Livret - 3

Modularité et bases de
la programmation orientée objet

Critères et concepts de programmation modulaire et objet.

RM di scala

Cours informatique programmation

Rm di Scala - http://www.discala.net

SOMMAIRE

3.1 Modularité

Plan du chapitre:

1. Modularité (selon B.Meyer)

1.1 Notion de module

Le mot **MODULE** est un des mots les plus employés en programmation moderne. Nous allons expliquer ici ce que l'on demande, à une méthode de construction modulaire de logiciels, de posséder comme propriétés, puis nous verrons plus loin comment dans cette notion de module se trouve implantée avec C# par la notion de classe.

B.Meyer est l'un des principaux auteurs avec G.Booch qui ont le plus travaillé sur cette notion. Le premier a implanté ses idées dans le langage orienté objet " Eiffel ", le second a utilisé la modularité du langage Ada pour introduire le concept d'objet qui a été la base de méthodes de conception orientées objet : OOD, HOOD,UML...

Nous nous appuyons ici sur les concepts énoncés par B.Meyer fondés sur 5 critères et 6 principes relativement à une méthodologie d'analyse de type modulaire. Une démarche (et donc le logiciel construit qui en découle) est dite modulaire si elle respecte au moins les concepts ci-après.

1.2 Critères principaux de modularité

Les 5 principes retenus :

- **La décomposabilité modulaire**
- **La composition modulaire**
- **La continuité modulaire**
- **La compréhension modulaire**
- **La protection modulaire**

La décomposabilité modulaire :

capacité de décomposer un problème en sous-problèmes, semblable à la méthode structurée descendante.

La composition modulaire :

capacité de recombinaison et de réagencement de modules écrits, semblable à la partie ascendante de la programmation structurée.

La continuité modulaire :

capacité à réduire l'impact de changements dans les spécifications à un minimum de modules liés entre eux, et mieux à un seul module.

La compréhension modulaire :

capacité à l'interprétation par un programmeur du fonctionnement d'un module ou d'un ensemble de modules liés, sans avoir à connaître tout le logiciel.

La protection modulaire :

capacité à limiter les effets produits par des incidents lors de l'exécution à un nombre minimal de modules liés entre eux, mieux à un seul module.

Attention *Les pointeurs en mode **unsafe** en C#*

Le type pointeur met fortement en défaut ce critère, car sa gestion mémoire est de bas niveau et donc confiée au programmeur ; les pointeurs ne respectent même pas la notion de variable locale!
En général le passage par adresse met en défaut le principe de protection modulaire.

1.3 Préceptes minimaux de construction modulaire

Etant débutants, nous utiliserons quatre des six préceptes énoncés par B.Meyer. Ils sont essentiels et sont adoptés par tous ceux qui pratiquent des méthodes de programmation modulaire :

- **Interface de données minimale**
- **Couplage minimal**
- **Interfaces explicites**
- **Information publique et privée**

Précepte 1 : Interface de données minimale

Un module fixé doit ne communiquer qu'avec un nombre " minimum " d'autres modules du logiciel. L'objectif est de minimiser le ***nombre*** d'interconnexions entre les modules. Le graphe établissant les liaisons entre les modules est noté " graphe de dépendance ". Il doit être le moins maillé possible. La situation est semblable à celle que nous avons rencontré lors de la description des différentes topologies des réseaux d'ordinateurs : les liaisons les plus simples sont les liaisons en étoile, les plus complexes (donc ici déconseillées) sont les liaisons totalement maillées.

L'intérêt de ce précepte est de garantir un meilleur respect des critères de continuité et de protection modulaire. Les effets d'une modification du code source ou d'une erreur durant l'exécution dans un module peuvent se propager à un nombre plus ou moins important de modules en suivant le graphe de liaison. Un débutant optera pour une architecture de liaison simple, ce qui induira une construction contraignante du logiciel. L'optimum est défini par le programmeur avec l'habitude de la programmation.

Précepte 2 : Couplage minimal

Lorsque deux modules communiquent entre eux, l'échange d'information doit être minimal. Ce précepte ne fait pas double emploi avec le précédent. Il s'agit de minimiser la *taille* des interconnexions entre modules et non leur *nombre* comme dans le précepte précédent.

Précepte 3 : Interfaces explicites

Lorsque deux modules M1 et M2 communiquent, l'échange d'information doit être lisible explicitement dans l'un des deux ou dans les deux modules.

Précepte 4 : Information publique et privée

Toute information dans un module doit être répartie en deux catégories : l'information privée et l'information publique.

Ce précepte permet de modifier la partie privée sans que les clients (*modules utilisant ce module*) aient à supporter un quelconque problème à cause de modifications ou de changements. Plus la partie publique est petite, plus on a de chances que des changements n'aient que peu d'effet sur les clients du module.

- La partie publique doit être la description des opérations ou du fonctionnement du module.
- La partie privée contient l'implantation des opérateurs et tout ce qui s'y rattache.

Enfin et pour mémoire nous citerons l'existence du précepte d'ouverture-fermeture et du précepte d'unités linguistiques.

3.2 Introduction à la programmation orientée objet

Plan du chapitre:

Introduction

Le lecteur qui connaît les fondements de ce chapitre peut l'ignorer et passer au chapitre suivant. Dans le cas contraire, **la programmation visuelle** étant intimement liée aux outils et aux concepts **objets**, ce chapitre est un minimum incontournable.

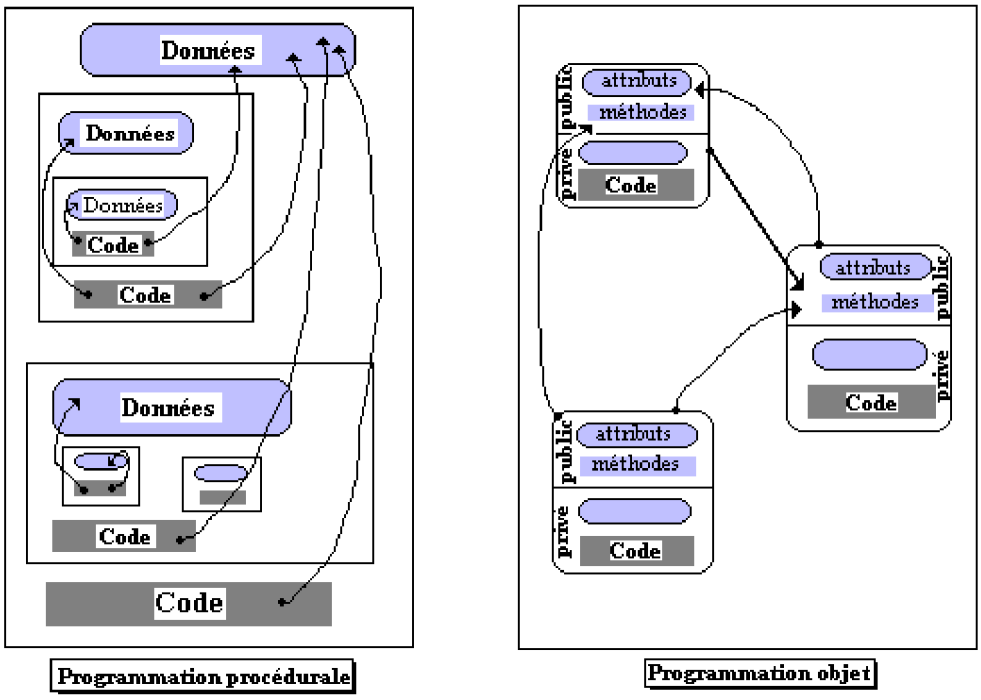
La programmation classique ou *procédurale* telle que le débutant peut la connaître à travers des langages de programmation comme Pascal, C etc... traite les programmes comme un ensemble de données sur lesquelles agissent des procédures. Les procédures sont les éléments actifs et importants, les données devenant des éléments passifs qui traversent *l'arborescence de programmation* procédurale en tant que flot d'information.

Cette manière de concevoir les programmes reste proche des machines de Von Neuman et consiste en dernier ressort à traiter indépendamment les données et les algorithmes (traduits par des procédures) sans tenir compte des relations qui les lient.

En introduisant la notion de modularité dans la programmation structurée descendante, l'approche diffère légèrement de l'approche habituelle de la programmation algorithmique classique. Nous avons défini des *machines abstraites* qui ont une autonomie relative et qui possèdent leurs propres structures de données; la conception d'un programme relevait dès lors essentiellement de la description des interactions que ces machines ont entre elles.

La programmation orientée objet relève d'une conception ascendante définie comme des "**messages**" échangés par des entité de base appelées objets.

comparaison des deux topologies de programmation

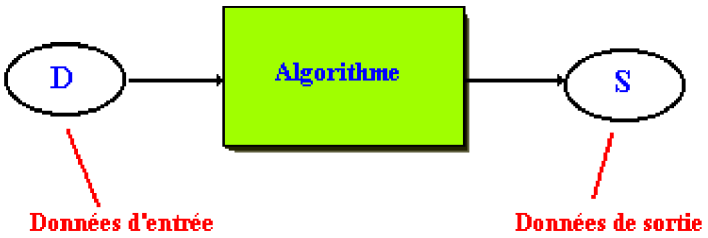


Les langages objets sont fondés sur la connaissance d'une seule catégorie d'entité informatique : l'objet. Dans un objet, traditionnellement ce sont les données qui deviennent

prépondérantes. On se pose d'abord la question : "de quoi parle-t-on ?" et non pas la question "que veut-on faire ?", comme en programmation algorithmique. C'est en ce sens que les machines abstraites de la programmation structurée modulaire peuvent être considérées comme des pré-objets.

En fait la notion de TAD est utilisée dans cet ouvrage comme spécification d'un objet, en ce sens nous nous préoccupons essentiellement des services offerts par un objet indépendamment de sa structure interne.

Programmation structurée :



1. Concepts fondamentaux de La P.O.O

Nous écrirons P.O.O pour : programmation orientée objet.

Voici trois concepts qui contribuent à la puissance de la P.O.O.

- Concept de **modélisation** à travers la notion de classe et **d'instanciation** de ces classes.
- Concept **d'action** à travers la notion d'envoi de messages et de méthodes à l'intérieur des objets.
- Concept de construction par **réutilisation et amélioration** par l'utilisation de la notion d'héritage.

1.1 les objets

Objet

Un **module** représente ***un objet ou une classe d'objet*** de l'espace du problème et non une étape principale du processus total, comme en programmation descendante.

Recenser les objets du monde réel

Lors de l'analyse du problème, on doit faire l'état de l'existant en recensant les objets du monde réel. On établit des classes d'objets et pour chaque objet on inventorie les connaissances que l'on a sur lui :

- Les connaissances déclaratives,
- les connaissances fonctionnelles,
- l'objet réel et les connaissances que l'on a sur lui sont regroupés dans une même entité.

On décrit alors les systèmes en classes d'objets plutôt qu'en terme de fonctions.

Par Exemple : **Une application de gestion bancaire est organisée sur les objets comptes, écritures, états.**

Les objets rassemblent une partie de la connaissance totale portant sur le problème. Cette connaissance est répartie sur tous les objets sous forme déclarative ou procédurale.

Les objets sont décrits selon le modèle des structures abstraites de données (TAD) : ils constituent des boîtes noires dissimulant leur implantation avec une interface publique pour les autres objets. Les interactions s'établissant à travers cette interface.

Vocabulaire objet

Encapsulation

c'est le fait de réunir à l'intérieur d'une même entité (*objet*) le code (*méthodes*) + données (*champs*).

Il est donc possible de masquer les informations d'un objet aux autres objets.

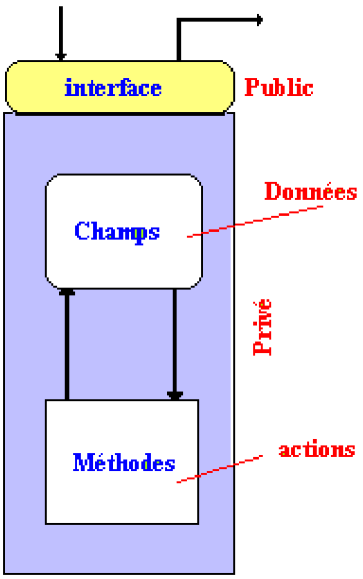
Deux niveaux d'encapsulation sont définis :

Privé

les **champs** et les **méthodes masqués** sont dans la partie privée de l'objet.

Public

les **champs** et les **méthodes visibles** sont dans la partie interface de l'objet.



Les notions de **privé** et de **public** comme dans un objet n'ont trait qu'à la communication entre deux objets, à l'intérieur d'un objet elles n'ont pas cours.

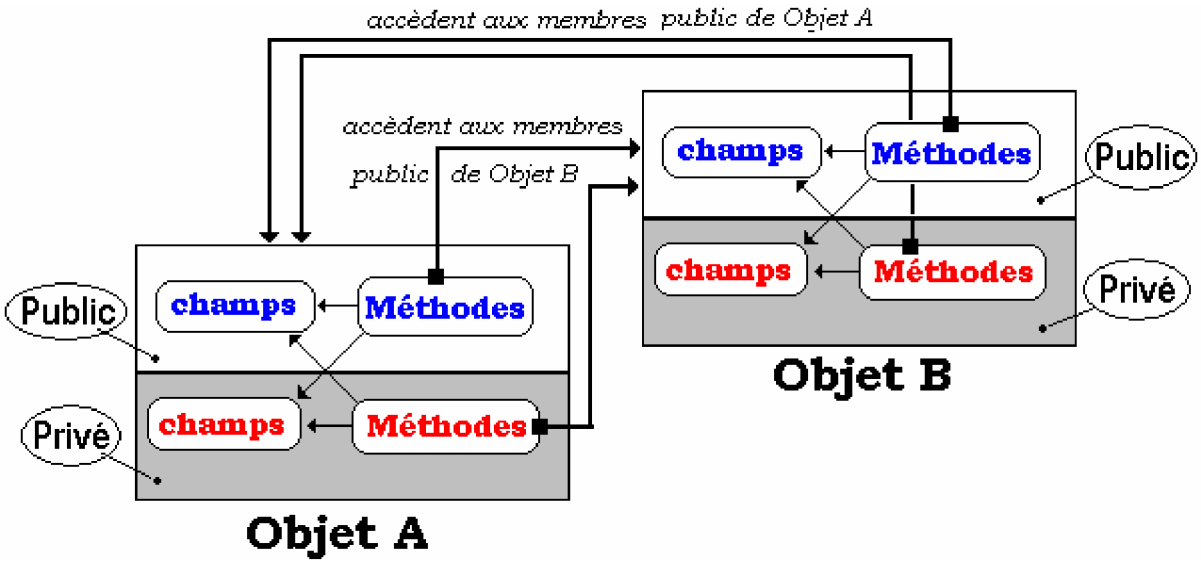


Figure sur la visibilité entre deux objets
Les méthodes de public ou privé de l'objet A accèdent et peuvent utiliser les méthodes et les champs public de B.
Les méthodes de public ou privé de l'objet B accèdent et peuvent utiliser les méthodes et les champs public de A.

1.2 les classes

Postulons une analogie entre les objets matériels de la vie courante et les objets informatiques. Un objet de tous les jours est souvent obtenu à partir d'un moule industriel servant de modèle pour en fabriquer des milliers. Il en est de même pour les objets informatiques.

Classe

Une **classe** est une sorte de ***moule ou de matrice*** à partir duquel sont engendrés les objets réels qui s'appellent des ***instances*** de la classe considérée.

Une classe contient

Des attributs (ou champs, ou variables d'instances).
Les attributs de la classe décrivent la structure de ses instances (les objets).

Des méthodes (ou opérations de la classe).
Les méthodes décrivent les opérations qui sont applicables aux instances de la classe.

Membres

les **attributs** et les **méthodes** d'une classe sont des **membres** de la classe.

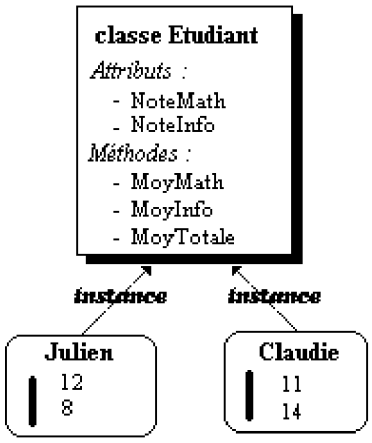
Instance

Un objet de classe A est appelé aussi une **instance** de classe A, l'opération de construction d'un objet s'appelle alors **l'instanciation**.

Remarque

En POO, programmer revient donc à décrire des classes d'objets, à caractériser leur structure et leur comportement, puis à instancier ces classes pour créer des objets réels. Un objet réel est matérialisé dans l'ordinateur par une zone de mémoire que les données et son code occupent.

Un exemple : des étudiants
Supposons que chaque étudiant soit caractérisé par sa note en mathématiques (NoteMath) et sa note en informatique (NoteInfo). Un étudiant doit pouvoir effectuer éventuellement des opérations de calcul de ses moyennes dans ces deux matières (MoyMath, MoyInfo)et connaître sa moyenne générale calculée à partir de ces deux notes (MoyTotale).



La classe Etudiant a été créée. Elle ne possède que les **attributs** *NoteMath* et *NoteInfo*. Les **méthodes** de cette classe sont par exemple *MoyMath*, *MoyInfo*, *MoyTotale*.

Nous avons créé deux **objets** étudiants de la classe Etudiant (deux **instances** : Julien et Claudie).

1.3 L'héritage

Dans un **LOO** (**L**angage **O**rienté **O**bjet), il existe une particularité dans la façon d'organiser ses classes : l'héritage de propriétés. L'objectif est de construire de nouvelles classes en **réutilisant** des attributs et des méthodes de classes déjà existantes.

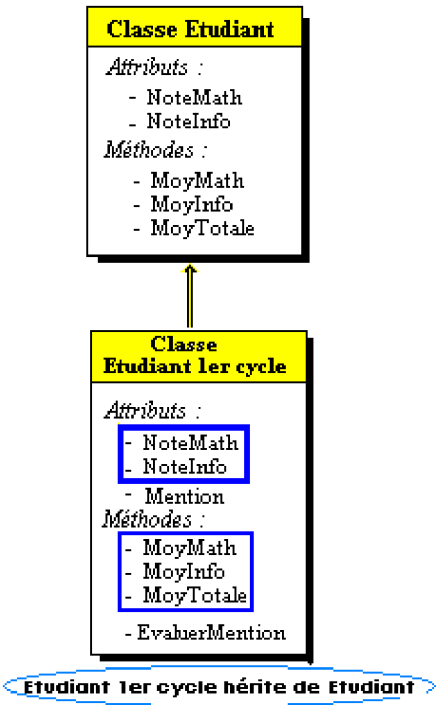
Héritage

C'est un mécanisme très puissant qui permet de décrire des structures génériques en transmettant depuis l'intérieur d'une même classe toutes les propriétés communes à toutes les " sous-classes " de cette classe. Par construction toutes les sous-classes d'une même classe possèdent toutes les attributs et les méthodes de leur classe parent.

Propriétés de l'héritage

- Les attributs et les méthodes peuvent être modifiés au niveau de la sous-classe qui hérite.
- Il peut y avoir des attributs et/ou des méthodes supplémentaires dans une sous-classe.
- Une classe A qui hérite d'une classe B dispose implicitement de tous les attributs et de toutes les méthodes définis dans la classe B.
- Les attributs et les méthodes définis dans A sont prioritaires par rapport aux attributs et aux méthodes de même nom définis dans

Exemple :
La classe " *Etudiant premier cycle*" héritant de la classe *Etudiant* précédemment définie.



Tout se passe comme si toute la classe *Etudiant* était recopiée dans la sous-classe *Etudiant premier cycle* (même si l'implémentation n'est pas faite ainsi).

La nouvelle classe dispose d'un attribut supplémentaire (Mention) et d'une méthode supplémentaire (EvaluerMention).

```
type Tmention=(Passable, Abien, Bien, Tbien);
Etudiant = class
    NoteMath : real;
    NoteInfo : real;
    procedure MoyMath(UneNote:real);
    procedure MoyInfo(UneNote:real);
    function MoyTotale : real ;
end;
 Etudiant1erCycle = class(Etudiant)
    Mention: Tmention ;
    function EvaluerMention: Tmention;
end;
```

Ceci est une implantation possible de la signature de la classe en :

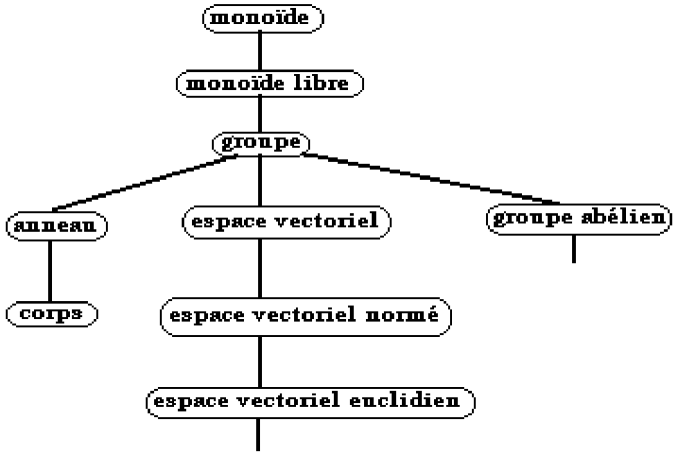
Delphi

```
class Etudiant {
   public float NoteMath;
   public float NoteInfo;
   public void MoyMath(float UneNote){
   ...    }
   public void MoyInfo(float UneNote){
   ...    }
   public float MoyTotale(){
   ...    }
 }// fin classe Etudiant
public class Etudiant1erCycle extends Etudiant{
   public String Mention;
   public String EvaluerMention( ){
   ...    }
   public static void Main(String[ ] args){
   ...    }
}// fin
```

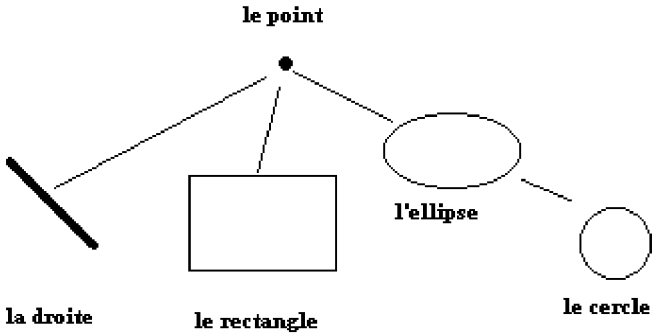
Ceci est une implantation possible de la signature de la classe en :

Exemples d'héritage dans d'autres domaines :

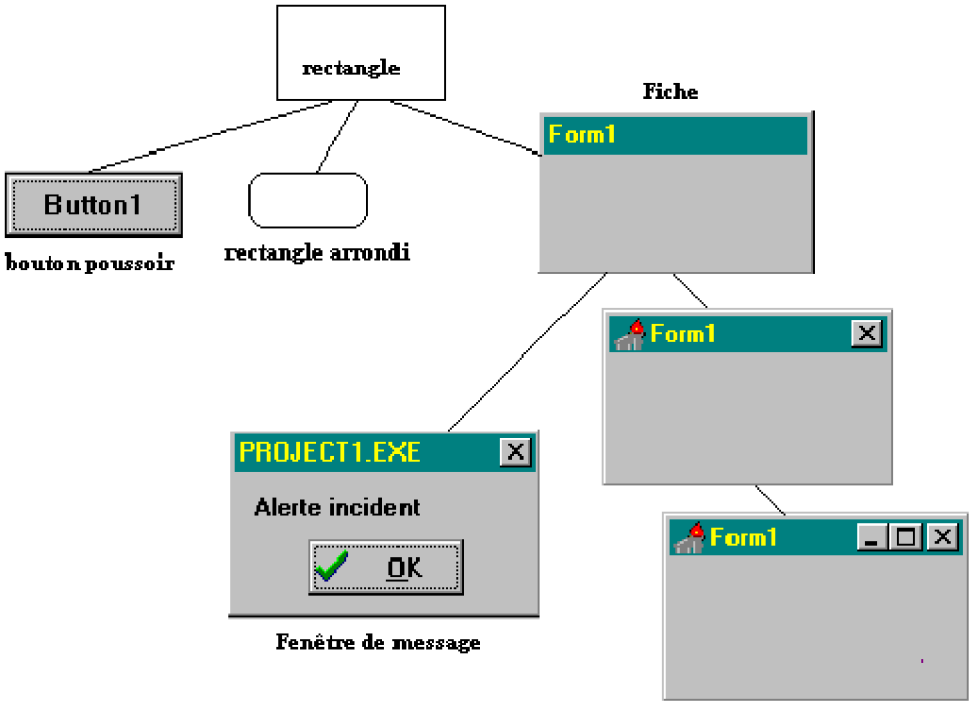
Héritage dans les structures mathématiques :



Héritage de figures de base en géométrie affine :



Héritage d'objets graphiques dans un système multi-fenêtré fictif :



2. Introduction à la conception orientée objet

L'attitude est ici résolument sous-tendue par un double souci : fournir des outils méthodologiques rationalisant l'effort de production du logiciel, sans que leur lourdeur rebute l'étudiant non professionnel et masque ainsi l'intérêt de leur utilisation. L'expérience d'enseignement de l'auteur avec des débutants a montré que si les étudiants sont appelés à développer sans outils méthodiques, ils pratiquent ce qu'appelle J.Arsac " la grande bidouille ". Mais dans le cas contraire, l'apprentissage détaillé de trop de méthodes strictes bien qu'efficaces (OOD, OMT, HOOD, UML,...)finit par ennuyer l'étudiant ou du moins par endormir son sens de l'intérêt. Dans ce dernier cas l'on retrouve " la grande bidouille " comme étape finale. Le chemin est donc étroit et il appartient à chaque enseignant de doser en fonction de l'auditoire l'utile et le superflu.

Nous utilisons ici un de ces dosages pour montrer à l'étudiant comment écrire des programmes avec des objets sans être un grand spécialiste. Une aide irremplaçable à cet égard nous sera fournie par l'environnement de développement visuel Delphi.

2.1 La méthode de conception OOD simplifiée

La méthode O.O.D (object oriented design) de G.Booch propose 5 *étapes* dans l'établissement d'une conception orientée objet. Ces étapes n'ont pas obligatoirement à être enchaînées dans l'ordre dans lequel nous les citons dans le paragraphe suivant. C'est cette souplesse qui nous a fait choisir la démarche de G.Booch, car cette méthode est fondamentalement incrémentale et n'impose **pas un cadre trop précis et trop rigide** dans son application. Cette démarche se révèle être utile pour un débutant et lui permettra de fabriquer en particulier des prototypes avec efficacité sans trop surcharger sa mémoire.

Identifier les objets et leurs attributs

On cherchera à identifier les objets du monde réel que l'on voudra réaliser.

Conseil : *Il faut identifier les propriétés caractéristiques de l'objet (par l'expérience, l'intuition,...).On pourra s'aider d'une description textuelle (en langage naturel) du problème. La lecture de cette prose aidera à déduire les bons candidats pour les noms à utiliser dans cette description ainsi que les propriétés des adjectifs et des autres qualifiants.*

Identifier les opération

On cherchera ensuite à identifier les actions que l'objet subit de la part de son environnement et qu'il provoque sur son environnement.

Conseil : *Les verbes utilisés dans la description informelle (textuelle) fournissent de bons indices pour l'identification des opérations. Si nécessaire, c'est à cette étape que l'on pourra définir les conditions d'ordonnancement temporel des opérations (les événements ayant lieu).*

Etablir la visibilité

L'objet étant maintenant identifié par ses caractéristiques et ses opérations, on définira ses relations avec les autres objets.

Conseil : *On établira quels objets le voient et quels objets sont vus par lui (les spécialistes disent alors qu'on insère l'objet dans la topologie du projet). Il faudra prendre bien soin de définir ce qui est visible ou non, quitte à y revenir plus tard si un choix s'est révélé ne pas être judicieux.*

Etablir l'interface

Dès que la visibilité est acquise, on définit l'interface précise de l'objet avec le monde extérieur.

Conseil : *Cette interface définit exactement quelles fonctionnalités sont accessibles et sous quelles formes. Cette étape doit pouvoir être décrite sous notation formelle; les TAD sont l'outil que nous utiliserons à cette étape de conception.*

Implémenter les objets

La dernière étape consiste à implanter les objets en écrivant le code.

Conseil : *Cette étape peut donner lieu à la création de nouvelles classes correspondant par exemple à des nécessités d'implantation. Le code en général correspond aux spécifications concrètes effectuées avec les TAD, ou à la traduction des algorithmes développés par la méthode structurée. Lors de cette étape, on identifiera éventuellement de nouveaux objets de plus bas niveau d'abstraction qui ne pouvaient pas être analysés en première lecture(ceci provoquant l'itération de la méthode à ces niveaux plus bas).*

Nous n'opposons pas cette méthode de conception à la méthode structurée modulaire. Nous la considérons plutôt comme complémentaire (en appliquant à des débutants une idée contenue dans la méthode HOOD). La méthode structurée modulaire sert à élaborer des algorithmes classiques comme des actions sur des données. La COO permet de définir le monde de l'environnement de façon modulaire. Nous réutiliserons les algorithmes construits dans des objets afin de montrer la complémentarité des deux visions.

2.2 Notation UML de classes et d'objets

La notion d'un langage de modélisation standard pour tout ce qui concerne les développements objets a vu le jour en 1997 il s'agit d'UML (Unified Modeling Language).

UML n'est pas une méthode, mais une notation graphique et un métamodèle.

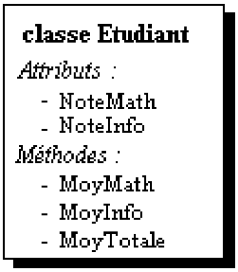
Nous allons fournir ici les principaux schémas d'UML permettant de décrire des démarches de conception objets simples. Le document de spécification de la version 1.4 d'UML par l'OMG (l'Object Management Group) représente 1400 pages de définitions sémantiques et de notations; il n'est donc pas question ici de développer l'ensemble de la notation UML (que d'ailleurs l'auteur ne possède pas lui-même).

Nous nous attacherons à détailler les diagrammes de base qui pourront être utilisés par la suite dans le reste du document.

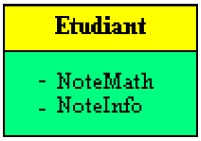
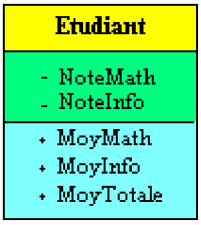
Nous nous limiterons aux notations relatives aux classes, aux objets, et à l'héritage.

SCHEMA UML DE CLASSE

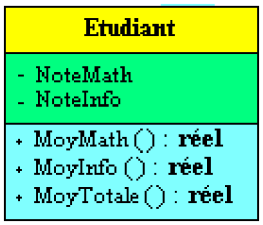
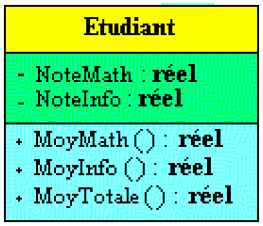
Notations UML possibles d'une classe :
Reprenons l'exemple précédent avec la classe étudiant :



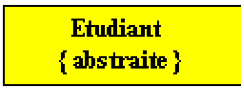
trois notations UML possibles

Simplifiée	*Avec attributs*	*Attributs et méthodes*
		

Deux autres notations UML plus complète pour la même classe

Attributs et méthodes typées	*Attributs typés et méthodes typées*
	

Une classe abstraite est notée :



VISIBILITE DES ATTRIBUTS ET DES METHODES

Notation préfixée UML pour trois niveaux de visibilité (+, - , # , $) :

Pour les attributs :

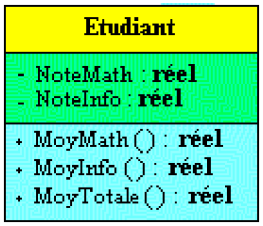
public	*privé*	*protégé*
+ Attribut1 : DeType1	- Attribut2 : DeType2	# Attribut3 : DeType3

Pour les méthodes :

public	*privé*	*protégé*
+ Methode1 ( ): DeType1	- Methode2 ( ) : DeType2	# Methode3 ( ) : DeType3

Méthode de classe
$ Methode4 ( ) : DeType4

Explicitation dans la classe Etudiant :



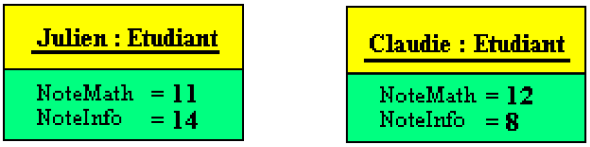
- Dans la classe **étudiant** les deux attributs **NoteMath** et **NoteInfo** sont de type réel et sont **privés** (préfixe -).
- Les trois méthodes de calcul de moyennes sont **publiques** (préfixe +).

SCHEMA UML D'UN OBJET

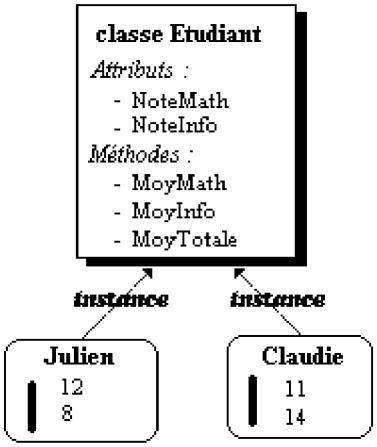
Notations UML pour deux objets étudiants instanciés à partir de la classe Etudiant :

Schéma UML simplifié :

Schéma UML avec valeur des attributs :

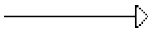


Ces notations correspondent à l'exemple ci-dessous :

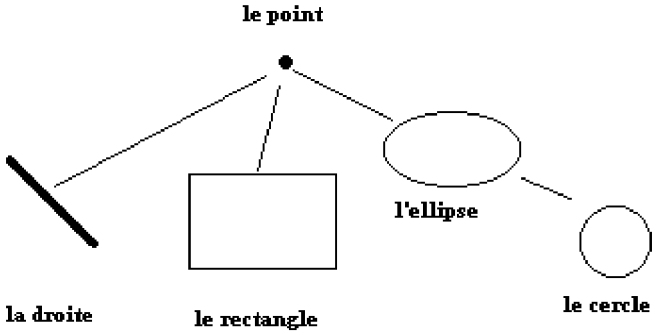


SCHEMA UML DE L'HERITAGE

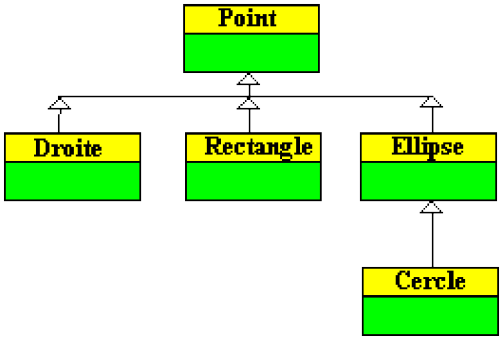
Notation UML de l'héritage :



Soit pour l'exemple de hiérarchie de classes fictives ci-dessous :

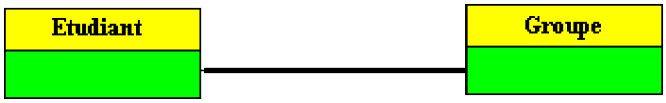


La notation UML suivante :



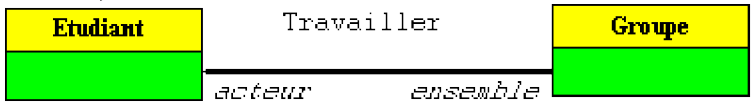
SCHEMA UML DES ASSOCIATIONS

Une association binaire (ou plus généralement n-aire), représente un lien conceptuel entre deux classes. Par exemple un étudiant travaille dans un groupe (association entre la classe **Etudiant** et la classe **Groupe**).

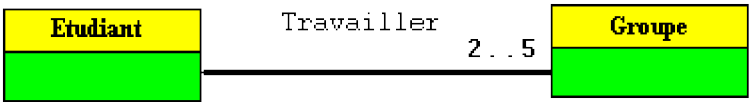


Une association peut être dénotée par une expression appelée nom d'association (nommé **Travailler** ci-dessous) :

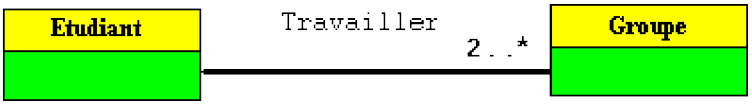
Chaque association possède donc deux extrémités appelées aussi rôles, il est possible de nommer les extrémités (nom de rôles, ci-dessous un étudiant est un acteur travaillant dans un groupe qui est un ensemble) :



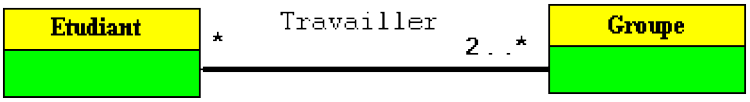
Une association peut posséder une multiplicité qui représente sous forme d'un intervalle de nombres entiers a..b, le nombre d'objets de la classe d'arrivée qui peut être mis en association avec un objet de la classe de départ. Supposons qu'un étudiant doive s'inscrire à au moins 2 groupes de travail et au plus à 5 groupes, nous aurons le schéma UML suivant :



La présence d'une étoile dans la multiplicité indiquant un nombre quelquonque (par exemple un étudiant peut s'inscrire à au moins 2 groupes sans limite supérieure):



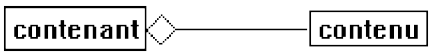
Par exemple pour dénoter en UML le fait qu'un nombre quelconque d'étudiants doit travailler dans au moins deux groupes nous écrirons:



UNE ASSOCIATION PARTICULIERE : L'AGREGATION

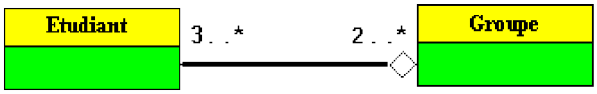
Une agrégation est une association correspondant à une relation qui lorsqu'elle est lue dans un sens signifie "est une partie de" et lorsqu'elle est lue dans l'autre sens elle signifie "est composé de". UML disposant de plusieurs raffinements possibles nous utiliserons l'agrégation comme composition par valeur en ce sens que la construction du tout implique la construction automatique de toutes les parties et que la destruction du tout entraîne en cascade la destruction de chacune de ses parties.

Notation UML de l'agrégation



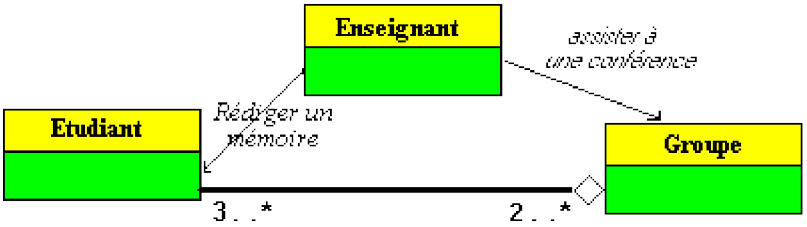
Exemple :

un groupe contient au moins 3 étudiants et chaque étudiant doit s'inscrire à au moins 2 groupes :



NOTATION UML DES MESSAGES

Un message envoyé par une classe à une autre classe est représenté par une flèche vers la classe à qui s'adresse le message, le nommage de la flèche indique le message à exécuter :



2.3 Attitudes et outils méthodologiques

Afin d'utiliser une méthodologie pratique et rationnelle, nous énumérons au lecteur les outils que nous utilisons selon les besoins, dans le processus d'écriture d'un logiciel.

En tout premier la notion de module : C'est la décomposition d'un logiciel en sous-ensembles que l'on peut changer comme des pièces d'un patchwork.
La notion de cycle de vie du logiciel : Développer un logiciel ce n'est pas seulement écrire du Pascal, de l'Ada etc...
Utiliser des TAD : Un type abstrait de données correspond très exactement à l'interface d'un module. Il renforce la méthodologie modulaire.
La programmation structurée par machines abstraites : On se sert d'une méthode de conception descendante et modulaire des algorithmes utiles pour certaines actions dans le logiciel.
La conception et la programmation orientées objet : On utilise une version simplifiée de la COO de G.Booch pour définir les classes et leurs relations en attendant une simplification pédagogique d'UML.