

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 - Schéma d'une chaîne de compilation classique

L’analyse lexicale est un processus qui permet de découper le code source en unités atomiques appelées lexème, 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’éventuels 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.

# 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.

## 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.

## 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 header définit précédemment. 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 dans le fichier .h.

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 recopiés. 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 joue alors le rôle de méta-structure par rapport à la première. La structure contenant les attributs n’aura alors plus qu’à contenir 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.

## 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 - La classe ObjetGraphique en C++

Avec la solution décrite précédemment, la classe ObjetGraphique se verrait traduite en C grâce à deux structures.



Figure - La classe ObjetGraphique en C

Une première structure contenant les attributs d’instances de la classe, avec les coordonnées x et y, et une seconde structure contenant les méthodes de 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. Effectivement, c’est aussi un point important de la POO. Les méthodes de constructions 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.

# L’héritage

## Principe

## Solution

## Exemple

# Le polymorphisme

Conclusion

Références webographiques