Chapitre 4
Concept objet en Delphi

1. Introduction à la programmation orientée objet

Il s'agit d'une approche différente de la programmation procédurale. Si un programme informatique écrit de manière procédurale est la réponse à la question "que veut-on faire?" le même programme écrit de manière orientée objet répond à la question "de quoi parle-t-on?". On découpe fonctionnellement le programme à réaliser par rôles, par concept ou par entités physiques modélisées dans le programme. Les entités ainsi définies sont appelées "objet".

Nous allons étudier dans ce chapitre l'exemple d'une application de gestion de location de véhicules. Rapidement, on peut énumérer quelques rôles et entités :

– Le véhicule (quoi ?)

– Les utilisateurs de l'application (client ou administrateur) (qui ?)

– L'agence de location (où ?)

– La réservation de location (date de début/date de fin) (quand ?)

En général, on modélise les objets à travers le langage UML qui permet de décrire schématiquement la structure, le comportement de chaque objet. On décrit aussi en UML le comportement des différents objets les uns par rapport aux autres. Au final, on peut décrire l'intégralité d'un programme informatique en UML.

Au niveau de l'implémentation en Delphi, les objets sont l'extension des records vus dans le chapitre précédent (pour rappel, le record est un type qui permet d'agréger tout un jeu d'autres types). On dit qu'ils possèdent une structure interne décrite par les champs et un comportement décrit par les méthodes que la classe implémente.

En Delphi, les objets sont décrits par la définition de la classe à laquelle il appartient.

2. Principes de la programmation objet

2.1 Les champs

Un champ est une variable interne à un objet. Comme toute variable, ce champ possède un type (entier, chaîne de caractères ou même une autre classe) et est manipulé par les méthodes de la classe à laquelle il appartient.

Un champ peut être :

– un champ d'instance, qui est propre à chaque objet.
– un champ statique ou champ de classe qui est propre à toute la classe. La valeur de cette variable est la même pour toutes les instances de la classe.

2.2 Les méthodes

Les méthodes sont les routines qui permettent de manipuler les champs de la classe. En fonction d'un certain état d'entrée A, selon la méthode exécutée M, l'objet se retrouve dans un autre état B.

L'ensemble formé par les champs et les méthodes est appelé "membres de la classe".

Une méthode peut être :

– une méthode d'instance, n'agissant que sur un seul objet (instance de la classe) à la fois,
– une méthode statique ou méthode de classe qui n'agit pas sur des variables d'instance mais uniquement sur des variables de classe. Elle est indépendante de toute instance de la classe (objet).

2.3 Encapsulation de données

L'intérêt d'un objet réside dans le fait qu'il rassemble toutes les données (champs) dont il a besoin au sein d'une même classe ainsi que les moyens de les gérer (méthodes). Dans ce principe d'encapsulation, il existe également une notion de visibilité qui permet de masquer ou non certaines données de l'objet que l'on manipule de l'extérieur.

Cela permet de garantir une cohérence des données qui ne pourront être modifiées de l'extérieur qu'à travers les méthodes visibles.

Dans le principe d'encapsulation, il existe deux modes de visibilité :

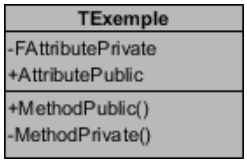
– Privé
– Public

Un membre privé est inaccessible vu de l'extérieur à la classe concernée.

2.4 Représentation UML et nommage

Voici un exemple de représentation UML (*Unified Modeling Language*) :

Ci-après est définie la classe **TExemple**. En général, on définit le nom d'une classe Delphi avec un T majuscule devant pour signifier « type ». Bien qu'un type et une classe sont deux notions différentes, il faut garder en mémoire que Delphi est une extension du langage Pascal où il n'existait que des types ; il s'agit uniquement d'une continuité de notation.



La représentation de cette classe possède deux zones : celle du dessus pour les champs et l'autre en dessous pour les méthodes. Ainsi, dans la seconde zone, après chaque nom sont ajoutés les symboles **()** spécifiant un appel de routine.

Remarque

*En Delphi, quand une routine ne possède pas de paramètre d'entrée, les symboles **()** ne sont pas obligatoires. Cependant, il est préférable de les rajouter pour une meilleure visibilité de code.*

Nous trouvons donc deux champs : **FAttributePrivate** et **AttributePublic**.

Le **–** devant **FAttributePrivate** signifie que la visibilité est privée et le **+** devant **AttributePublic** signifie que la visibilité est publique. Pour un champ privé, on met en général un F pour désigner «field» (champ en anglais).

De même, pour les méthodes, on trouve **MethodPublic()** en visibilité publique et **MethodPrivate()** en visibilité privée.

2.5 Héritage

Il s'agit du mécanisme qui permet de créer une classe B à partir d'une classe existante A. On dit qu'on a « spécialisé » ou « enrichi » la classe A par ce mécanisme.

La notion de visibilité évoquée précédemment rentre en compte également :

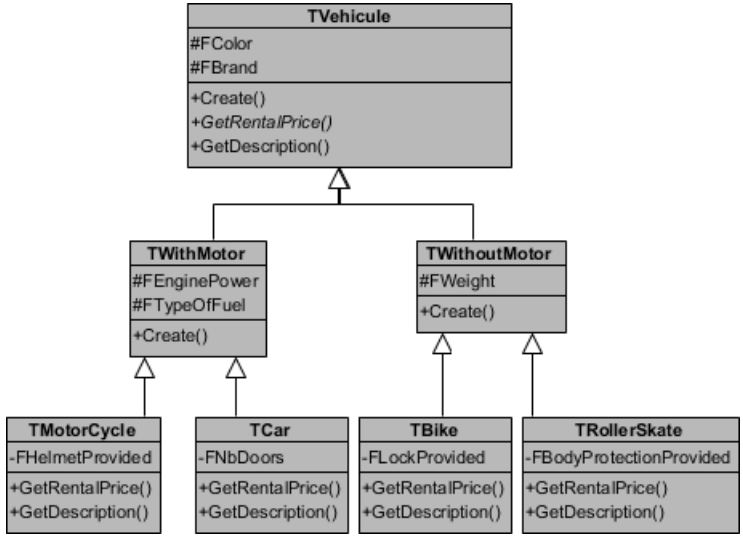
La classe B aura connaissance par le mécanisme d'héritage des membres publics ou protégés de la classe A.

Le terme « protégé » est une visibilité à mi-chemin entre public et privé, car comme un membre privé, un membre protégé ne peut pas être manipulé de l'extérieur de la classe mais il est accessible dans la classe héritée.

En UML, le symbole utilisé pour noter une visibilité protégée est le #.

Pour reprendre l'exemple de l'application de location de véhicules qui permet de gérer des vélos, des rollers des voitures et des motos, il est possible de définir deux catégories de véhicules : une avec moteur et une autre sans moteur.

Ainsi, on peut imaginer une hiérarchie de classe avec comme classe de base **TVehicule** avec deux classes héritées, une pour les véhicules à moteur (**TWithMotor**) et l'autre sans moteur (**TWithoutMotor**). À ce stade les classes **TWithMotor** et **TWithoutMotor** représentent encore deux concepts très généraux. Il est alors possible d'ajouter un héritage supplémentaire pour modéliser l'ensemble des produits disponibles à la location, à savoir les vélos (**TBike**), les rollers (**TRollerSkate**), les voitures (**TCar**) et les motos (**TMotorCycle**) qui représentent spécifiquement des objets du monde réel.



Dans un schéma UML, l'héritage entre deux classes est modélisé par une flèche sur fond blanc. Dans le schéma ci-dessus, on a bien **TBike** qui hérite de **TWithoutMotor** et **TWithoutMotor** qui hérite à son tour de **TVehicule**.

Pour rappel, les champs précédés d'un # sont de visibilité protégée.

En reprenant l'exemple ci-dessus, **TCar** et **TMotorCycle** auront accès à la propriété **FTypeOfFuel** qui représente le type de carburant de l'objet à louer. Cette notion n'apparaît pas dans la hiérarchie des véhicules sans moteur **TBike** ou **TRollerSkate** car elle est non applicable pour ces objets du monde réel. Un vélo ne possède pas de type de carburant.

Ainsi, on considère que tous les véhicules possèdent une couleur (**FColor**) et une marque (**FBrand**).

Pour un véhicule sans moteur, le poids devient important, et c'est en positionnant **FWeight** dans la classe **TWithoutMotor** que l'on met l'accent sur cette notion qui sera partagée entre les vélos et les rollers.

Pour un véhicule avec un moteur, il est important de connaître sa puissance et le type de carburant. Ainsi, on positionne ces deux champs dans **TWithMotor** (**FEnginePower** et **FTypeOfFuel**).

Dans les classes modélisant les produits à louer, on retrouve en visibilité privée le casque fourni pour une moto (**FHelmetProvided** dans **TMotorCycle**), le nombre de portes pour une voiture (**FNbDoors** dans **TCar**), le cadenas fourni pour un vélo (**FLockProvided** dans **TBike**) et la protection corporelle genoux, coudes et tête fournie pour un roller (**FBodyProtectionProvided** dans **TRoller**).

Ainsi, on a bien une voiture qui possède un nombre de portes, un type de carburant, une puissance, une marque et une couleur. De même, on a bien un vélo qui possède un cadenas, un poids, une couleur et une marque.

2.6 Polymorphisme

Par polymorphisme on entend la possibilité qu'à une classe enfant de redéfinir une méthode et donc un comportement de la classe parent.

Une méthode qui porte le même nom entre le parent et l'enfant peut posséder la même signature et une implémentation différente.

Pour rappel, la signature d'une routine est constituée de son type de retour (rien, s'il s'agit d'une procédure ou du type renvoyé par une fonction) associé à la liste ordonnée des paramètres d'entrée de la routine. La signature de procedure **MyProc(param1:string;param2:integer);** est différente de **procedure MyProc(param1,param2:string);**. Dans un cas, les paramètres d'entrée sont un entier et une chaîne de caractères et dans l'autre, deux chaînes de caractères.

Grâce à ce principe de polymorphisme, le compilateur effectue une liaison dynamique des appels. Ainsi, soit la méthode enfant ou la méthode parent est appelée en fonction du contexte d'exécution.

Il ne faut pas confondre polymorphisme et surcharge car dans le cas du polymorphisme les paramètres d'entrée de la méthode sont les mêmes et dans le cas de la surcharge ils sont différents.

La surcharge est le processus de multiples définitions de routines avec des paramètres différents abordés dans le chapitre précédent. Ces définitions de méthodes utilisent la directive **overload**.

En Delphi, pour utiliser le polymorphisme à travers l'héritage, il faut utiliser au minimum les directives **virtual** et **override** et éventuellement la directive **abstract**.

- La directive **virtual** indique que la méthode est candidate à une redéfinition dans une classe héritée.
- La directive **override** indique que c'est la méthode de la classe héritée qui doit été exécutée.
- La directive **abstract** indique que cette méthode n'a pas d'implémentation dans la classe courante. Il faut utiliser conjointement **abstract** avec **virtual** car sinon cette méthode ne sera jamais implémentée dans aucun contexte, hérité ou non. Dans le cas où une méthode notée comme **abstract** ne possède aucune implémentation dans la hiérarchie de classe, une erreur d'appel à une méthode non implémentée est possible. Un avertissement est levé par le compilateur lors de la compilation.