou bien pour obtenir des références à d'autres interfaces.

Une implémentation et une interface sont reliées par une relation *uses* si une implémentation possède une référence vers une interface, et ceci indépendamment de l'utilisation qu'elle en fait, c'est à dire si elle appelle ou non ses méthodes (Nous ferons cette supposition, car une connaissance plus détaillée impliquerait une analyse du flot de contrôle et de données à l'intérieur des méthodes, ce qui est en dehors des objectifs de cette étude).

Par sa nature, cette relation englobe certaines situations telles que la création d'instances des composants, car un client n'aura accès qu'aux interfaces des fabriques qui les créent, et c'est à travers celles ci qu'il recevra une référence à une interface du nouveau composant. Dans ce cas, le client aura donc au moins deux relations *uses*, la première vers l'une des interfaces de la

fabrique du composant en particulier et la deuxième vers l'une des interfaces de ce nouveau composant.

Le modèle UML qui définit cette relation est le suivant :



Cette relation peut exister entre une implémentation et une interface qui se trouvent regroupées à l'intérieur d'un même composant, cependant. Il se peut qu'elles se trouvent aussi dans des composants différents. Cela donnera alors lieu, à un niveau de granularité supérieur, à une dépendance entre ces composants.

Ce type de relation peut être vu comme une dépendance de contrôle car son existence se traduit en un possible transfert du flot d'exécution du coté de l'élément dont on dépend. De plus il n'existe pas de dépendances de données entre implémentations.

Obtention de la dépendance

Du fait que cette dépendance a son origine à l'intérieur des implémentations, il est nécessaire d'analyser le code de ces implémentations pour la trouver. Nous allons décrire deux techniques permettant d'obtenir ces relations sans devoir réaliser une analyse approfondie des dépendances de contrôle et de données à l'intérieur des méthodes. Les deux solutions effectuent une analyse du code.

La première solution consiste à trouver les dépendances entre les fichiers des implémentations et ceux où sont déclarées les interfaces (fichiers .h). Ceci pourrait permettre de faire une approximation de l'ensemble des interfaces référencées par une implémentation. Cependant cette approche risque d'être imprécise du fait que les fichiers d'entête peuvent s'inclure récursivement et qu'un fichier qui est inclus n'implique pas que l'interface qu'il déclare soit utilisée.

La deuxième solution consiste à extraire du code le nom de toutes les interfaces référencées. Ces noms peuvent apparaître dans diverses situations :

1) - Une interface est déclarée comme un attribut d'une implémentation.

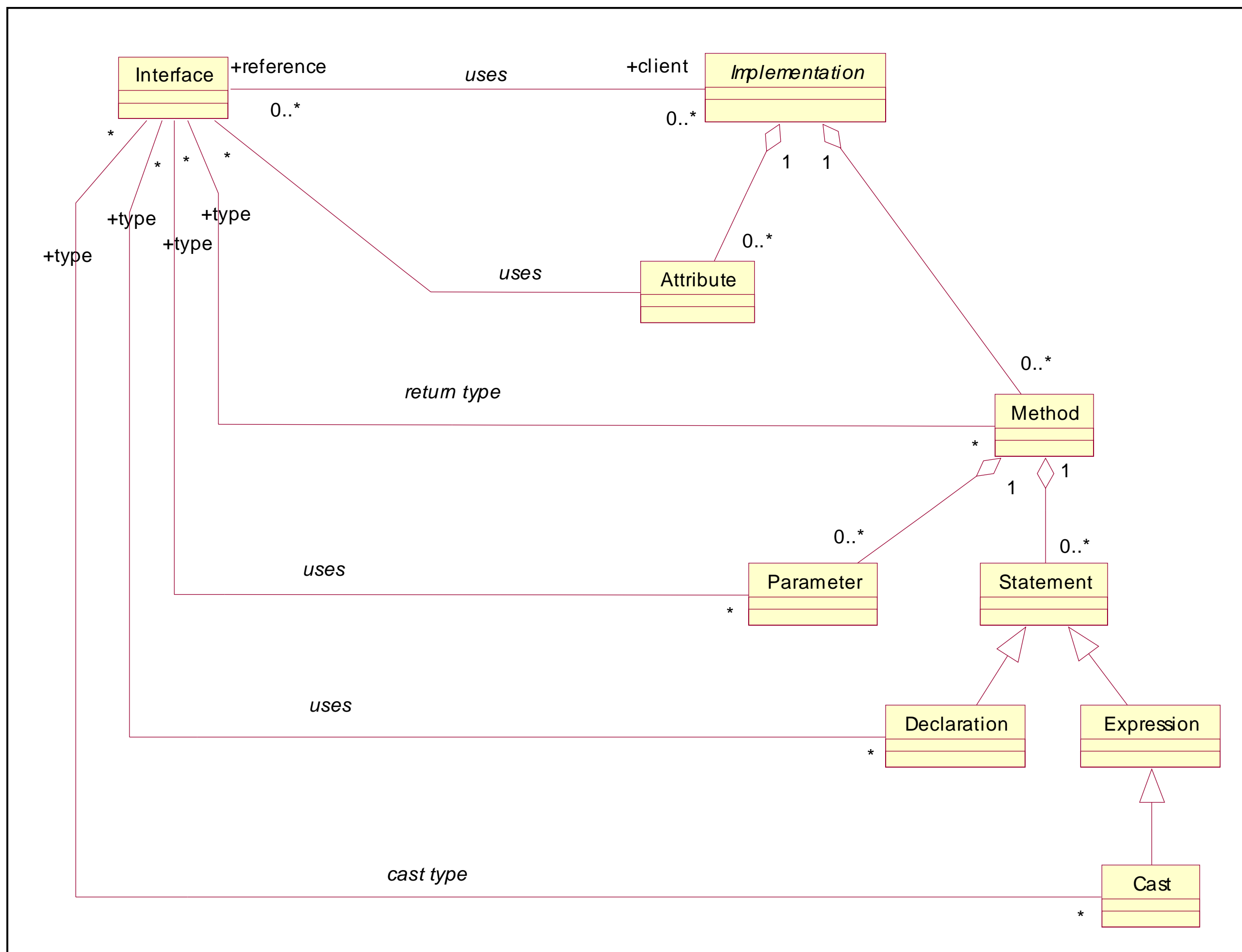
2) - Une interface apparaît dans la signature d'une méthode (paramètre/valeur de retour)

3) - Une interface est déclarée comme une variable locale à une méthode.

4) - Un changement de type (*cast*) est effectué en utilisant une interface.

L'avantage de cette solution est qu'elle permet d'avoir une précision par rapport à la première. De plus, l'ensemble de ces situations inclut l'appel de *QueryInterface* car celui ci retourne un pointeur générique qui doit être changé à travers un cast au type de l'interface que l'on désire accéder.

Le modèle qui représente cette relation ainsi que l'ensemble des cas qui l'originent est le suivant :

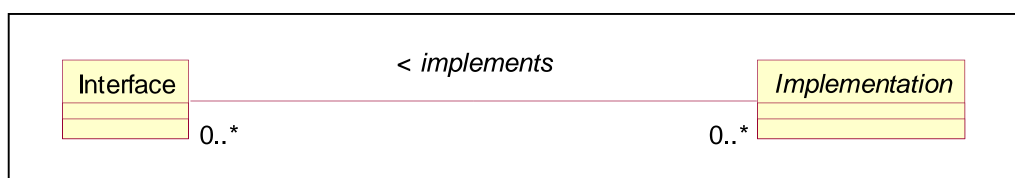


L'importance de ce modèle réside dans le fait qu'une fois que l'on connaît les situations où l'on peut trouver cette dépendance, il est possible de proposer une méthode d'analyse du code source qui permette de trouver la liste des interfaces desquelles dépend une implémentation donnée.

3.3.2 La relation *implements*

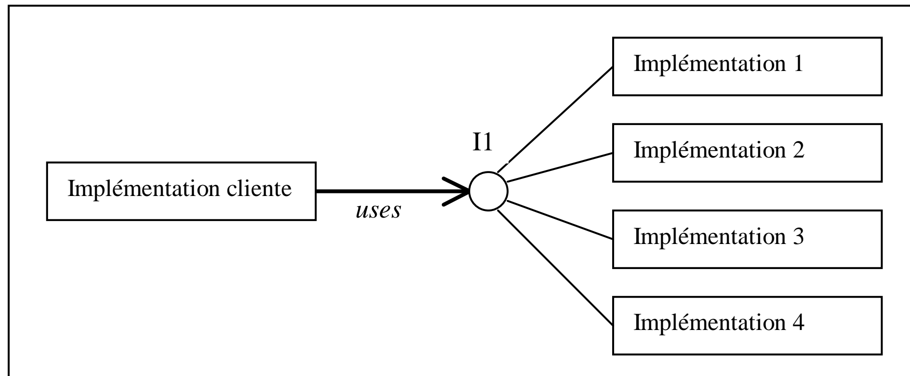
Nous avons déjà parlé de la relation d'implémentation dans la section précédente; cependant nous la traitons à nouveau, cette fois dans le contexte du rôle qu'elle joue auprès des dépendances entre composants.

Comme nous avons vu, la relation *implements* existe entre une interface et une implémentation. Elle est définie de la manière suivante :



Il existe donc pour toute interface un ensemble d'implémentations susceptibles de fournir le code correspondant aux méthodes déclarées ; cependant lors d'une analyse statique il est difficile de savoir laquelle des implémentations en particulier va prendre en charge les appels à l'interface. Ceci est dû au fait que les clients ne manipulent que des interfaces et jamais des implémentations.

D'un point de vue statique, un client qui à une relation *uses* avec une interface se voit confronté à une situation similaire à celle qui est schématisée ici :



Les implementations numerotées de 1 à 4 sont des possibles candidats d'implementation de I1, la liaison en pointillés entre l'interface et l'implémentation représente le fait que le choix est multiple.

Du fait qu'à l'intérieur de l'Object Modeler il est toujours nécessaire de passer par une interface pour avoir accès à une implémentation, cette difficulté apparaît pour toutes les relations de dépendance. Et donc, toutes les dépendances dans le modèle sont résolues de manière dynamique.

3.3.3 Synthèse

Nous avons présenté les deux relations qui originent l'ensemble des dépendances d'utilisation entre les éléments du logiciel, il s'agit des relations *uses* et *implements.*

Nous avons mentionné aussi les possibilités d'obtenir la relation *uses* à travers l'analyse du code et finalement nous avons décrit les raisons pour lesquelles les dépendances *uses – implements* ne peuvent être résolues de manière statique

3.4 Conclusion

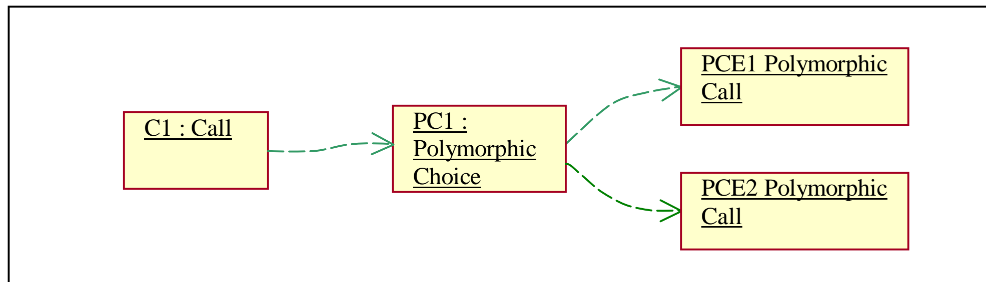
Dans ce chapitre nous avons présenté les raisons pour lesquelles il est nécessaire d'étudier les dépendances à l'intérieur des logiciels construits avec l'Object Modeler, le niveau de granularité où devra être réalisée l'étude et finalement nous avons introduit les concepts de l'Object Modeler.

Nous avons discuté en particulier sur un type de dépendance, celle d'*utilisation*, qui a lieu entre une implémentation et une interface et qui à un niveau supérieur d'abstraction génère des

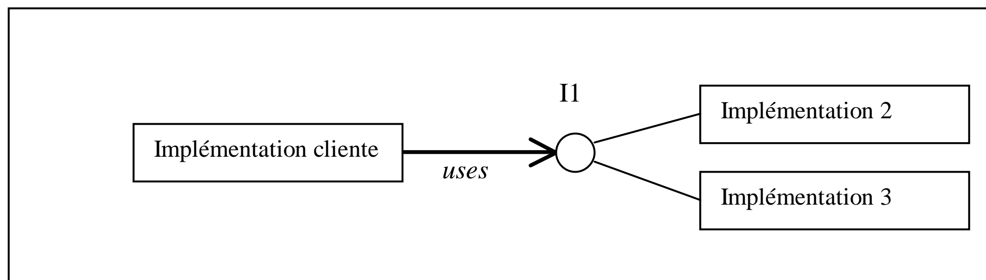
dépendances entre les composants. Une caractéristique importante de ce type de dépendance est qu'elle n'est résolue qu'en temps d'exécution, ce qui la rend impossible à connaître de manière exacte lors d'une analyse purement statique du logiciel. Pendant cette analyse, cette relation apparaît comme un 'éventail' qui représente les possibilités multiples d'implémentation d'une interface.

Cette situation est très similaire à celle des appels polymorphiques qui ont lieu à un niveau d'abstraction plus bas, celui des méthodes à l'intérieur des classes et dont nous avons parlé dans la section correspondante aux graphes de dépendance orientés objet.

Niveau Méthode :



Niveau Elément :



Cette similarité est très importante, car nous pourrons essayer d'adapter les solutions qui existent déjà pour le niveau Méthode au niveau Elément en ce qui concerne les destinations multiples d'une dépendance.

CHAPITRE 4 : Graphe de dépendance et Découpe pour l'OM

Dans la section précédente nous avons introduit le modèle OM. Nous avons mentionné les raisons pour lesquelles nous cherchons à étudier les dépendances à l'intérieur des logiciels construits avec celui ci, ainsi que les difficultés qui existent pour réaliser cette étude. Dans ce chapitre, nous introduirons l'OMDG, la découpe à l'intérieur de celui ci et les applications qui peuvent être obtenues à partir de ces techniques.

4.1 Graphe de Dépendances de l'Object Modeler

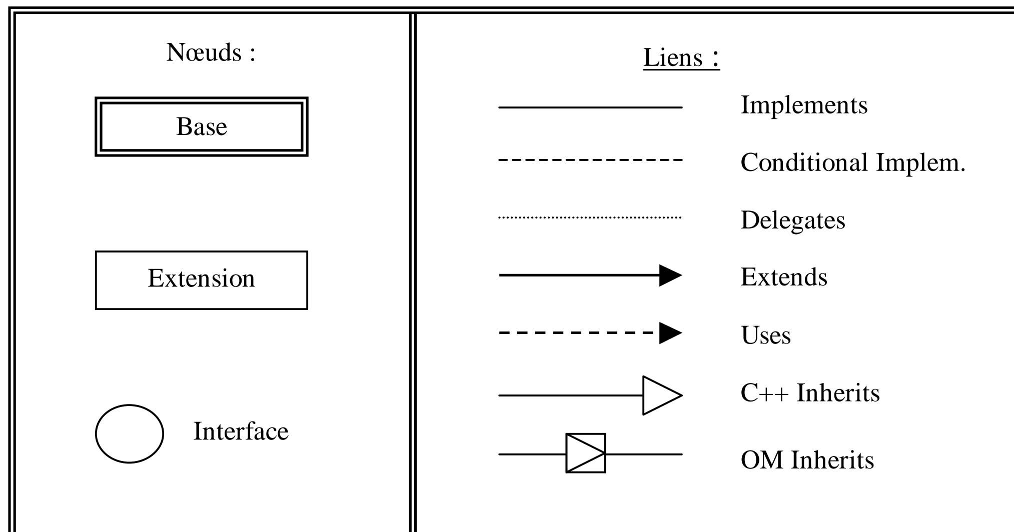
Dans cette section, nous allons proposer un nouveau type de graphe que nous appellerons OMDG (Object Modeler Dependence Graph) et qui nous permettra de représenter les logiciels construits avec l'Object Modeler.

4.1.1 Eléments du Graphe

L'OMDG est une représentation sous forme de graphe (nœuds, liens) des logiciels construit suivant le modèle OM. Cette représentation est réalisée à partir des concepts du niveau *élément* introduits dans le chapitre précédent ainsi que de l'ensemble des relations qui existent entre eux. Les nœuds et liens du graphe sont les suivants :

Entités (nœuds):
- Interfaces
- Implémentations de Base
- Extensions

Relations (liens):
- Implémentation (Standard, Conditionnelle et Délégation)
- Extension
- Héritage (OM/C++)
- Dépendance (uses)

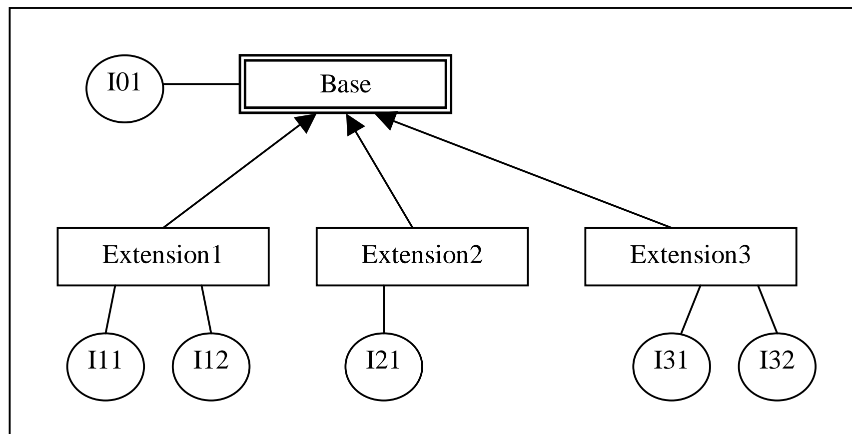


La relation d'implémentation et ses variantes (conditionnelle, délégation) ne sont par dirigées, car elles peuvent être suivies dans les deux sens dans le graphe, soit à partir d'une implémentation pour trouver les interfaces qu'elle implémente, soit d'une interface, pour trouver les implémentations qui l'implémentent.

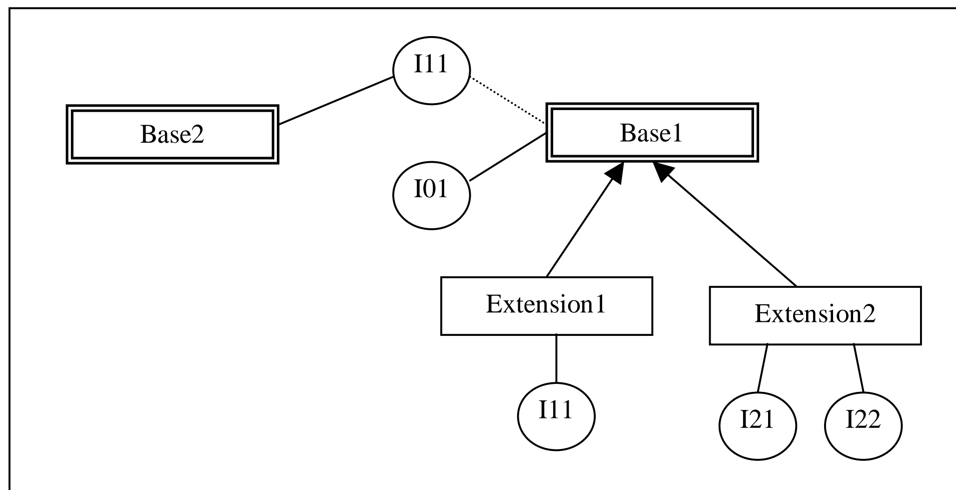
4.1.2 Exemples

Suivant cette syntaxe, nous pouvons représenter les divers concepts présentés auparavant :

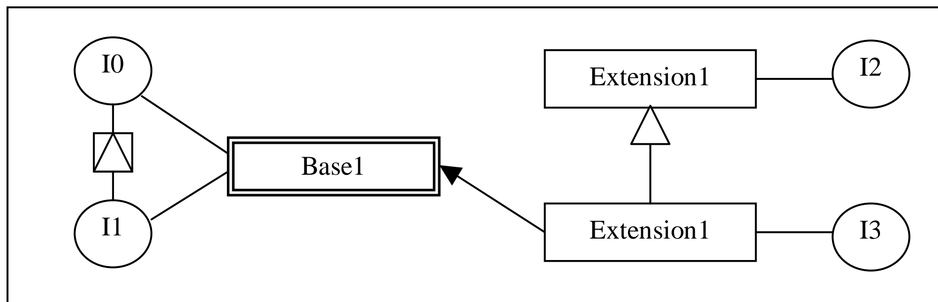
Exemple de composant simple:



Exemple de composant (Base1) qui délègue une interface implémentée par le composant Base2 :



Exemple de composant qui possède une interface qui hérite au sens OM d'une autre. Une des extensions de la base hérite au sens C++ d'une autre.



Autre exemple :

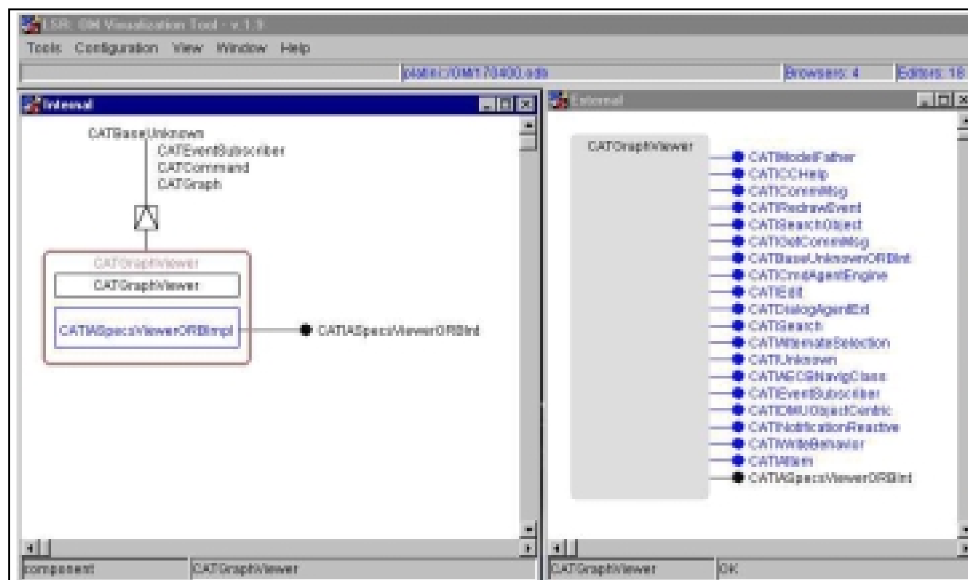
Dans cet exemple nous pouvons voir quatre composants différents. Le composant Base2 hérite de Base1 et il implémente l'interface I21 qui est aussi implementée par Base3 et deleguée par Base5. Le composant Base4 est client de I21 à travers Base5. Nous pouvons remarquer que dans cet

exemple il est impossible de savoir laquelle des deux implémentations (Base2 ou Base3) prend en charge la réalisation de l'interface I21.

4.1.3 Obtention du Graphe

L'obtention d'une partie de L'OMDG est envisageable grâce à divers travaux qui ont été réalisés au laboratoire LSR dans l'équipe Adèle [9]. Ces travaux ont abouti à un prototype qui permet de visualiser les composants ainsi que les interfaces et les implémentations qu'ils contiennent, cependant les dépendances d'utilisation n'apparaissent pas. La syntaxe que nous avons employée dans l'ODMG reprend les mêmes éléments qui ont été utilisés pour ce prototype.

La figure ci-dessous est une capture d'écran obtenue à partir du prototype de visualisation de l'OM, qui montre a gauche un composant et les éléments qui le conforment et a droite une vue 'externe' (celle qu'aura un client) de celui ci.



Nous avons décrit dans la section 3.3.1 deux solutions pour obtenir les relations de dépendances, cependant elles n'ont pas encore pu être testées car elles nécessitent d'avoir accès au code.

4.1.4 Application : Compréhension du logiciel

Comme nous avons mentionné auparavant, la compréhension de la structure du logiciel un besoin indispensable pour maintenir la liaison entre le code et son architecture.

L'OMDG peut être employé pour visualiser la structure du logiciel, ce qui est une manière 'directe' de la comprendre ou au moins d'avoir une idée générale de celle ci. Le fait d'avoir une "vue" du logiciel et de ses relations dans l'ensemble ou dans le détail peut être très utile à l'observation de situations particulières, comme l'utilisation excessive de certains éléments ou au contraire la non utilisation d'autres éléments.

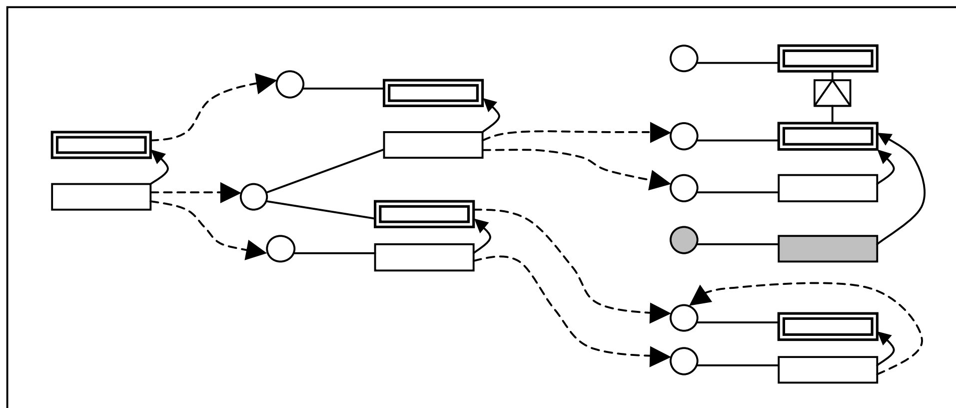
La visualisation de l'ODMG est cependant "imprécise" du au fait que l'ODMG est une représentation statique du logiciel et que ceci implique que les dépendances d'utilisation présenteront un "éventail" de destinations possibles.

Il existe néanmoins la possibilité de complémenter l'OMDG par de l'information obtenue lors de l'exécution du logiciel. Nous allons présenter à la suite le principe de ces deux variétés de visualisation.

4.1.4.1 Visualisation 'statique' du logiciel

Une application qui peut être réalisée de manière directe à partir de l'information contenue dans l'OMDG est la visualisation 'statique' de l'architecture du logiciel. Dans ce cas les choix multiples dans les relations d'implémentation sont montrés dans la représentation. L'utilité de ce type de visualisation est qu'il est possible d'apprécier quels sont les éléments du logiciel les plus utilisés, ainsi que ceux qui ne le sont pas, et ceci en observant le nombre de dépendances qui existent entre les éléments.

Exemple :

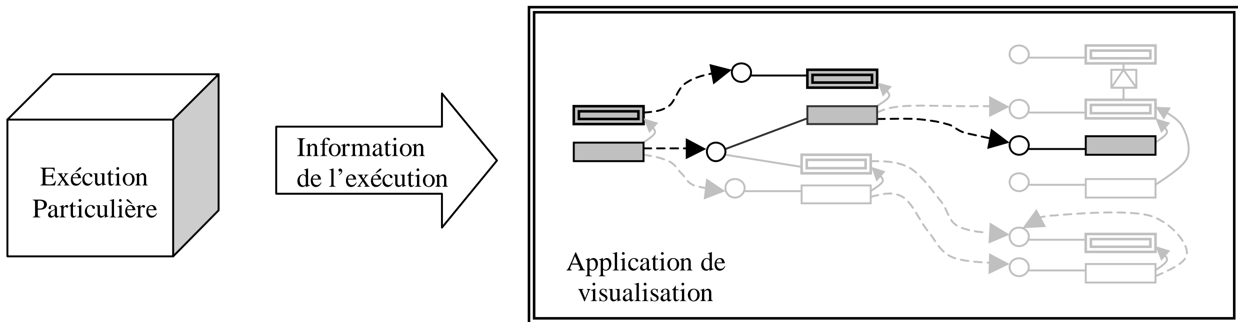


Dans cet exemple simple, nous montrons certains éléments ainsi que leurs dépendances. Visuellement il est possible identifier une extension qui n'est pas utilisée (en gris). Cet exemple est très simple, et en raison de la taille du logiciel les représentations réelles seront beaucoup plus complexes.

4.1.4.2 Visualisation 'dynamique' du logiciel

La représentation visuelle statique peut être complétée avec de l'information obtenue pendant l'exécution du logiciel, ceci permettrait de voir par exemple en temps d'exécution quels sont les éléments qui jouent un rôle lors d'une exécution particulière.

L'information de l'exécution du logiciel doit être générée par les éléments mêmes du logiciel et elle peut être reçue par une autre application qui prend en charge l'affichage du graphe. L'ajout d'informations d'exécution est décrit dans la partie concernant la découpe dynamique de l'OMDG.



Dans ce schéma nous représentons les divers éléments nécessaires à ce type de visualisation. L'application montre ici une section de l'ODMG et les éléments qui sont exécutés sont rehaussés.

Il est possible d'envisager diverses possibilités pour la visualisation du logiciel en exécution, par exemple en changeant de couleur les éléments chaque fois qu'ils sont exécutés selon le nombre de fois qu'ils ont "pris le contrôle" de l'application, de manière à observer non seulement quels sont les éléments qui contribuent pour une exécution particulière mais aussi leur importance dans celle ci.

4.1.4.3 Détection d'éléments non utilisés

La visualisation du graphe peut être utile aussi pour repérer des éléments du logiciel qui ne sont pas utilisés, et qui peuvent être identifiés par le fait qu'il n'existe pas de relations vers eux. Ceci est particulièrement utile dans le cas de l'identification d'interfaces qui ne sont pas utilisées.

4.1.5 Synthèse

Nous avons décrit dans cette section l'OMDG, un type de graphe de dépendances permettant de représenter les programmes construits avec l'Object Modeler. Nous avons montré des exemples qui illustrent diverses situations qui peuvent avoir lieu dans les logiciels construits avec l'Object Modeler.

Nous avons aussi mentionné la viabilité de la construction de ce type de graphes et finalement nous avons décrit brièvement un type d'application qui peut être envisagé de façon directe avec l'ODMG et qui est la visualisation du graphe, soit de manière statique soit en apportant de l'information de l'exécution, c'est à dire de manière dynamique.

4.2 Découpe de programmes construits avec l'Object Modeler

Nous avons suivi dans l'état de l'art l'évolution de la technique de découpe jusqu'à son implémentation dans les graphes de dépendance orientés objet. La différence principale qui existe entre les techniques décrites auparavant et celles que nous allons introduire dans cette section réside dans le niveau d'abstraction où elles se situent. En effet, les divers graphes de dépendance qui sont présentés dans l'état de l'art modélisent le logiciel en prenant en compte le niveau *instruction* tandis que le graphe que nous avons proposé se trouve à un niveau supérieur d'abstraction, celui des *éléments*.

Pour le niveau d'abstraction dans lequel nous nous situons, le critère de découpe que nous allons employer est l'*élément*, c'est à dire que nous allons réaliser la découpe à partir d'un nœud qui correspond soit à une interface soit à une implémentation.

Les découpes dans l'OMDG que nous allons décrire suivent le principe présenté pour les graphes de dépendance, c'est à dire le choix d'un critère et le suivi des dépendances du graphe dans un sens déterminé de façon statique ou dynamique. Cependant les éléments qui sont retrouvés à partir des coupes de l'OMDG représentent des situations différentes à celles des coupes au niveau *instruction*.

4.2.1 Découpe statique arrière

Au niveau *instruction*, les coupes arrière permettent de trouver un ensemble d'intructions qui peuvent affecter un critère choisi. Au contraire, la réalisation de coupes arrière dans l'OMDG nous permettra de connaître l'ensemble d'éléments qui peuvent être affectés de manière directe par l'élément choisi comme critère (ce qui à un sens plus rapproché à celui de la découpe avant au niveau *instruction*).

Nous cherchons à réaliser des coupes arrière de type statique et non exécutables. Ces coupes ne contiendront que les nœuds qui ont une dépendance de manière directe et non transitive au nœud choisi comme critère, la justification de ceci étant que le mécanisme des interfaces et implémentations créé une 'barrière' efficace qui évite la propagation des dépendances (ce qui n'est pas le cas pour les dépendances d'héritage).

Selon le critère choisi, qui peut être une interface ou une implémentation, la coupe regroupe des ensembles d'éléments différents.

4.2.1.1 L'interface comme critère de coupe

Lors du choix d'une interface comme critère de coupe, nous allons trouver l'ensemble des éléments qui dépendent directement de celle ci à travers divers types de liens que nous allons énumérer à la suite:

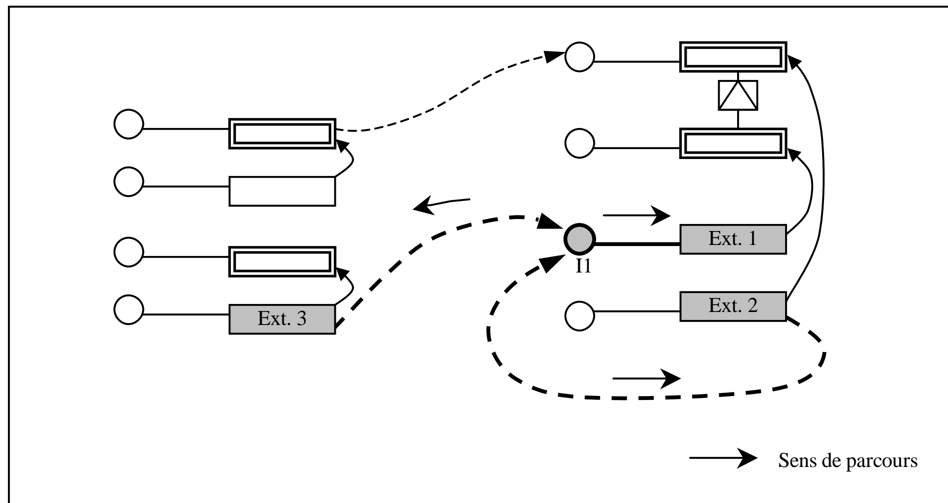
- Liens d'utilisation.

- Liens d'implémentation (implémentation normale, conditionnelle et délégation).

- Liens d'héritage entre interfaces.

- Liens d'héritage entre implémentations pour les implémentations qui utilisent ou implémentent l'interface choisie comme critère.

Exemple :



Ce schéma représente une interface (I1) qui a été modifiée. Les liens qui sont suivis sont ceux d'utilisation qui mènent vers les extensions 2 et 3 ainsi que celui d'implémentation qui mène vers l'extension 1.

4.2.1.2 L'implémentation comme critère de coupe

Lorsqu'une implémentation est choisie comme critère de coupe, les liens qui seront suivis sont les suivants :

- Liens d'héritage entre implémentations.

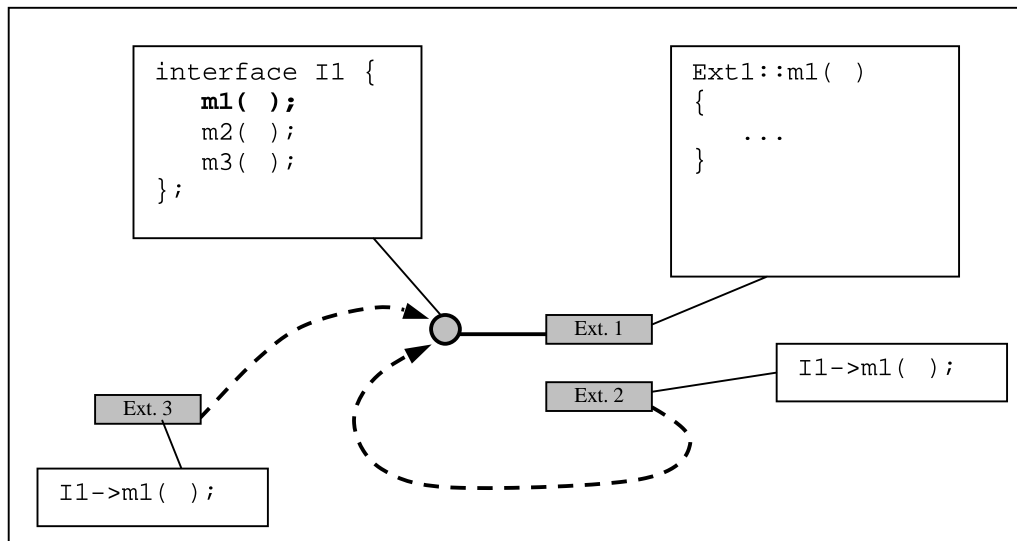
Dans le cas des implémentations comme critère de coupe le nombre de types de liens à suivre est réduit à un seul. Ceci est la conséquence que le lien d'implémentation évite qu'une interface soit liée à une implémentation particulière.

4.2.1.3 Exemple d'application: Analyse d'impact

La découpe statique arrière peut être utilisée pour réaliser de l'analyse d'impact en cas de modification de l'élément choisi comme critère.

Une fois que l'on obtient la coupe, il serait envisageable de travailler à un niveau de granularité inférieur pour faire des analyses plus fines, notamment au niveau des méthodes.

Exemple :



Cet exemple nous montre le résultat de la coupe réalisée auparavant et qui avait pour critère l'interface I1. Si à l'intérieur de I1 la méthode m1 est éliminée nous pouvons retrouver dans les autres éléments obtenus par la découpe les endroits où cette méthode est référencée.

4.2.2 Découpe statique avant

La coupe statique avant que nous présentons ici à pour but de trouver l'ensemble des éléments nécessaires à un nœud particulier pour son exécution.

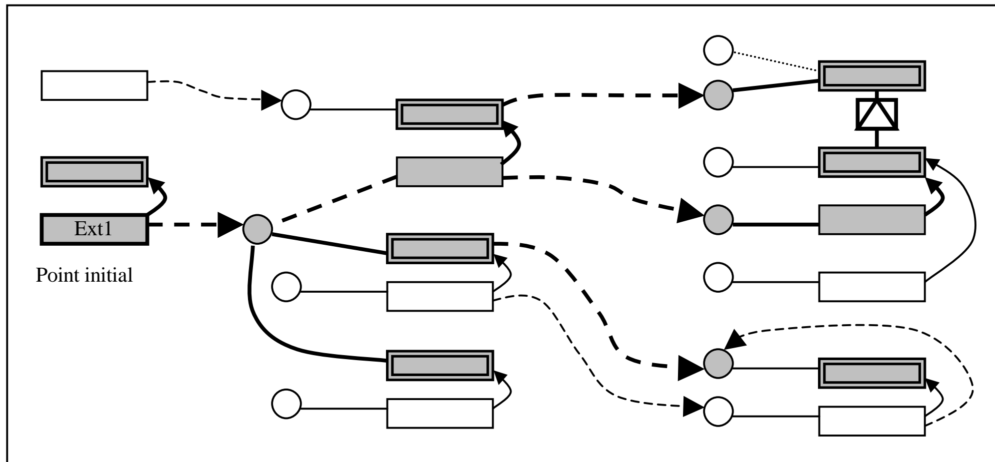
4.2.2.1 Règles de découpe

Pour réaliser ce type de coupe, nous choisirons un noeud quelconque (interface ou implémentation) et les dépendances seront suivies transitivement.

Les liens qui doivent être suivis pour réaliser cette technique sont les suivants :

- Liens d'utilisation.
- Liens d'implémentation : (implémentation normale, conditionnelle et délégation).
- Liens d'extension.
- Liens d'héritage.

Exemple :



Dans cet exemple, nous montrons une coupe avant à partir de l'extension Ext1. L'ensemble des éléments et les relations qui sont impliqués dans l'exécution de Ext1 sont pris en compte (éléments montrés en gris, relations en traits grossis).

4.2.2.2 Exemple d'application: Optimisation de la livraison

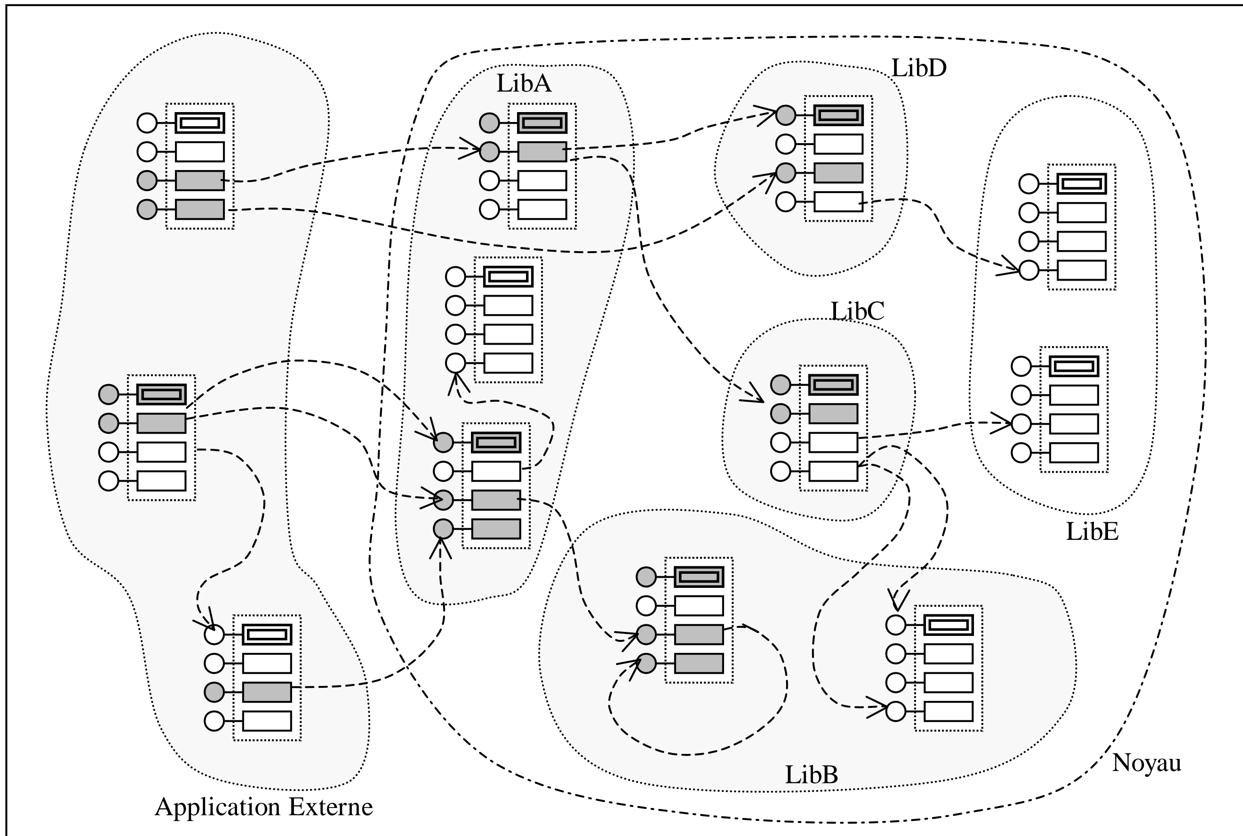
Nous avons mentionné auparavant que le logiciel CATIA est un noyau qui peut être étendu par d'autres applications externes construites avec l'Object Modeler. La construction de ces applications s'effectue en utilisant des librairies qui contiennent le code compilé. Ces librairies sont des regroupements physiques qui contiennent un nombre variable d'implémentations.

Une application particulière de la découpe avant est de connaître quelles sont les librairies nécessaires à l'exécution d'une application donnée. Ceci permettrait d'optimiser la livraison des librairies qui jusqu'à présent sont incluses dans leur totalité à chaque livraison en raison du manque d'informations relatives à ces dépendances.

Pour obtenir cette information il est nécessaire de réaliser des coupes avant sur l'ensemble des éléments de l'application externe qui ont une dépendance avec le noyau. Le résultat de cette étape est un ensemble de nœuds représentant tous les éléments que nécessite l'application externe pour son exécution. A partir de cet ensemble il est possible d'obtenir le nom des librairies qui contiennent ces éléments.

Nous montrons ci-dessous un exemple de ce type de coupe. Dans les frontières représentant l'application externe et les diverses librairies, nous montrons en gris les éléments qui sont inclus dans les coupes. La seule librairie qui ne contient pas des éléments obtenus à partir des coupes est LibE. Ainsi, lors de la livraison des diverses librairies nécessaires pour exécuter l'application externe il n'est pas nécessaire d'inclure la librairie LibE.

Exemple:



4.2.3 Découpe dynamique

Le concept de découpe dynamique que nous avons présenté dans l'état de l'art consiste à obtenir un graphe qui comporte des informations obtenues lors d'une exécution particulière pour y appliquer ensuite les techniques statiques de découpe. Lors de l'exécution, les nœuds des graphes sont marqués comme ayant été exécutés ou non.

L'apport d'informations obtenues lors de l'exécution peut être très utile dans le cas de l'analyse de l'Object Modeler, en raison principalement de la résolution dynamique des dépendances d'utilisation. Cette solution n'est cependant pas complètement précise car elle dépend du 'chemin' suivi lors de l'exécution et certains nœuds qui jouent un rôle important peuvent ne pas être marqués si lors d'une exécution particulière ils ne sont pas exécutés.

Pour réaliser la découpe dynamique de l'OMDG nous reprendrons un principe similaire à celui qui a été présenté pour la découpe au niveau des instructions. Tout d'abord, nous décrirons l'obtention d'informations à partir de l'exécution. Ensuite nous discuterons ensuite la réalisation de la découpe en question.

4.2.3.1 Obtention de l'information d'exécution

L'information que nous devons obtenir lors de l'exécution des logiciels construits avec l'Object Modeler est en relation directe avec les éléments que représente son graphe de dépendance. Il est donc nécessaire de connaître si une interface à été accédée ainsi que de savoir quelle implémentation particulière à pris en charge l'implémentation des appels à cette interface. Il doit être possible aussi de connaître quels liens d'utilisation ont été suivis lors de l'exécution.

La construction de l'Object Modeler rend cette approche possible du fait que les liens d'implémentation sont réalisés la plupart du temps par des objets (appelés TIE) qui sont capables de générer la trace nécessaire. Les implémentations peuvent elles aussi générer une trace de leur exécution. Par ailleurs, il doit également être possible "d'instrumenter" le compilateur de l'OM afin qu'il génère la trace par exemple à chaque appel du *QueryInterface*, ou à chaque chargement d'une librairie dynamique.

Nous n'avons pas besoin de faire une distinction entre les diverses instances qui peuvent être créées à partir d'un même type d'élément, car ceci est un aspect qui n'est pas modélisé dans l'OMDG.

Les éléments qui doivent générer une trace de leur exécution sont les suivants:

- Implémentations.
- Relations de type *uses*. Cette information peut être générée par les implémentations, par exemple lors des *QueryInterface*.
- Relations de type *impléments*. Comme nous avons mentionné, cette information peut être générée par les objets qui réalisent la relation.

La trace générée pourrait correspondre à l'exemple suivant :

```
BaseA en exécution
BaseA : QueryInterface(I1)
BaseA : QueryInterface(I2)
TIE : Source = I1, Destination = ExtensionB
ExtensionB en exécution
BaseA en exécution
TIE : Source = I2, Destination = ExtensionC
ExtensionC en exécution
ExtensionC : QueryInterface(I3)
BaseB en exécution
. . .
```

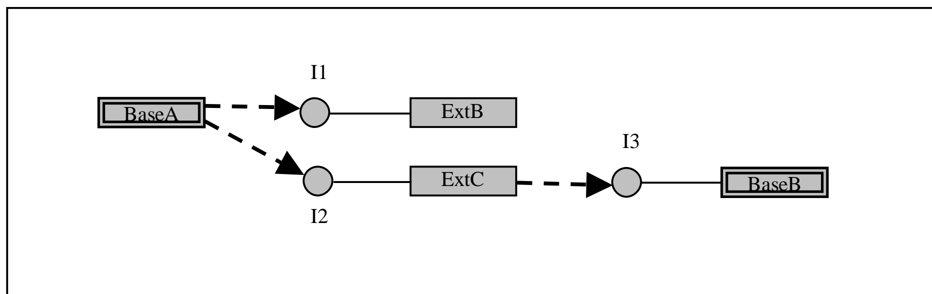
Remarquons que la trace peut être obtenue à partir d'une seule exécution si nous sommes intéressés uniquement par les éléments qui la génèrent. Elle peut être obtenue également à partir

d'une fusion d'un ensemble de traces résultant d'une suite d'exécutions qui vise à marquer un nombre maximum d'éléments qui jouent un rôle dans l'exécution du logiciel en général.

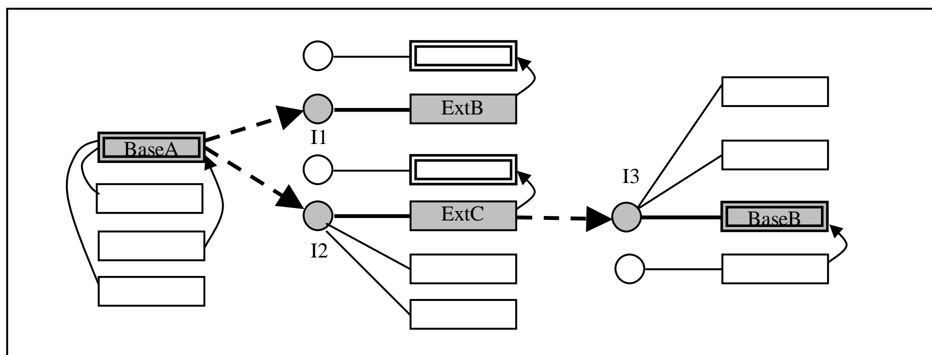
4.2.3.2 Règles de découpe

A partir de la trace obtenue lors de l'exécution, deux choix sont possibles. Le premier consiste à créer un graphe de dépendance uniquement à partir de l'information obtenue lors de l'exécution, et ce graphe peut ensuite être découpé. Le deuxième choix consiste à prendre l'OMDG obtenu de manière statique et à réaliser la découpe sans prendre en compte les éléments qui n'ont pas été exécutés en gardant toutefois ceux qui sont nécessaires pour que la coupe soit cohérente (par exemple les bases lorsque les extensions sont exécutées).

Exemple de graphe construit uniquement à partir de la trace présentée dans l'exemple précédant.



Exemple de graphe complété par l'information de l'exécution.



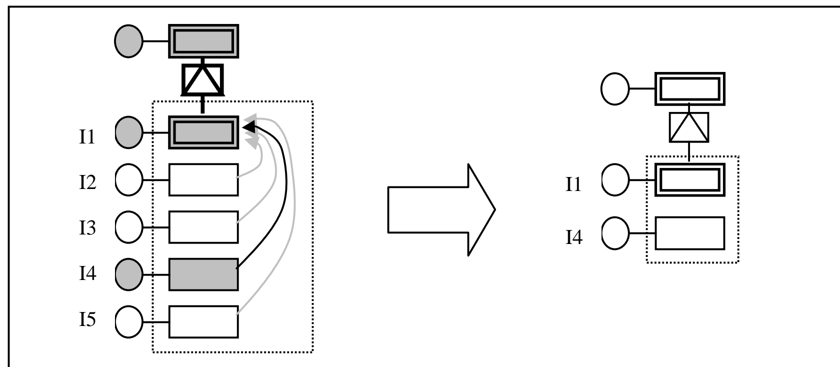
Nous pouvons apprécier que l'apport d'information dynamique permet de résoudre le problème des destinations multiples dans les relations d'implémentation. L'information obtenue dynamiquement permet de prouver qu'un élément a été exécuté, et qu'il est donc utile, cependant, elle ne permet pas de prouver qu'un élément est inutile à l'application.

4.2.4 Découpe d'Interfaces

Il est possible de reprendre le principe proposé par la technique de découpe d'interfaces et de l'appliquer à l'intérieur des composants.

Dans ce cas le critère de découpe est une interface et cette découpe est réalisée en réalisant une fermeture transitive des nœuds du graphe en suivant les dépendances qui existent entre eux. Puisque cette coupe est réalisée à l'intérieur d'un composant seuls les liens d'héritage sont suivis vers des éléments qui se trouvent en dehors du composant choisi. Les liens d'utilisation sont suivis seulement dans le cas ou ils relient une interface et une implémentation du même composant.

Exemple:



Dans cet exemple, le critère de coupe est l'interface I4, le lien d'extension est suivi vers la base qui est incluse dans la coupe ainsi que son implémentation et une autre base dont elle hérite.

4.2.5 Synthèse.

Dans cette partie nous avons décrit les diverses variétés de découpe appliquées au graphe de dépendance de l'Object Modeler ainsi que les applications qui peuvent être obtenues à partir de ces coupes, nous résumons ceci dans le tableau suivant.

Type de découpe :	*Critère :*	*Liens suivis :*	*Application :*
Statique Arrière	Interface ou Implémentation	- utilisation - implémentation - héritage	Analyse d'impact
Statique Avant	Interface ou Implémentation	- utilisation - implémentation - extension - héritage	Optimisation de la livraison
Dynamique		Selon découpe	Obtenir un graphe qui représente une ou plusieurs exécutions pour réaliser ensuite la découpe.
D'interfaces	Interface	Liens à l'intérieur d'un composant sauf héritage qui peut être en dehors	Simplification des fonctionnalités des composants.

4.3 Conclusion

Dans ce chapitre nous avons décrit premièrement le graphe de dépendances de l'Object Modeler, qui permet de représenter les logiciels construits avec celui-ci.

Les caractéristiques propres à l'Object Modeler font que ce graphe ne peut pas être une représentation exacte du logiciel de manière statique, car il est en général impossible de connaître quelle implémentation particulière implémente une interface lorsqu'un lien d'utilisation aboutit à une interface.

Le graphe peut cependant être utilisé tel qu'il est pour visualiser la structure du logiciel, soit de manière statique, soit dynamique.

Nous avons décrit ensuite diverses variétés de découpe appliquées à l'ODMG. Les coupes dans ce graphe sont réalisées à un niveau d'abstraction supérieur à celles que nous avons décrit dans l'état de l'art. Ceci, en plus des caractéristiques de l'Object Modeler, fait que les coupes au niveau *élément* ont un objectif différent à celui des coupes du niveau *instruction*.

Il est important de noter que certaines des caractéristiques du graphe de dépendances de l'Object Modeler sont similaires à celles qui existent à un niveau d'abstraction inférieur, nous pouvons citer notamment le choix multiple dans les relations et la similitude des composants avec les classes. Ceci à permis d'envisager l'utilisation dans l'OMDG de méthodes similaires, comme la découpe d'interfaces entre autres, à celles qui avaient été proposées pour un niveau d'abstraction inférieur.

Diverses applications pouvant être réalisées à partir des différentes techniques de découpe ont été présentées, cependant cette liste n'est pas exhaustive, et il est envisageable de trouver d'autres applications pouvant être obtenues à partir de méthodes similaires.

CHAPITRE 5 : Conclusions et Perspectives

5.1 Synthèse

A partir d'un besoin réel qui est celui de connaître les dépendances qui existent à l'intérieur des logiciels construits avec l'Object Modeler, nous avons proposé un type de graphe de dépendances (l'OMDG) permettant de représenter ces logiciels sous forme de graphe.

Nous avons ensuite étudié diverses variétés de découpes appliquées à ce graphe de dépendance ainsi que certaines applications que l'on peut réaliser à partir des graphes et des coupes, notamment:

- La visualisation du graphe de dépendances.
- L'analyse d'impact lors de modifications.
- L'étude des éléments nécessaires à l'exécution d'un élément particulier du graphe, afin d'optimiser la livraison des librairies par exemple.
- La simplification des fonctionnalités à l'intérieur des composants.

5.2 Conclusions

La proposition présentée dans ce travail peut être vue comme une suite logique à l'évolution des graphes de dépendance réalisés antérieurement.

En effet, l'Object Modeler permet de construire des composants qui sont des assemblages d'objets et qui représentent une évolution par rapport au modèle objet conventionnel. Pour cette raison, le graphe qui permet de le représenter les dépendances dans l'OM complète les propositions réalisées auparavant.

L'évolution des graphes est schématisée dans le tableau suivant :

Nom du Graphe :	*Type de Programme Représentés :*	*Niveau d'abstraction :*
Graphe de dépendance de programme (PDG).	Monolithiques	Instruction
Graphe de dépendance de Système (SDG)	Composé de multiples procédures	Procédure / Instruction
Graphe de dépendance Orienté Objet	Orientés Objet	Classe / Méthode / Instruction
Graphe de dépendance de l'Object Modeler (OMDG)	Construits avec l'OMDG	Interface / Implémentation

Il est intéressant de noter que lorsque nous avons analysé les caractéristiques de l'Object Modeler nécessaires à la réalisation du graphe de dépendances, nous avons remarqué qu'il existait des situations similaires à celles qui sont rencontrées dans les graphes de dépendance orientés objet mais à un niveau d'abstraction inférieur.

Nous pouvons souligner l'analogie qui existe entre certains concepts dans les deux niveaux de granularité:

Concepts du niveau Orienté Objet	*Concepts de l'Object Modeler*
Classe	Composant
Signature de Méthode	Interfaces
Corps de Méthode	Implémentations
Appel de Méthode	Dépendance d'utilisation

Cette analogie à permis d'envisager la possibilité d'appliquer les techniques de découpe utilisées dans les graphes existants mais cette fois dans l'OMDG.

A cause des contraintes sur l'information disponible pour la réalisation de ce travail, nous avons du réaliser une proposition assez générale, qui pourra être complétée postérieurement à l'aide de nouvelles informations.

La proposition des techniques de découpe n'a pas pu être appliquée car elle dépend de l'obtention du graphe de dépendance. Nous avons confiance cependant dans la possibilité de réaliser ces techniques une fois les graphes construits.

5.3 Perspectives

La proposition que nous avons donné du graphe de dépendance OM reste spécifique au modèle construit par Dassault Systèmes, ce qui n'est pas en accord avec la philosophie des propositions antérieures des graphes de dépendance, qui ont cherché à rester indépendantes d'un langage spécifique.

Ceci est dû au fait que le modèle crée par Dassault Systèmes n'est pas un modèle à composants défini très clairement et les caractéristiques particulières qu'il possède obligent à créer une représentation spécifique comme celle que nous avons présentée.

Une perspective de ce travail serait de définir un graphe de dépendances à composants, qui permettrait de représenter les logiciels construits selon ce paradigme. Ceci peut être d'un grand intérêt du fait que la construction des logiciels à base de composants semble avoir beaucoup d'avenir dans le domaine du génie logiciel.

Une autre perspective serait de réaliser des tests permettant de valider les diverses propositions qui ont été décrites dans ce travail.

Le manque de temps et surtout d'informations nous à empêché de pouvoir construire des prototypes pour visualiser les dépendances, cependant les propositions que nous faisons dans ce travail visent à être intégrées dans le prototype qui est construit au laboratoire LSR et que nous avons présenté dans la section 4.1.3.

BIBLIOGRAPHIE

[1] Susan Horwitz, Tomas Reps
"The Use of Program Dependence Graphs in Software Engineering"
Proceedings of the 14th International Conference on Software Engineering, May 1992

[2] Anand Krishnaswamy
"Program Slicing : An Application of Object Oriented Program Dependency Graphs"
Technical Report, July 1994
http://www.cs.clemson.edu/techreports/94-108

[3] Loren D. Larsen, Mary Jean Harrold
"Slicing Object Oriented Software"
Technical Report, March 1995
http://www.cs.clemson.edu/techreports/95-103

[4] Suzan Horwitz, Tomas Reps, David Binkley
"Interprocedural Slicing Using Dependence Graphs"
ACM TOPLAS Vol. 12 No. 1, January 1990

[5] David Binkley, Keith Brian Gallhagher
"Program Slicing"
Advances In Computers

[6] Frank Tip
"A Survey Of Program Slicing Techniques"
Technical Report CS-R9438, 1994

[7] Jon Beck, David Eichmann
"Program and Interface Slicing for Reverse Engineering"
0270-5257/93, 1993 IEEE

[8] "Why CNext ?"
Dassault Systèmes Internal Documentation. Version 1.0

[9] Divers travaux réalisés au LSR dans l'équipe Adèle.
Serveur de l'équipe : http://www-lsr.imag.fr/adele.html

[10] John J. Marciniak
Encyclopedia of Software Engineering, Volume 2 (O-Z) p873-877

[11] C. Steindl
"Program Slicing For Object Oriented Languages"
Thèse, Johannes Kepler University Linz, April 1999
http://www.ssw.uni-linz.ac.at/Staff/CS/Slicing.html

[12] Susan Horwitz, Jan Prins, Thomas Reps
"On the Adequacy of Program Dependence Graphs for Representing Programs"
Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages.

[13] Thomas Reps
"Program Analysis via Graph Reachability"
Information and Software Technology *40*, 11-12 (Nov. / Dec. 1998), pp. 701-726.

[14] Christoph Steindl
"Static Analysis of Object Oriented Programs"
9th ECOOP Workshop for PhD Students in OOP. 1999

[15] Brian A. Malloy, John D. McGregor, Anand Krishnaswamy, Murali Medikonda
"An Extensible Program Representation for Object Oriented Software"
SIGPLAN Notices, Volume 29, Number 12, December 1994

[16] Hermant D. Pande, Barbara G. Ryder
"Static Type Determination for C++"
USENIX Sixth C++ Technical Conference, April 1994

SOMMAIRE

RESUME 1
CHAPITRE 1 : Introduction 2
1.1 Problématique 2
1.2 Contexte de l'étude 2
1.3 Objectifs 2
1.4 Plan du document 3
CHAPITRE 2 : Graphes de dépendance et Découpe de programmes 4
2.1 Graphes de dépendance 4
2.1.1 Le Graphe de Dépendance de Programmes 4
2.1.2 Le Graphe de Dépendance de Système 7
2.1.3 Le Graphe de Dépendance Orienté Objet 9
2.1.3.1 Représentation des classes 10
2.1.3.2 Création d'instances 12
2.1.3.3 Héritage 12
2.1.3.4 Polymorphisme 13
2.1.4 Niveaux de granularité 15
2.1.5 Synthèse 16
2.2 Découpe de programmes 17
2.2.1 Origines de la technique de Découpe 17
2.2.2 Caractéristiques des coupes 18
2.2.3 Découpe du Graphe de Dépendance de Programmes 19
2.2.4 Découpe du Graphe de Dépendance de Système 20
2.2.5 Découpe du Graphe de Dépendance Orienté Objet 21
2.2.6 Autres variétés de Découpes 22
2.2.6.1 Découpe Avant 22
2.2.6.2 Découpe Dynamique 22
2.2.6.3 Découpe d'Interfaces 23
2.2.7 Applications de la découpe 24
2.2.8 Synthèse 25
2.3 Conclusion 25
CHAPITRE 3 : L'Object Modeler : Un cas d'étude 26
3.1 Introduction 26
3.1.1 Origines de l'Object Modeler 26
3.1.2 Le besoin d'étudier les dépendances 26
3.1.3 Granularité et choix des éléments de l'étude 27
3.2 Concepts de l'Object Modeler 28
3.2.1 Entités du niveau "élément" 28
3.2.1.1 Interfaces 28
3.2.1.2 Implémentations 28
3.2.2 Entités du niveau "regroupement" 29
3.2.2.1 Regroupements conceptuels 29

3.2.2.1 Regroupements physiques 29
3.2.3 Relations de regroupement en composants 30
3.2.4 Synthèse 33
3.3 Relations de dépendance **34**
3.3.1 La relation *uses* 34
3.3.2 La relation *implements* 36
3.3.3 Synthèse 37
3.4 Conclusion **37**
CHAPITRE 4 : Graphe de dépendance et Découpe pour l'OM *39*
4.1 Graphe de Dépendances de l'Object Modeler **39**
4.1.1 Eléments du Graphe 39
4.1.2 Exemples 40
4.1.3 Obtention du Graphe 42
4.1.4 Application : Compréhension du logiciel 42
4.1.4.1 Visualisation 'statique' du logiciel 43
4.1.4.2 Visualisation 'dynamique' du logiciel 43
4.1.4.3 Détection d'éléments non utilisés 44
4.1.5 Synthèse 44
4.2 Découpe de programmes construits avec l'Object Modeler **45**
4.2.1 Découpe statique arrière 45
4.2.1.1 L'interface comme critère de coupe 45
4.2.1.2 L'implémentation comme critère de coupe 46
4.2.1.3 Exemple d'application: Analyse d'impact 46
4.2.2 Découpe statique avant 47
4.2.2.1 Règles de découpe 47
4.2.2.2 Exemple d'application: Optimisation de la livraison 48
4.2.3 Découpe dynamique 49
4.2.3.1 Obtention de l'information d'exécution 50
4.2.3.2 Règles de découpe 51
4.2.4 Découpe d'Interfaces 51
4.2.5 Synthèse. 52
4.3 Conclusion **53**
CHAPITRE 5 : Conclusions et Perspectives *54*
5.1 Synthèse **54**
5.2 Conclusions **54**
5.3 Perspectives **55**
B I B L I O G R A P H I E *57*