Institut Supérieur d'Informatique de
Modélisation et de leurs Applications

Campus des Cézeaux
24 avenue des Landais
BP 10125
63173 AUBIÈRE Cedex

Rapport d'Ingénierie Dirigée par les Modèles
Filière Génie Logiciel et Systèmes Informatiques

Conception et Implémentation du Modèle Objet : 1ére partie

Présenté par : **Nicolas PRUGNE & Antoine COLMARD**

Table des matières

Table des figures et illustrations

Introduction

L'ingénierie des modèles (IDM) est une discipline relativement récente en informatique et qui trouve sa source dans un problème récurrent du génie logiciel. En effet, bien souvent le code produit par les ingénieurs en charge d'un programme informatique satisfait pleinement les spécifications de ce dernier. Cependant si ces spécifications viennent à évoluer, comme c'est le cas lors de mises à jour ou lors de sorties de nouvelles versions d'un produit par exemple, il est fort probable que ce code soit assez difficile à faire évoluer. Ce manque d'évolutivité peut alors conduire les personnes responsables du développement à repasser par une phase de conception très couteuse pour l'entreprise qui finance le produit.

En réponse à ce problème l'*Object Management Group* (OMG) a décidé d'introduire, à la fin du XXème siècle, une technique de conception baptisée *Model Driven Architecture* (MDA) qui n'est rien d'autre qu'une variante particulière de l'IDM. Cette discipline propose de recentrer la production de logiciels autour de la conception de modèles.

Dans un processus de développement classique, les ingénieurs élaborent des modèles à partir des spécifications du soft qu'ils doivent produire. Ensuite, il leur est éventuellement possible de générer le squelette du code correspondant grâce à des outils de génie logiciel. De là, s'ensuit alors une phase d'écriture et de maintenance du code avec tous les problèmes que cela comporte, notamment lorsque la modélisation du problème devient obsolète.

Dans un processus de développement dirigé par les modèles, l'objectif est d'arriver à produire des outils qui vont permettre aux ingénieurs de se concentrer uniquement sur la conception et sur la maintenance de modèles. A partir des modèles conçus, ces outils vont ensuite être capables de générer automatiquement le code correspondant pour aboutir à un programme fonctionnel. L'objectif étant, d'une part de faciliter le travail des développeurs et d'autre part de réduire fortement les coûts liés à l'étape de modélisation.

Pour comprendre où se situe le lien entre ce TP et l'IDM, il faut voir les concepts du modèle objet comme les spécifications d'un projet à réaliser. D'un point de vue théorique, l'objectif est alors de développer un outil capable de transcrire un modèle objet en programme exécutable. La première chose qu'il est alors nécessaire de définir, pour l'utilisateur, est un moyen de décrire son modèle à l'outil, afin que ce dernier soit en mesure de le transformer en exécutable.

Dans ce TP il a été choisi que le moyen pour décrire un modèle objet serait le langage C++. Ainsi, l'outil qu'il faut développer est un compilateur. Reste à savoir comment faire pour passer du C++ à un programme exécutable. Pour répondre à ce problème, il a été choisi que le langage C servirait de passerelle entre un programme écrit en C++ et un exécutable. Ainsi, le compilateur qu'il faut réaliser devra produire en sortie un code C qui pourra ensuite être compilé en exécutable. Cette technique est inspirée des premiers compilateurs C++ qui étaient des « *cfront* » et qui généraient du C.

Cependant la création d'un compilateur de A à Z nécessite un travail poussé et ne peut se faire sur quelques séances de TP. Pour rappel la compilation d'un programme fait intervenir plusieurs composants en amont du processus qui va réellement générer le code exécutable. Les trois principaux sont l'analyseur lexical, l'analyseur syntaxique et l'analyseur sémantique (Figure 1).

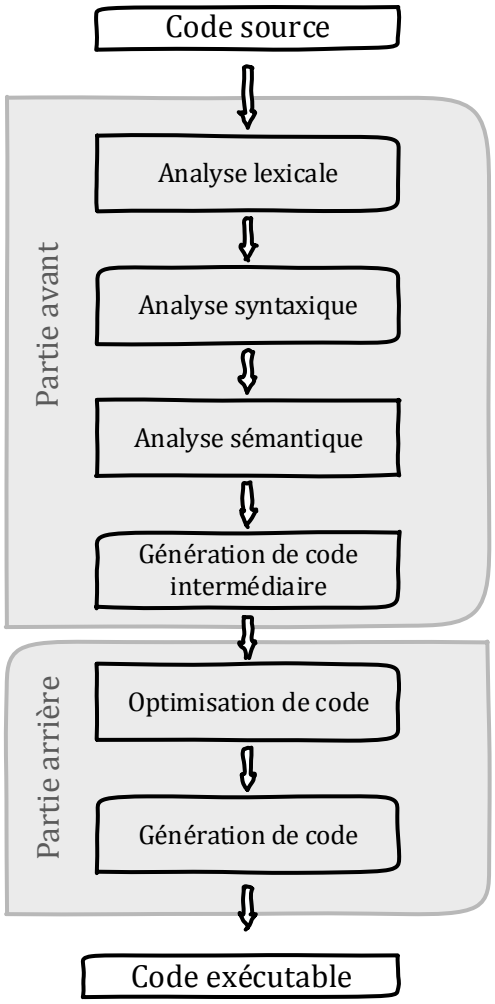

Figure 1 - Schéma d'une chaîne de compilation classique

L'analyse lexicale est un processus qui permet de découper le code source en unités atomiques appelées lexèmes, il peut s'agir de mots clefs du langage, d'identifiants ou de symboles. Cet ensemble de lexèmes va ensuite être traité par l'analyseur syntaxique (ou *parser* en anglais) qui va vérifier que cette suite de jetons est conforme à la grammaire du langage considéré, ici le C++. Si tel est le cas, c'est ensuite l'analyseur sémantique qui prendra la relève et qui aura pour rôle de vérifier que le code en cours de compilation ait une certaine logique, un certain sens. Il sera question, par exemple, de vérifier si les noms des fonctions utilisés dans le code existent bien, ou encore, de contrôler si deux variables impliquées dans une affectation ont des types cohérents.

Bien que ces trois étapes fassent partie du schéma de compilation classique, elles ne sont pas directement liées à de l'IDM, c'est pourquoi, elles ne sont abordées que brièvement dans ce rapport. En effet, ce dernier traite surtout de l'étape de génération de code intermédiaire. Il s'agit d'une phase pendant laquelle le compilateur va retranscrire le code source vers un autre langage, appelé langage intermédiaire, plus adapté pour d'éventuelles optimisations par exemple. De nombreux langages de programmation utilisent le C comme langage intermédiaire et c'est le cas du compilateur dont il est question dans ce TP.

Cependant, l'objectif ici est non pas de créer un outil capable de transformer du C++ en C, mais plutôt de voir comment ce code C pourrait être structuré si réellement il fallait réécrire un compilateur « *cfront* ». Effectivement, bien que le C offre un niveau d'abstraction plus

élevé que des langages comme l'assembleur par exemple, il n'en reste pas moins dépourvu des concepts objets. Il faut donc trouver un moyen d'organiser ce code C afin qu'il puisse émuler les mécanismes d'un langage objet comme le C++. C'est précisément de cette organisation dont il est question dans ce rapport.

Ainsi, le travail à réaliser consiste à implémenter un modèle objet en C en adoptant les principes de la programmation modulaire. Cette tâche nécessite donc d'étudier et de modéliser les principaux concepts objets afin de pouvoir les retranscrire en modules C. Si l'on prend un peu de recul sur ce travail, on remarque qu'il consiste à concevoir un modèle qui décrit les concepts du modèle objet, on parle alors de méta-modèle. Les méta-modèles sont un des outils clefs de l'IDM puisqu'ils permettent de passer d'une modélisation à une autre facilement. Ici la transformation serait celle d'un code C++ décrivant un modèle objet que l'on changerait en un code C retranscrivant exactement les mêmes fonctionnalités que le code d'origine. Tout le problème réside dans la conception du méta-modèle qui permet de passer d'une représentation C++ à son équivalent C. C'est précisément ce dont il est question dans la suite de ce rapport.

1 L'encapsulation

L'encapsulation est l'un des fondements du modèle objet. La partie suivante explique quelles sont ses exigences et comment celles-ci peuvent être émulées dans un code écrit en C.

1.1 Principe

L'encapsulation est un des principes fondamentaux de la programmation orientée objet. Elle préconise de regrouper les données et les méthodes agissant sur ces données dans une même structure appelée objet. De plus, elle stipule que ces données doivent être protégées, c'est-à-dire non accessibles (ou de manière contrôlée via des accesseurs) aux acteurs qui gravitent autour de l'objet. Ainsi, de l'extérieur, l'objet est vu comme une boite noire fournissant un service particulier et avec lequel on communique grâce aux méthodes qu'il expose.

Le choix des données membres et méthodes exposées se fait via les mots clefs `public`, `private` ou `protected`. Cependant, la mise en œuvre de ce principe repose surtout sur l'analyseur sémantique du compilateur qui va contrôler que l'accès aux données et aux méthodes d'un objet s'effectue correctement selon le contexte des appels. Comme ce TP ne traite que de la partie génération de code intermédiaire, il ne s'agit pas réellement d'une contrainte à prendre en compte pour bâtir le méta-modèle.

Ce dernier se concentre plutôt sur la description des procédés qui permettent de générer un module C correspondant à une classe C++. Cette solution est décrite dans la partie suivante.

1.2 Solution

L'encapsulation implique de regrouper deux types de données bien différentes. D'une part il y a les attributs de l'objet, souvent composés par des types primitifs (`int`, `double`, `float`, etc…), et d'autre part ses méthodes.

Pour le premier type de données, le passage du C++ au C se fait relativement bien grâce aux structures de données que propose ce dernier. En effet, pour transformer la description C++ d'un objet ne contenant que des attributs, il suffit de générer le code d'une structure C contenant exactement les mêmes attributs. Pour une meilleure organisation du code source, cette structure pourra être placée dans son propre header (fichier .h) définissant ainsi un module qui pourra être partagé et réutilisé par d'autres composants.

Pour le regroupement de méthodes au sein d'un objet, la transformation en C se révèle plus complexe. La première solution qui vient instinctivement, est celle qui consiste à reprendre le code de chaque méthode de l'objet, afin de les recopier dans le fichier .c du module de la structure précédente. Dès lors, chaque structure de données correspondant à une classe C++ doit embarquer un ensemble de pointeurs de fonctions qui font références aux méthodes recopiées.

Cependant, cette modélisation présente un inconvénient majeur. En effet, avec cette représentation, chaque structure de données contiendra autant de pointeurs de fonctions qu'il n'y avait de méthodes sur l'objet transformé. Ainsi, chaque structure allouée contiendra un ensemble de pointeurs qui feront références exactement aux mêmes adresses, celles des méthodes recopiées. Cette solution offre une certaine facilité dans le sens où la transformation du C++ au C est relativement aisée, mais au moment du *runtime*, elle se révèle très couteuse en termes de ressources mémoires et ne peut être envisagée.

Ainsi pour résoudre ce problème, la solution consiste à dissocier le regroupement des attributs et des méthodes. D'une part, pour chaque objet C++ à transformer, il faudra générer une structure contenant ses attributs, comme expliqué précédemment, mais aussi générer une seconde structure chargée de contenir les adresses des méthodes de l'objet. Cette dernière jouera alors le rôle de méta-structure par rapport à la première. La structure contenant les attributs n'aura alors plus qu'à définir un pointeur vers sa méta-structure pour connaitre les traitements qui lui sont associée.

Cette conception est mise en application à travers un exemple dans la partie suivante.

1.3 Exemple

L'exemple propose d'étudier la transformation d'un objet quelconque décrit en C++ vers sa représentation en C. La classe considérée se nomme `ObjetGraphique` (Figure 2). Dans un programme concret elle pourrait servir à mémoriser les coordonnées d'une figure qu'il serait possible d'afficher à l'écran.

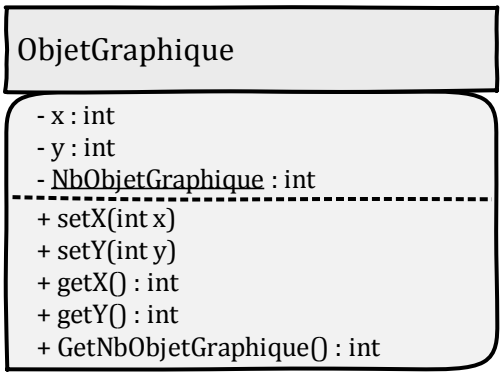

Figure 2 - La classe ObjetGraphique en C++

Avec la solution décrite précédemment, la classe `ObjetGraphique` se verrait traduite en C grâce à deux structures (Figure 3).

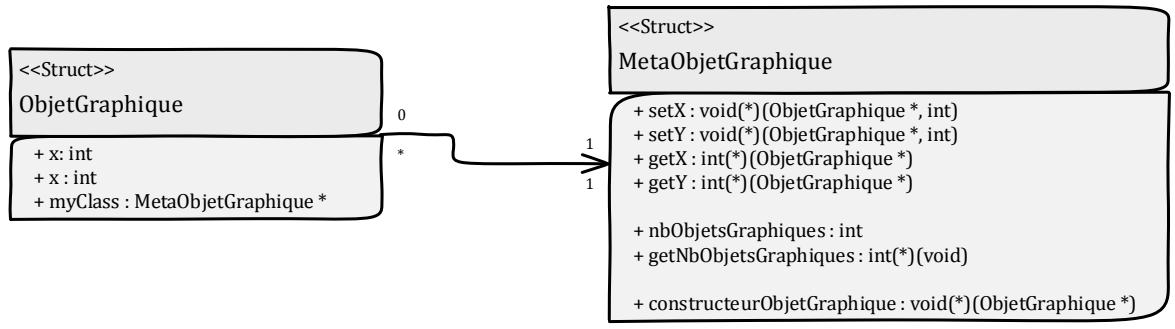

Figure 3 - La classe ObjetGraphique en C

Une première structure contenant les attributs d'instance de la classe, avec les coordonnées `x` et `y`, et une seconde structure contenant les méthodes d'instance de la classe. Une particularité, dont il n'a pas été question dans la partie présentation de la solution, est celle de la gestion des attributs et méthodes de classe. En effet, ici, la classe C++ `ObjetGraphique` contient un attribut de classe appelé `NbObjetGraphique`. Pour modéliser ces attributs en C, il suffit tout simplement de les placer dans la méta-structure de la classe considérée.

Une autre particularité qu'il faut remarquer est que toutes les méthodes d'instance se voient rajouter un pointeur sur un `ObjetGraphique` en plus des paramètres de la méthode d'origine. Ce pointeur permet ainsi de désigner l'objet sur lequel la méthode devra être appliquée, c'est un peu l'équivalent du pointeur `this` en C++. Ce dernier est uniquement accessible dans les méthodes d'instance d'un objet et permet d'accéder à ses attributs.

Enfin, le dernier point important de cette modélisation concerne la construction des objets. C'est aussi un point important de la POO. Les méthodes de construction permettent d'effectuer de nombreuses tâches afin de rendre les objets utilisables comme, l'initialisation des attributs ou l'allocation de certaines données membres par exemple. Lors de la traduction en C, le constructeur se voit bien entendu transformé en pointeur de fonction placé dans la méta-structure et est chargé d'initialiser le pointeur `myClass` de chaque `ObjetGraphique` qu'il construit. Techniquement, ce pointeur doit être mis à l'adresse d'une variable globale qui référence un `MetaObjetGraphique` correctement initialiser.

En pratique dans ce TP, une méthode `allouerMetaObjetGraphique` permet de créer une de ces méta-structures correctement et d'initialiser un pointeur global vers celle-ci afin de la rendre accessible partout dans le code. Au passage la méthode assigne correctement tous les pointeurs de fonctions de la structure aux adresses des implémentations réelles des méthodes. Dans un vrai générateur de code, ces implémentations réelles pourraient être obtenues recopiant le code des méthodes de l'objet transformé vers le module de la méta-structure. La seule contrainte serait

d’ajouter à la liste des paramètres le pointeur sur `ObjetGraphique` dont il était question précédemment.

2 L'héritage

Le second grand principe de la POO s'appelle l'héritage. Cette partie propose de voir comment ce concept a pu être implémenté pour satisfaire les spécifications du TP.

2.1 Principe

L'héritage est la capacité qu'ont les objets à transmettre leurs caractéristiques à d'autres objets qui, ainsi, héritent de ces propriétés. L'héritage implique donc qu'un objet qui descend d'un autre comporte les mêmes données et méthodes membres que son ancêtre, tout en pouvant y rajouter les siennes. On parle de spécialisation pour l'objet qui joue le rôle d'héritier dans cette relation. En effet, ce dernier reprend les comportements de son ancêtre et y ajoute les siens ce qui le rend encore plus spécifique à un cas d'utilisation donné.

Au même titre que la visibilité des membres d'un objet, il existe plusieurs types d'héritages en C++ que l'on peut spécifier à l'aide des mots clefs `private`, `public`, ou `protected`. Ces différents types permettent simplement de changer la visibilité des données membres d'une classe dans le code de ses héritières. Ici encore, la mise en place de ce mécanisme repose plus sur l'analyseur sémantique du compilateur que sur le générateur de code intermédiaire. De ce fait, cette spécification du modèle objet n'est pas vraiment traitée dans ce TP puisqu'on considère qu'elle est déjà implémentée à un niveau supérieur du schéma de compilation.

La partie suivante a pour but d'expliquer comment cette exigence du modèle objet peut venir se greffer sur la modélisation de l'encapsulation faite précédemment.

2.2 Solution

D'un point de vue technique l'héritage implique qu'une classe qui hérite d'une autre contienne physiquement les mêmes attributs et méthodes que son ancêtre. Cette spécification peut parfaitement s'intégrer à la modélisation du problème faite jusqu'à maintenant.

Pour rappel, le générateur de code associe à toute classe C++ une structure et sa méta-structure correspondante. Dès lors, pour qu'il puisse gérer de l'héritage, il suffit que le générateur incorpore dans la structure des classes filles une structure correspondant à la transformation C de la classe mère. Ainsi, dès qu'un lien d'héritage est détecté entre deux classes, ce dernier se contente seulement d'ajouter une structure du type de la classe mère dans le code de la structure fille.

Grace à cette « injection », la structure correspondant à la classe fille contient à la fois ses propres attributs, mais aussi ceux de sa classe mère. En ce qui concerne l'accès aux méthodes de la classe mère, il faut se rappeler que chaque structure, représentant une classe, embarque un pointeur vers la méta-structure qui lui correspond. Ce dernier permet ainsi de remonter jusqu'aux traitements (méthodes) associés à la classe. De ce fait, la classe fille possède automatiquement un lien vers les méthodes de la classe mère.

La partie suivante explique la mise en œuvre de cette solution dans un cas concret.

2.3 Exemple

Cet exemple reprend la classe de type `ObjetGraphique` présentée précédemment et propose de l'étendre à travers deux classes filles de type `Cercle` et `Rectangle` (Figure 4).

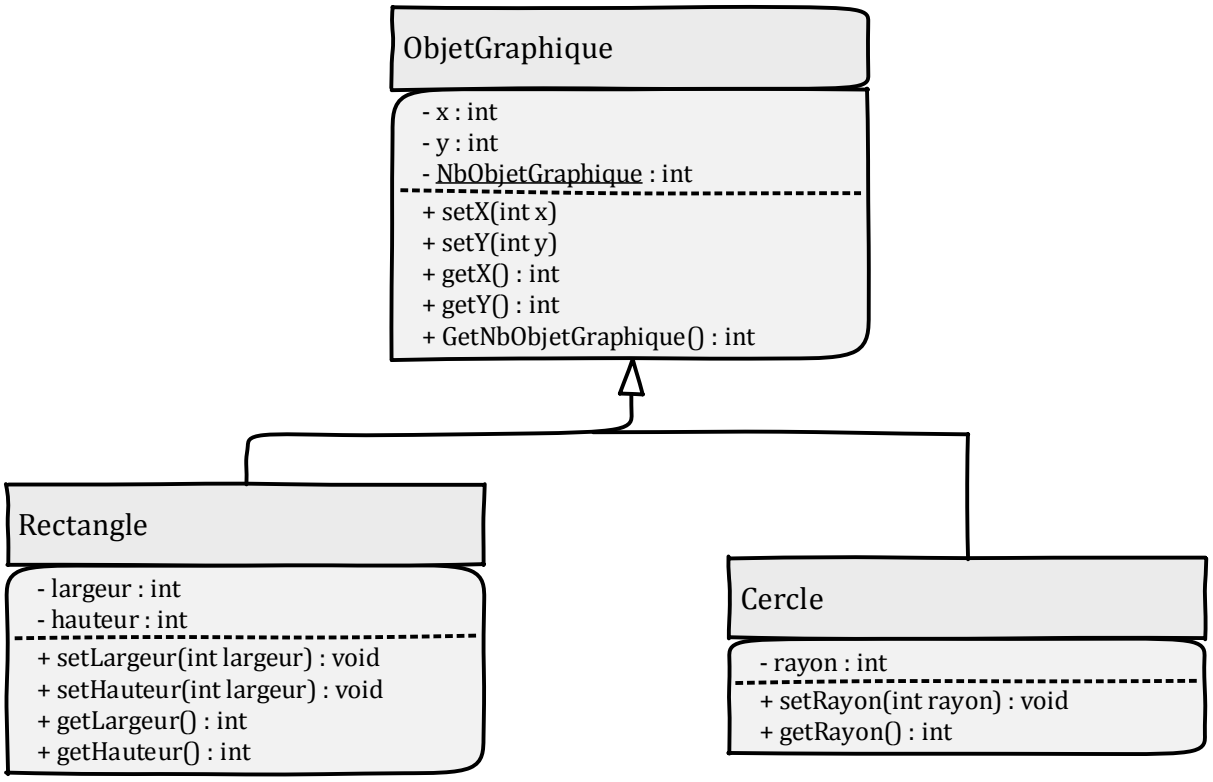

Figure 4 - Relation d'héritage entre les classes ObjetGraphique, Rectangle et Cercle

Avec la solution proposée précédemment, le générateur de code devrait produire au total six structures correspondantes aux descriptions C++ des trois classes précédentes (Figure 5).

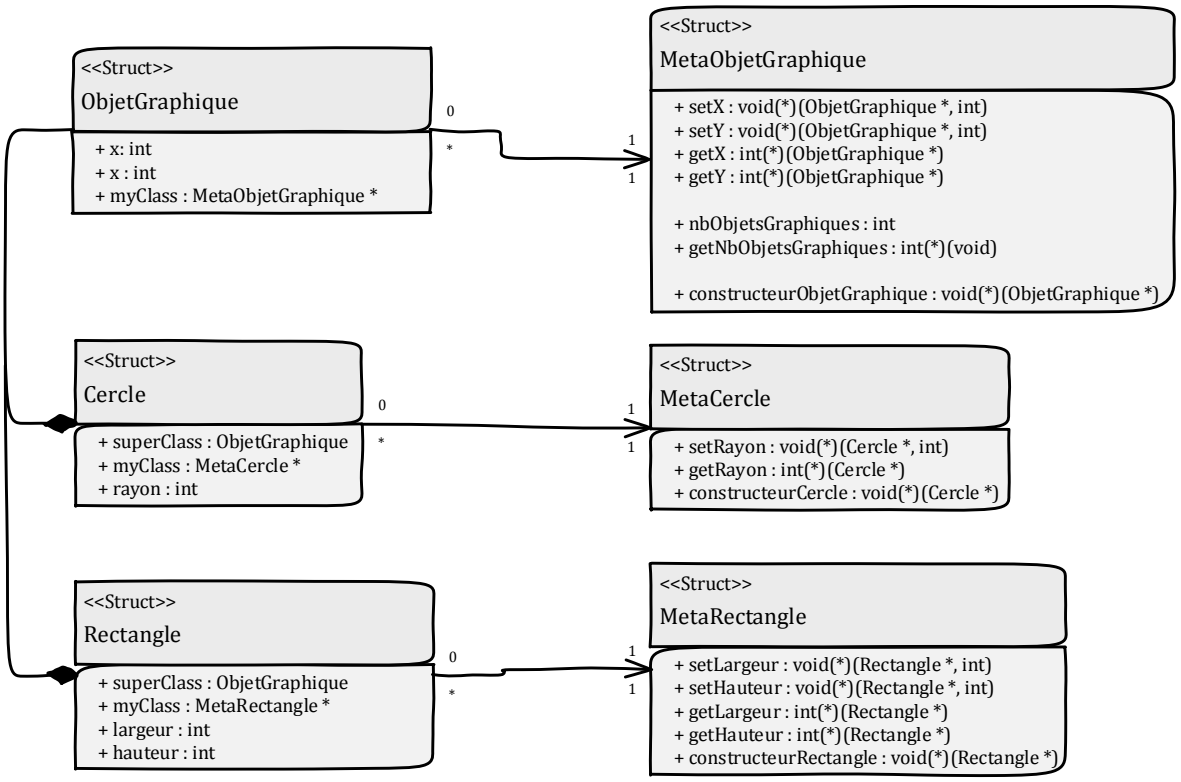

Figure 5 - Relation d'héritage entre les classes, ObjetGraphique, Rectangle et Cercle, implémentée en C

Deux structures de type `Cercle` et `Rectangle` sont alors nécessaires. La première contient un attribut de type entier représentant le rayon d'un cercle et la seconde deux entiers correspondant à la hauteur et largeur d'un rectangle. La relation d'héritage se matérialise par la présence d'une structure de type `ObjetGraphique` parmi les données membres des deux structures `Cercle` et `Rectangle`. Grace à celle-ci, les structures héritières contiennent les attributs de leur classe mère et peuvent remonter jusqu'à la méta-structure de cette dernière via l'indirection suivante : `superClass.myClass`. Ainsi, les comportements de la mère sont également accessibles depuis les structures filles.

En ce qui concerne la gestion des méthodes définies dans les classes filles, le principe est le même que précédemment. Pour chaque classe C++, deux structures sont produites, une structure contenant les attributs et une méta-structure contenant des pointeurs sur les méthodes de la classe. En pratique, la méta-structure est déclarée dans un *header* et les méthodes de la classe sont recopiées dans un fichier .c, le tout définissant ainsi un module. Ce travail resterait relativement simple pour le générateur, cependant il ne lui faudrait pas oublier de rajouter comme paramètre un pointeur du type de la structure traitée (`Rectangle *,` ou `Cercle *`) aux méthodes recopiées. Pour rappel ce pointeur est l'équivalent du `this` lorsque l'on programme en C++.

Enfin, le dernier point important qu'il faut aborder est la construction des classes filles. Avec de l'héritage, cet étape comporte une particularité. En effet, comme une classe fille

contient l'intégralité de ce que sa mère lui lègue, il ne faut oublier d'appeler le constructeur de la mère, dans le code du constructeur de la fille. Ce dernier permet ainsi d'initialiser les données membres de la mère correctement. En C++, il est possible d'appeler explicitement le constructeur d'une classe mère depuis celui de sa fille, mais si ce n'est pas fait, ou si aucun constructeur n'est défini, le compilateur le gère automatiquement.

De ce fait, lorsque le code des constructeurs des classes filles est retranscrit en C, il faut également gérer cet appel. Il est réalisé grâce à la ligne suivante qui fait intervenir le pointeur sur la méta-structure d'`ObjetGraphique` :

`metaObjetGraphique->constructeurObjetGraphique(&this`[1]`->superClasse)`.

La question de savoir quelle méta-structure utiliser pour appeler le bon constructeur peut facilement être résolu par le générateur en regardant de quelle classe hérite les structures dont il génère le code. Ici, il s'agit du code d'une structure censée hériter d'`ObjetGraphique` donc la méta-structure correspondante est `metaObjetGraphique`. Ce pointeur étant globale, il est accessible depuis n'importe où dans le code ce qui rend l'appel précédant valide. Enfin, la dernière chose qu'il ne faut pas oublier dans le code du constructeur des structures filles est l'initialisation du pointeur `myClass`. Celui-ci a pour rôle de référencer les méta-structures qui leur sont associées. En pratique ces méta-structures sont encore une fois des variables globales, et sont initialisées en début de programme.

[1] `this` est le nom donné au paramètre de type `Rectangle *` ou `Cercle *` passé aux méthodes `constrcteurRectangle` ou `constructeurCercle`.

3 Le polymorphisme

La dernière caractéristique que proposent les langages objet est le polymorphisme. Il s'agit d'un concept très puissant dont le principe est expliqué à la suite de ce rapport.

3.1 Principe

Le polymorphisme dérive directement du principe d'héritage. Pour rappel, ce dernier permet à un objet d'hériter des comportements de ses ancêtres. Avec le polymorphisme, il est alors possible de surcharger, voire de redéfinir, ces méthodes héritées afin de les rendre plus spécifiques aux cas d'utilisation d'un objet. Ainsi, un objet héritier pourra contenir parmi les traitements qu'il expose, une méthode avec le même nom qu'une méthode héritée. Le modèle implique alors que la bonne méthode soit appelée en fonction du contexte. Si une méthode est redéfinie dans une classe fille, celle-ci prend le pas sur la méthode de la classe mère.

Certains de ces appels peuvent être résolus pendant la compilation. Par exemple si un appel à une méthode redéfinie se fait depuis un objet de type fille, le compilateur sait automatiquement quelle méthode choisir pour générer le code correspondant. La liaison du code se fait donc au moment de la compilation et porte le nom de liaison précoce. Ce type de liaison peut être gérée par l'analyseur sémantique qui sera ensuite en mesure de dicter au générateur de code quelle fonction appelée. Cependant, le polymorphisme implique également que les appels de méthodes redéfinies puissent être résolus lors du *runtime*.

La liaison du code est alors faite au moment de l'exécution et porte le nom de liaison tardive. C'est précisément ce comportement que le générateur doit prendre en compte au moment de transformer une classe C++ en code C. Ainsi, en C++, lorsqu'une méthode d'une classe mère est destinée à être redéfinie dans une classe fille, il faut penser à rajouter le mot `virtual` dans son prototype afin que ce comportement soit pris en charge par le compilateur. Grâce à cela, il est possible qu'un appel à une méthode virtuelle, faite depuis un pointeur de la classe mère qui référence une instance de fille, soit résolu à l'exécution. La méthode appelée est alors celle de la fille.

La partie suivante explique quelles sont les stratégies à mettre en œuvre pour reproduire ce comportement lorsque les classes C++ sont retranscrites en C.

3.2 Solution

Tout d'abord le fait de pouvoir résoudre des appels vers les méthodes d'une classe fille depuis un pointeur de type mère, implique que cette dernière soit en mesure, d'une manière ou d'une autre, de connaitre son type réel. C'est pourquoi la première chose qu'il est nécessaire de faire pour gérer le polymorphisme est de rajouter un attribut `type` dans les structures correspondant aux classes mères.

Dès lors, il faut réellement trouver une stratégie pour que le code généré soit capable de résoudre dynamiquement ce genre d'appels. La première solution qui vient

instinctivement est de rajouter des pointeurs de fonctions dans les structures des classes mères pour chaque méthode virtuelle. Ainsi, au moment de la construction, s'il s'agit d'une mère imbriquée dans une fille, le générateur sera capable d'initialiser le pointeur vers la méthode fille et les appels pourront être résolus depuis un pointeur de type mère. Cependant cette conception remet en cause toute la stratégie précédente qui consiste à séparer les méthodes des données au travers de méta-structures.

L'autre technique envisageable consiste à utiliser des tableaux de pointeurs de fonctions. Ceux-ci sont alors stockés dans la méta-structure de la classe mère et sont indexés sur le nombre de classes présentes dans la hiérarchie générée. Un tableau est alors nécessaire par méthode virtuelle et chaque case correspond à une implémentation de la méthode. L'indice de la case permet de déterminer de quelle implémentation il s'agit. Par exemple, à l'indice 0 seront toujours stockées les versions mères des méthodes virtuelles traitées et à l'indice 1 les méthodes redéfinies dans une classe fille. Il est alors nécessaire d'initialiser correctement ces tableaux avant de pouvoir les utiliser.

Ensuite, il faut trouver une stratégie pour utiliser ces tableaux correctement et gérer les liaisons tardives. Pour ce faire, les prototypes des méthodes virtuelles vont être ajoutés comme pointeurs de fonctions dans les méta-structures des classes mères. Ces pointeurs vont alors faire référence à des méthodes présentes dans le fichier .c des méta-structures mères. Ces routines auront alors pour rôle de rediriger l'appel d'une méthode virtuelle vers la bonne implémentation en fonction du type d'appelant. Pour cela, elles n'auront qu'à utiliser le tableau de pointeurs de fonctions correspondant à la méthode virtuelle et grâce au `type` ajouté dans la structure de la mère, elles pourront sélectionner la bonne case du tableau.

La partie suivante met en application cette solution pour la gestion du polymorphisme au travers d'un exemple.

3.3 Exemple

L'exemple reprend la hiérarchie de classes précédente avec l'`ObjetGraphique`, le `Rectangle` et le `Cercle` et y rajoute des méthodes virtuelles dont certaines sont redéfinies dans les classes filles et d'autres pas (Figure 6). Les cinq méthodes virtuelles sont `GetCentreX`, `GetCentreY`, `effacer`, `deplacer` et `afficher`.

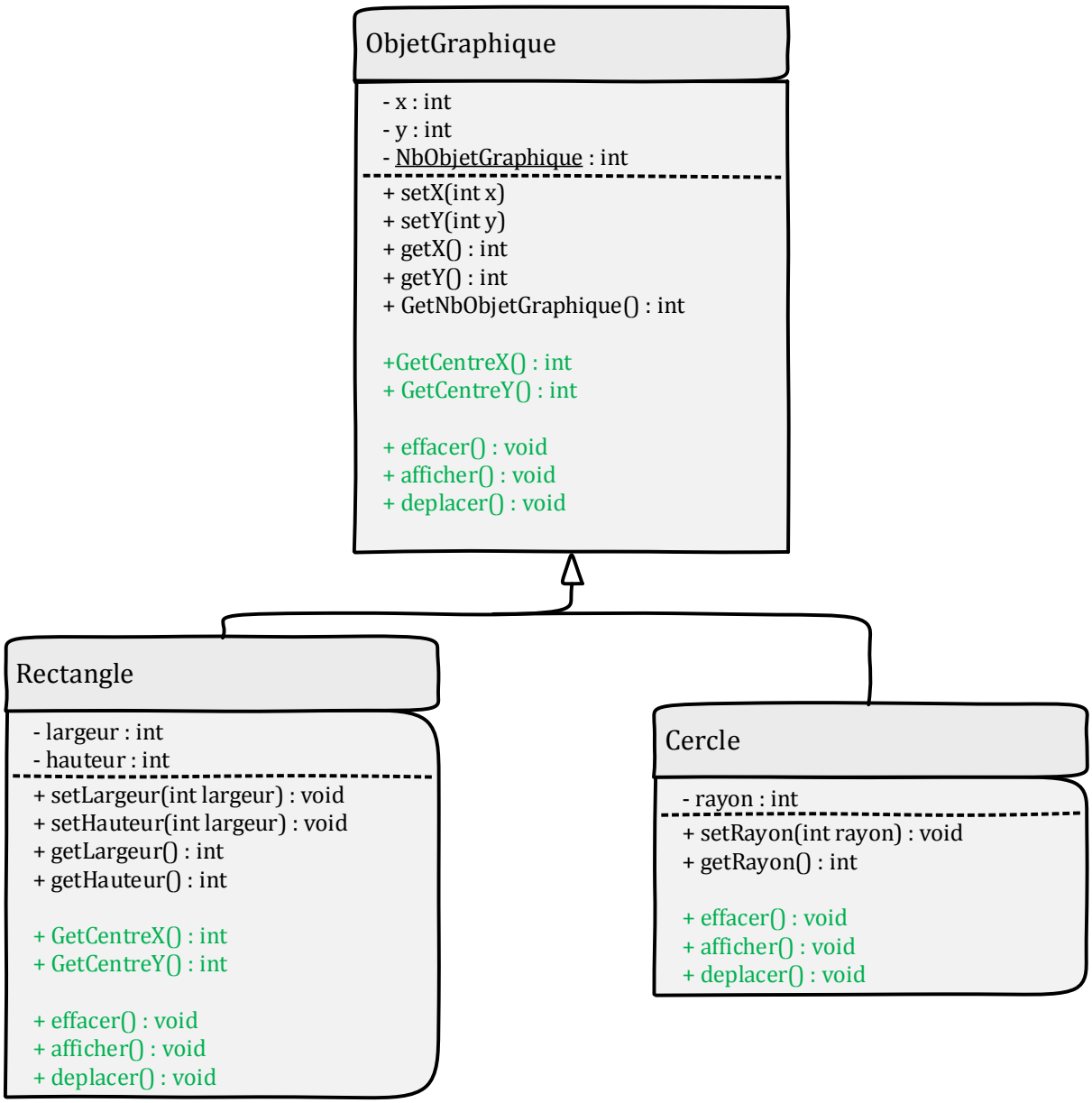

Figure 6 - Redéfinition de méthodes dans les classes filles

La génération du code C correspondant n'implique pas de modifications dans les classes qui héritent d'`ObjetGraphique` mais uniquement dans le code de cette dernière. Ces modifications sont écrites en vert sur le diagramme suivant (Figure 7).

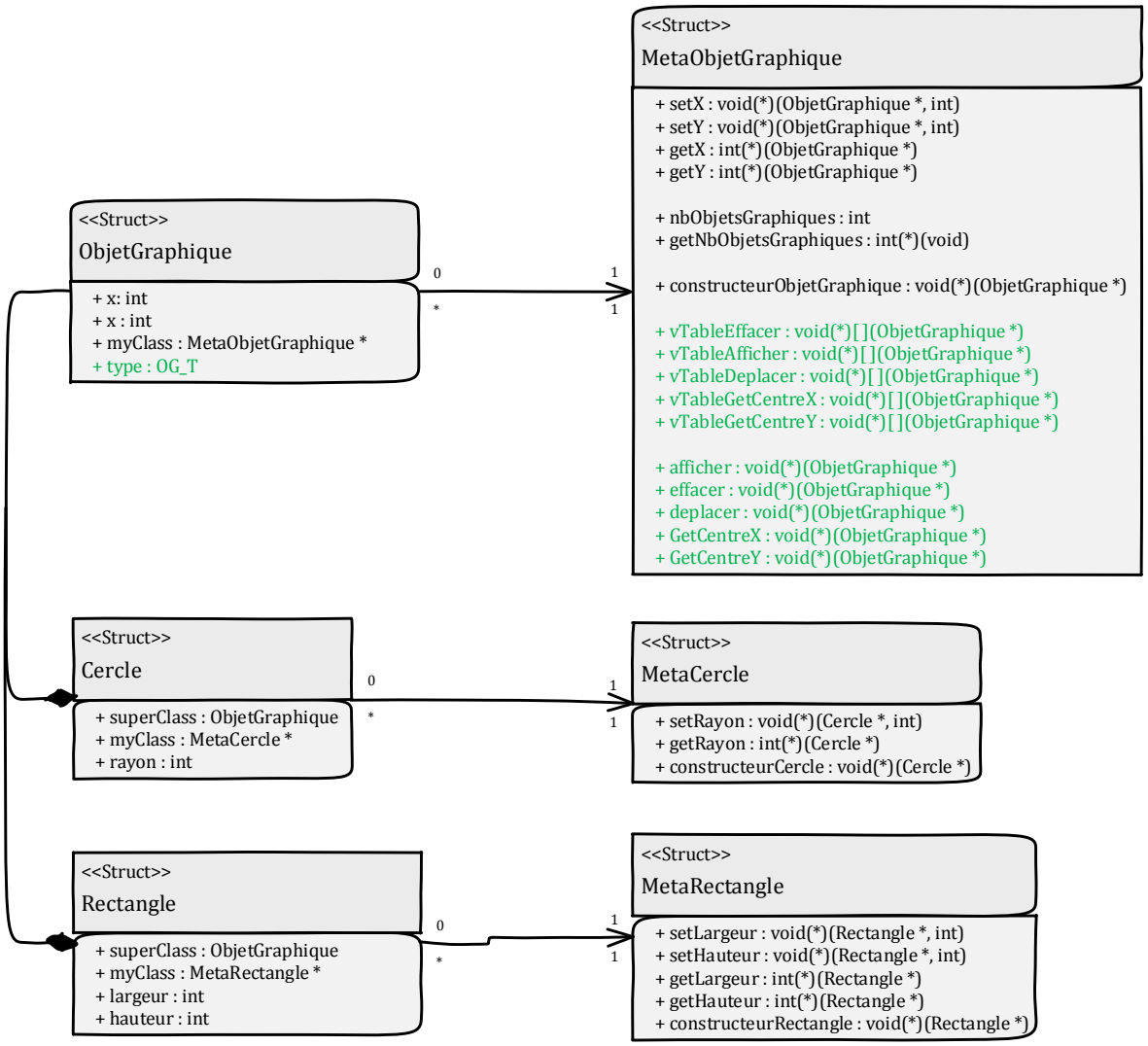

Figure 7 - Gestion du polymorphisme en C

La méta-structure d'objet graphique se voit donc rajouter ses tableaux de pointeurs de fonctions pour la gestion des méthodes virtuelles. Ce sont les attributs qui commencent par `vTable` suivis du nom de la méthode virtuelle. La structure correspondant à la classe `ObjetGraphique` se voit quant à elle rajouter une donnée `type` de type `OG_T`. Cette dernière correspond à une énumération contenant un membre par classe dérivée. Ici `OG_T` contient donc `RECTANGLE`, `CERCLE` et `OG`. Ces constantes permettent de faire en sorte que les cases des `vTable` soient toujours initialisées avec de la même manière, c'est-à-dire, avec les mêmes indices pour les méthodes du `Cercle`, `Rectangle` ou `ObjetGraphique`. Par la suite un appel C++ du type :

```
Rectangle rectangle ;

ObjetGraphique * pObjetGraphique = (ObjetGraphique *)&rectangle ;

pObjetGraphique->afficher() ;
```

Devient en C :

```
Rectangle rectangle ;

ObjetGraphique * pObj = (ObjetGraphique *)&rectangle;

metaObjetGraphique->afficher(pObj) ;
```

C'est le pointeur de fonction présent dans la méta-structure de l'objet graphique qui permet de résoudre l'appel. Ici, l'exemple porte sur le pointeur `afficher` qui fait référence à une méthode contenue dans le fichier .c de la méta-structure. Cette dernière est très simple :

```
void afficher(ObjetGraphique_t * this)

{

    metaObjetGraphique->vTableAfficher[this->type](this);

}
```

En effet, elle utilise le tableau de pointeurs de fonctions contenu dans la méta-structure de l'objet graphique pour rediriger l'appel vers la bonne méthode. Pour ce faire, elle utilise la donnée `type` stockée dans la structure qui indique la bonne case du tableau à utiliser. En l'occurrence cette case pointe vers une méthode `afficherRectangle` présente dans le fichier .c de la méta-structure de `Rectangle` et qui effectue un travail très simple :

```
void afficherRectangle(ObjetGraphique_t * this)

{

    Rectangle_t * rectangle = (Rectangle_t *)(this);

     printf("Je suis un rectangle de hauteur %d et de largeur
     %d.\n", rectangle->hauteur, rectangle->largeur);

}
```

Encore une fois l'étape d'allocation du `metaObjetGraphique` est très importante puisque c'est elle qui doit initialiser les cases des `vTables` correctement. Cette tâche doit être effectuée en début de programme et en pratique dans ce TP, elle s'effectue dans une méthode `allouerMetaObjetGraphique` appelée en tout début de programme.

Conclusion

Ce projet nous a permis d'apprendre de nombreuses choses, non seulement sur l'IDM, mais aussi sur les langages objets et en particulier sur le C++. En effet, réfléchir à la manière dont sont implémentés les mécanismes que propose ce langage nous a, en quelque sorte, obligé à mieux le comprendre. Certains aspects, comme la gestion des appels polymorphiques, assez obscures lorsque l'on apprend le C++, ont pu être éclaircis et mieux compris grâce à la réalisation de ce TP.

Enfin, ce dernier nous a également permis de constater la puissance de l'IDM. Cette discipline donne un cadre pour la réalisation de tâches complexes comme le passage d'une modélisation d'un problème à une autre. C'est le cas dans ce TP, où l'objectif était de passer d'une modélisation objet d'un problème en C++ à son équivalent en C. Les notions de méta-modélisation des concepts objets, vues en cours, se sont alors révélées très utiles.

Lien utile

Lien vers le Github du projet : https://github.com/LebonNic/tp_cimo.