Les principes de la programmation orientée objet

La notion d'objet est omniprésente lorsque l'on développe avec C#. Nous allons donc voir dans un premier temps ce que représente cette notion, puis nous verrons comment la mettre en œuvre.

La programmation procédurale, telle qu'utilisée avec des langages comme C ou Pascal, définit un programme comme un flot de données qui seront transformées, au fil de l'exécution, par des procédures et fonctions. Aucun lien fort n'existe entre données et actions.

La programmation orientée objet (POO) introduit la notion d'ensembles cohérents de données et d'actions en transposant au monde du développement des concepts communs et intuitifs issus du monde qui nous entoure.

En effet, nous utilisons au quotidien des objets ayant tous des propriétés et des actions qui leur sont associées. Ils peuvent également interagir ou être composés les uns des autres, ce qui permet de former des systèmes complexes.

Une infinité d'analogies existe pour matérialiser ce concept, mais nous choisirons pour cette introduction celle de l'automobile, à la fois suffisamment simple et suffisamment complexe pour illustrer parfaitement les notions associées à la POO.

Une voiture a des propriétés qui lui sont propres, comme sa marque ou sa couleur, ainsi que des actions qui lui sont associées, comme le fait de démarrer, de freiner ou d'allumer les feux et peut déclencher des événements, comme l'allumage d'un voyant du tableau de bord en cas de problème mécanique.

L'existence de ces éléments dans le code C# se traduit par l'écriture de classes. Une classe est un modèle qui définit les propriétés et actions de chacun des objets qui seront créés à partir de lui. L'étape de création d'un objet s'appelle l'instanciation. Dans le cas de notre analogie, on peut considérer qu'une classe est un plan de voiture qui sera livré aux usines. L'instanciation correspond quant à elle à l'étape de fabrication. Comme dans une usine, il est possible de construire une infinité d'objets à partir du même modèle.

L'exécution de l'action d'allumage des feux sur une voiture implique des éléments qui sont masqués à l'utilisateur : ils sont sous le capot et ne sont pas destinés à être utilisés séparément. Ceci va nous permettre d'introduire un des trois piliers de la POO : l'encapsulation.

L'encapsulation

Chaque objet expose les données et les fonctions qui permettent aux autres objets d'interagir avec lui, et seulement celles-ci. Toutes les données propres au fonctionnement de l'objet et inutiles dans le cadre des interactions avec l'extérieur sont masquées. L'objet se comporte ainsi comme une boîte noire exposant seulement certaines données et fonctionnalités dont l'utilisation ne pourra pas corrompre l'état de l'objet.

Dans le cas de notre analogie, on pourrait dire qu'un objet Voiture expose la donnée *Niveau de carburant* en lecture et écriture, la donnée *État des feux* en lecture seule et la fonctionnalité *Allumage des feux*, entre autres.

On constate que l'état des feux ne peut être modifié que par l'exécution de l'action *Allumage des feux*, ce qui permet de garantir que l'état des feux correspond à l'état du levier d'allumage correspondant. D'autre part, le niveau d'essence peut être modifié en remplissant ou en vidant le réservoir directement, ce qui ne peut pas corrompre l'état interne de notre véhicule, l'état *En Panne* étant certes désagréable, mais parfaitement logique et prévisible.

Pour définir le second fondement de la POO, il est possible de présenter l'automobile comme étant un concept générique à partir duquel tous les modèles sont conçus. Toutes les voitures ont le même fonctionnement logique, malgré des détails de conception différents : elles ont toutes quatre roues, un volant, un moteur ainsi qu'une carrosserie dans laquelle tous ces éléments sont imbriqués. Ainsi, dans le contexte de la POO, nous pouvons dire que toute voiture hérite des propriétés et fonctionnalités de notre automobile générique.

L'héritage

L'héritage définit une relation de type "est un" entre deux classes. Cette relation signifie qu'une classe expose les mêmes fonctionnalités et données qu'une autre classe, en ajoutant potentiellement ses propres fonctionnalités. En général, on utilise l'héritage pour implémenter la notion de spécialisation.

Dans notre cas, il serait possible de dire qu'une voiture de type 4x4 "est une" automobile, tout comme il serait possible de dire qu'une voiture de type traction "est une" automobile. On peut donc définir la classe `Automobile` comme étant la classe de base de laquelle héritent les classes `Voiture4x4` et `VoitureTraction`. Les deux classes filles peuvent ainsi bénéficier des fonctionnalités implémentées dans `Automobile` et implémenter les fonctionnalités spécifiques au mode de fonctionnement qu'elles utilisent.

L'héritage est une notion profondément intégrée dans .NET, et donc dans C#. En effet, tous les types héritent, directement ou non, de la classe `System.Object` qui est, par conséquent, la classe mère de toutes les classes et n'hérite donc d'aucune classe. Cette relation d'héritage est implicite : si une classe n'hérite pas explicitement d'une autre classe, alors elle hérite de `System.Object`.

Lorsqu'une classe est la classe fille dans une relation d'héritage, on dit d'elle qu'elle est dérivée de sa classe parente.

Le polymorphisme

Le dernier concept fondamental de la POO est en lien direct avec l'héritage. Le polymorphisme est en effet la capacité d'un objet à être vu sous plusieurs formes. Mais ces formes ne sont pas aléatoires. Elles dépendent directement de la hiérarchie d'héritage dont fait partie l'objet. Chaque classe peut donc être vue sous sa forme propre, mais aussi celle de sa classe mère ainsi que toutes les classes "ancêtres" de sa classe mère. Chaque objet a donc au minimum deux formes possibles : celle définie par sa classe, ainsi que celle définie par le type `Object`.

Dans le cas d'héritage que nous venons de voir, il est donc possible de dire qu'un objet `VoitureTraction` est aussi accessible sous la forme d'un objet de type `Automobile` et d'un objet de type `System.Object`. Ce comportement permet de créer des fonctionnalités relatives aux objets de type `VoitureTraction` et `Voiture4x4` en ciblant les objets de type `Automobile`.