1 Module B.2 : Autour de l'objet

1.1 Éléments partagés entre instances

Pour l'instant, la distinction entre la classe qui décrit les données et le fonctionnement d'un type d'objet et l'instance qui est une entité de ce type d'objet a été clairement établie. Il existe des éléments partagés au niveau du type et non seulement au niveau de l'instance. Les sous-sections suivantes permettent de mettre en lumière plusieurs d'entre eux.

1.1.1 Constantes

Dans les définitions de classes, on peut définir des constantes partagées par toutes les instances de la classe à l'aide du modificateur **const**. Les constantes sont des champs qui doivent être parmi les types suivants : types numériques, **char**, **bool**, **string** et énumérations. Les constantes doivent être initialisées nécessairement au moment de leur déclaration et ne sont plus modifiables après.

Exemple : Constante

```
1  public class Point
2  {
3    // Constante.
4    private const double _pi = 3.14;
5  }
```

Le modificateur **readonly**, moins restrictif, existe pour déclarer des champs qui ne sont qu'en lecture seule. Ces champs peuvent être initialisés au moment de la déclaration et dans un constructeur, donc ces champs peuvent avoir des valeurs différentes dans les différentes instances.

Exemple : Champ en lecture seule

```
1  public class Point
2  {
3    // Champ en lecture seule.
4    private readonly int _dimension;
5    // Champ simple.
6    private double[] _coordonnees;
7    ...
8    // Constructeur.
9    public Point( params double[] coords)
10   {
11     // Modification du champ en lecture seule possible.
12     _dimension = coords.Length;
13     _coordonnees = new double[_dimensions];
14     // Copie profonde, tableau type par référence.
15     for (int i = 0; i < _dimension; i++)
16     {
17       _coordonnees[i] = coords[i];
18     }
19   }
20 }
```

1.1.2 Statiques

Dans l'ensemble, les éléments partagés au niveau du type et non au niveau de l'instance sont définis à l'aide du modificateur **static**.

Champs et propriétés statiques

Un champ statique est unique et est partagé par toutes les instances d'un type. Il est fréquemment utilisé pour gérer un compteur d'instance ou en relation avec **readonly** pour réaliser une constante. La différence avec **const** revient au fait que cette dernière est toujours la même alors qu'un champ **static readonly** peut être initialisé dans le constructeur **static** de la classe donc être différente entre les applications. Les membres statiques sont appelés à partir du nom de la classe et pas à partir de celui d'une de ces instances.

Exemple : Compteur statique

```
1  public class CompteBancaire
2  {
3    ...
4    // Propriété statique : compteur de compte
5    public static NombreComptes { get; private set; }
6
7    // Constructeurs
8    CompteBancaire (string nom)
9    {
10     _possesseur = nom;
11     // Incrémentation du nombre de compte
12     NombreComptes++;
13   }
14   ...
15 }
16 ...
17 Console.WriteLine ("Il y a " + CompteBancaire.NombreComptes + " comptes.");
```

Constructeurs statiques

Un constructeur peut être statique et servir notamment à initialiser les champs et propriétés statiques. Ce constructeur doit être sans paramètres et est appelé une seule fois. Il est appelé lorsque la première instance de la classe est créée ou lors du premier accès à un membre statique. Les seuls autres modificateurs autorisés pour les constructeurs statiques sont **unsafe** et **extern** (ces modificateurs dépassent le cadre de cette formation).

Pour résumer l'ordre d'exécution des processus de création :

— Une fois pour la classe :
- Initialisation des champs statiques avec la valeur par défaut de leur type.
- Initialisation des champs statiques avec leur valeur par défaut si elle existe.
- Appel du constructeur statique.

— Une fois pour chaque création d'instance :
- Initialisation des champs non statiques avec la valeur par défaut de leur type.
- Initialisation des champs non statiques avec leur valeur par défaut si elle existe.

- Appel du constructeur.

Exemple : Constructeur statique

```
1  public class GestionHoraire
2  {
3    public static DateTime DateCourante { get; private set; }
4
5    // Constructeur statique.
6    static GestionHoraire ()
7    {
8      // Valeur différente tous les jours.
9      DateCourante = DateTime.Today;
10   }
11 }
```

Méthodes et classes statiques

Le modificateur **static** peut également s'appliquer aux classes et aux méthodes. Une classe statique ne doit contenir que des éléments statiques. Ces classes servent en général de réserve de méthodes utilitaires à l'instar de la classe **Math** de la bibliothèque **System**. Les méthodes statiques ne peuvent pas accéder aux champs non statiques, sauf si on les passe en paramètres.

Exemple : Classe et méthode statique

```
1  public static class Geometrie
2  {
3    private static double deuxPi = 6.2832;
4
5    public static double Circonference (double rayon)
6    {
7      return deuxPi * rayon;
8    }
9  }
```

1.2 Bonnes pratiques autour des classes

Il existe toute une série de bonnes pratiques à appliquer d'une manière générale lorsque l'on travaille dans le contexte du paradigme de la programmation orientée objet. Quelques-unes de ces règles sont présentées juste après. Bien entendu, le travail du développeur informatique n'étant pas parfait, l'objectif est d'améliorer la lisibilité, la qualité et la sécurité des codes informatiques de manière progressive quand cela est possible. Les règles suivantes sont assez simples pour être appliquées en amont de la rédaction d'un code informatique au moment de la conceptualisation et de la rédaction de la solution au problème fonctionnel posé.

1.2.1 Copie superficielle ou profonde

La première règle naît de la nature même des objets. Les classes sont des types par référence. Lorsque l'on réalise l'affectation d'une instance d'un type X à partir de celle d'une autre instance,

seule une copie de la référence est réalisée. On parle de *copie superficielle*. Certains comportements étranges peuvent se produire en ce cas.

Exemple : Copie superficielle

```
1  public class X
2  {
3    // Propriété publique pour l'exemple.
4    public string Nom { get; set; }
5  }
6  ...
7  X premier = new X();
8  premier.Nom = "Arbre";
9  X second = premier;              // Copie superficielle.
10 Console.WriteLine (second.Nom); // Arbre.
11 premier.Nom = "Fruit";
12 // La valeur que premier et second référencent a changé.
13 Console.WriteLine (second.Nom); // Fruit.
```

Si cela n'est pas désiré, il existe une méthode pour éviter les problèmes liés aux copies superficielles. Par contre, l'utilisation de l'opérateur = conduira toujours à une *copie superficielle*.

1.2.2 Constructeur de copie

C# ne fournit pas par défaut de constructeur de copie (contrairement au C++) mais il peut être défini explicitement par le développeur de la classe. Ce constructeur de copie permet de réaliser une *copie profonde* de l'instance copiée. Attention toutefois avec les types par référence qui doivent avoir eux-mêmes une copie profonde pour éviter la copie superficielle. Les champs avec type par valeur n'ont pas ce problème.

Exemple : Constructeur de copie

```
1  public class X
2  {
3    public string Nom { get; set; }
4    // Constructeur de copie.
5    public X (X autre)
6    {
7      // Copie de la propriété.
8      // string est un type par référence avec copie profonde.
9      Nom = autre.Nom;
10   }
11 }
12 X premier = new X();
13 premier.Nom = "Arbre";
14 X second = new X(premier);      // Appel constructeur de copie.
15 Console.WriteLine (second.Nom); // Arbre.
16 premier.Nom = "Fruit";
17 // La valeur que second référence n'a pas changé.
18 Console.WriteLine (second.Nom); // Arbre.
```

1.3 Objet ou structure de données

Pour le moment, l'on a présenté les classes en suivant le principe général des objets à savoir cacher les champs et le fonctionnement interne en exposant des méthodes fonctionnelles. Ce fonctionnement s'oppose à celui des structures de données (à ne pas confondre avec les structures définies avec **struct**) qui exposent leurs membres en ne fournissant pas de méthodes fonctionnelles. Les deux fonctionnent de manière différente, mais peuvent être représentées à l'aide de classes ou de structures. Penser à bien saisir le rôle de ses classes. Il existe plusieurs mécanismes permettant d'aider l'implémentation de structures de données.

1.3.1 Indexeurs

Les éléments accessibles par un indice ont été vus dans le cadre des tableaux et des collections. On peut implémenter un accès avec des indices à l'aide d'un indexeur. Un indexeur est une propriété particulière qui a pour nom **this**.

```
modificateur type this [liste paramètres]
{
  get { ... } set { ... }
}
```

Si l'on omet l'accesseur **set**, l'indexeur devient en lecture seule. On peut mettre une liste de paramètres qui désignent la liste des indices à renseigner.

Exemple : Indexeur

```
1  public class Phrase
2  {
3    public string[] mots = "Le grand ours brun".Split ();
4
5    // Indexeur.
6   public string this[int numeroMot]
7   {
8     get { return mots[numeroMot]; }
9     set { mots[numeroMot] = value; }
10  }
11 }
12 ...
13 Phrase phrase = new Phrase();
14 Console.WriteLine (phrase[3]); // brun.
15 phrase[3] = "roux";
16 Console.WriteLine (phrase[3]); // roux.
```

1.4 Principe de connaissance minimale

Le principe de connaissance minimale ou heuristique de Déméter est une règle simple stipulant qu'une méthode ne doit pas connaître les détails et le fonctionnement interne des objets qu'elle manipule. L'objectif de ce principe de diminuer la dépendance des objets entre eux. Une moins grande dépendance permet une plus grande maintenance et évolution contrôlée d'un code car dans le cas où l'on doit modifier le comportement et la signature d'un objet, seuls les objets étroitement liés seront à modifier.

Plus précisément, la loi de Déméter stipule qu'une méthode f d'une classe C ne doit appeler que les méthodes des éléments suivants :

— C.

— un objet passé en argument de f.

— un objet contenu dans une variable d'instance de C.

Chaque unité ne doit parler qu'à ses amis et ne pas parler avec les étrangers. Cette loi ne s'applique qu'aux objets et ne concernent pas les structures de données car ces dernières donnent accès à leur contenu interne. Par contre, un exemple concret de violation du principe de connaissance minimale est d'invoquer les méthodes d'objets retournés par des méthodes autorisées.

Exemple : Loi de Déméter

```
1  public class Banque
2  {
3    private List<CompteBancaire> _comptes;
4    ...
5    // Sommer toutes les transactions dans une période donnée.
6    public decimal SommeTtcTransactions (DateTime debut, DateTime fin)
7    {
8      decimal somme;
9      List<Transaction> transactions
10     foreach (CompteBancaire compte in _comptes)
11     {
12       // Appel méthode ListeTransactions.
13       transactions = compte.ListeTransactions (debut, fin);
14       foreach (var transaction in transactions)
15       {
16         // Violation de la loi de Déméter.
17         somme += transaction.MontantTtc ();
18       }
19     }
20   }
21 }
```

L'exemple ci-dessus montre une violation courante du principe de connaissance minimale. La méthode **SommeTtcTransactions** connaît beaucoup trop de choses. En suivant la loi de Déméter, cette méthode devrait avoir accès aux méthodes de **DateTime** (paramètre) et **CompteBancaire** (champ). **Transaction** ne fait pas partie de ces objets car elle est créée par le retour de la méthode **ListeTransaction** (la méthode ne fait que déclarer une liste d'un type, elle ne l'initialise pas). Cette méthode ne devrait pas avoir à communiquer avec une instance de **Transaction** car cela relève le fonctionnement

du cumul des transactions. Ce fonctionnement devrait être caché par la classe **CompteBancaire**. Bien entendu, si les transactions sont de simples structures de données (Montant Ttc est juste une propriété transparente), la loi de Déméter n'est pas violée.