UML – Diagramme de classe

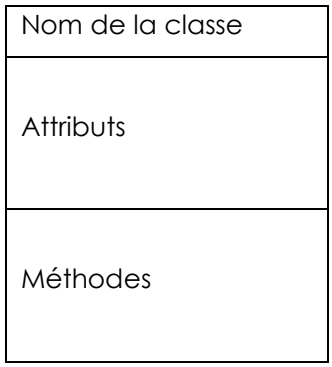
UML (Unified Modeling Language) est un langage de modélisation graphique standardisé pour visualiser, concevoir et documenter les systèmes logiciels. Développé dans les années 1990 par des experts en génie logiciel comme Grady Booch, Ivar Jacobson et James Rumbaugh, UML est devenu une norme pour la modélisation des architectures orientées objet.
UML vise à offrir un langage commun aux développeurs, analystes et autres parties prenantes pour comprendre et communiquer la structure et les comportements d'un système, souvent complexe, avant sa réalisation.
UML comporte trois catégories de diagrammes :

1. Les diagrammes de structure (ou statiques)
2. Les diagrammes de comportement
3. Les diagrammes d'interaction

Il existe une quinzaine de types de diagrammes différents. Vous pouvez consulter un résumé sur Wikipedia, vous pouvez voir le dernier des détails dans la spécification officielle (800 pages !).
Mais pour commencer, dans notre cours, nous allons nous limiter à un type de diagramme par catégorie.
Ce document traite donc du diagramme de classe, un diagramme de structure, statique.

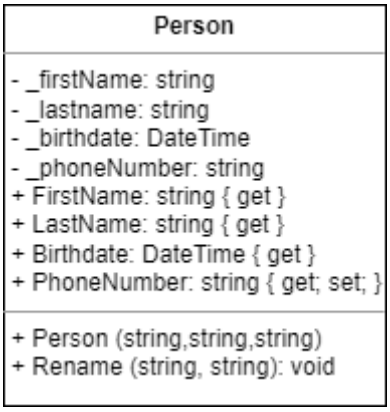
Les classes

Comme son nom le dit, nous allons essentiellement trouver des classes dans un diagramme de classe !
Elles sont représentées par un rectangle divisé en 3 zones :



La première zone se passe de commentaires.
La seconde contient la liste de tous les attributs (et les propriétés C#) quelle que soit leur visibilité. Le type est indiqué.

La troisième contient la liste des signatures de toutes les méthodes, quelle que soit leur visibilité.
Le (ou les) constructeur n'est présenté que si cela apporte de l'information.
Le diagramme de classe est un document technique proche du code. Il est par conséquent dans la même langue que les variables, constantes et méthodes du code (souvent l'anglais).
Exemple :



Explications :

- La visibilité des attributs, propriétés et méthodes sont indiquées en début de ligne par :
  - « - » pour `private`
  - « + » pour `public`
  - « # » pour `protected`
- Le type est indiqué après les « : », autant pour les attributs que pour les méthodes
- La signature des méthodes ne contient que les types de paramètres (pas les noms)
- Les « `{ get ; set ; }` » ne sont pas des éléments formels d'UML. C'est une convention généralement acceptée pour indiquer les propriétés C#

Association

Une association modélise une relation entre deux classes.
Reprenons l'exemple de notre voiture, nous allons réduire sa classe à son strict minimum, à savoir une voiture a une couleur.

```
class Car
{
    /// <summary>
    /// Constructor
    /// </summary>
    /// <param name="color"> color of car </param>
    public Car(string color)
    {
        _color = color;
    }

    /// <summary>
    /// Color attribute (in english, for instance : orange, blue, …)
    /// </summary>
    private string _color;

}
```

Maintenant, nous voulons ajouter un propriétaire à cette voiture. Un propriétaire est une personne qui possède la voiture.

Nous créons une classe pour la personne où nous désirons connaître son prénom et son nom :

```
class Person
{
    public string FirstName { get; set; }

    public void LastName { get; set; }
}
```

Maintenant, nous souhaitons que lors de la construction de la voiture, le nom du propriétaire soit inscrit. A noter, que nous pouvons aussi construire une voiture sans propriétaire (constructeur par défaut). Nous ajoutons donc un propriétaire à la voiture.

```
class Car
{
    public Car(string color)
    {
        _color = color;
    }

    public Car(string color, Person owner)
    {
        _color = color;
        _owner = owner;
    }

    private string _color;
    private Person _owner;

}
```

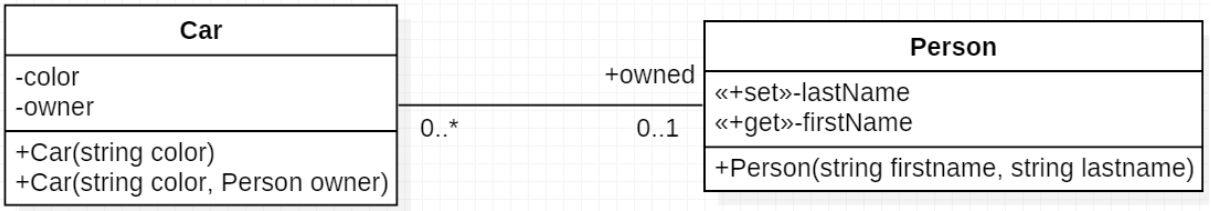
Lors de la création d'une voiture, notre code ressemblera à cela :

```
class Program
{
    /// <summary>
    /// Main
    /// </summary>
    /// <param name="args"></param>
    static void Main(string[] args)
    {

        // Create new person
        Person owner = new Person("Dupond", "Jean");

        // Create two car for Jean Dupond
        Car porsche = new Car("green", owner);
        Car audi = new Car("yellow", owner);

        Console.ReadLine();
    }
}
```

Représentation UML

Tout comme une classe, l'association est représentée en UML. Elle se compose d'un lien, sur ce dernier, sont représentées les multiplicités ainsi que la liaison.

Exemple pour notre voiture et notre propriétaire



Multiplicité des associations

Chaque extrémité d'une association peut porter une indication de multiplicité.

1		Un et un seul
0 .. 1		Zéro ou un
N		N (entier naturel)
M..N		De M à N (entiers naturels)
*	0 .. *	De zéro à plusieurs
1 .. *		De un à plusieurs

Agrégation

Définition et exemple

L'agrégation est un cas particulier d'association. Le principe est que l'une des classes utilise une autre classe. Si l'instance de la classe principale se détruit, celle agrégée n'est pas détruite, elle continue à être disponible.

Reprenons l'exemple de notre voiture, nous allons y ajouter des pneumatiques.
Nous créons à l'intérieur de notre classe une liste de pneumatiques, cela nous permet d'avoir des voitures à 3, 4, 6 roues ou plus…
Maintenant, nous devons définir la classe « Tire » qui va permettre de créer des objets pneumatiques :

Nous créons une classe pour les pneumatiques :

```
class Tire
{
    public enum type { studed, allseason, winter, summer };

    private type _tiretype;
    private string _model;
    private string _brand;

    public type Tiretype
    {
        set { _tiretype = value; }
    }

    public string Model {
         set{_model=value;}
    }

    public string Brand {
         set{_brand=value;}
    }

}
```

Maintenant, nous souhaitons ajouter un pneu à la voiture et ensuite le retirer !

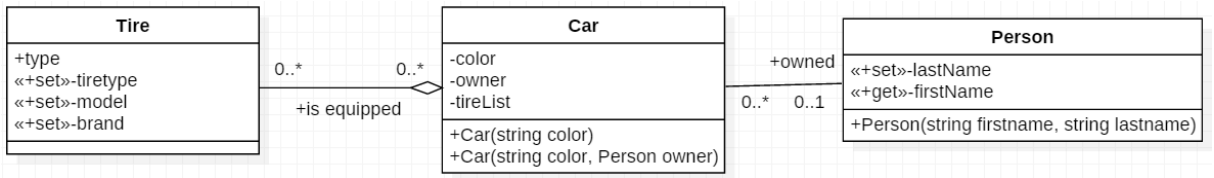
```
static void Main(string[] args)
    class Car
    {  // Create new person
        /// Constructornew Person("Dupond", "Jean");
        public Car(string color)
        {re leftFrontTire = new Tire();
            _color = color;| = "Michelin";
        }ftFrontTire.Model = "EnergySaver";
       leftFrontTire.tiretype = Tire.type.summer;
        /// Constructor
        public Car(string color, Person owner)
        {                                    Car("green", owner);
            _color = color;"yellow", owner);
            _owner = owner;
        }
       porsche.AddTire(leftFrontTire);
        private string _color;e = porsche.RemoveTire(0);
       Console.ReadLine();
        private Person _owner;

        private List<Tire> _tiresList = new List<Tire>();

        public void AddTire(Tire tire)
        {
            _tiresList.Add(tire);
        }

        public Tire RemoveTire(int index)
        {
            Tire removedTire = _tiresList.ElementAt(index);
            // remove for list of tires
            _tiresList.RemoveAt(index);
            return removedTire;
        }
}
```

Lors de la création d'une voiture, nous pouvons y ajouter un pneu et le retirer, celui-ci et indépendant de la voiture et peut être ainsi placé sur une autre voiture.

Représentation UML

L'agrégation est une association, elle se compose d'un losange à une extrémité du lien, sur ce dernier, sont représentées les multiplicités ainsi que la liaison.

Exemple pour notre voiture, notre propriétaire et notre liste de pneumatiques



Composition

Définition et exemple

La composition est un cas particulier d'une agrégation, elle est dit forte. Le principe est que l'une des classes dépend de l'autre pour survivre. Si l'instance de la classe principale se détruit, celle composée est également détruite.

Reprenons l'exemple de notre voiture, nous allons réduire sa classe à son strict minimum, à savoir une voiture à une couleur.

```
class Car
{
    /// Constructor
    public Car(string color)
    {
        _color = color;
    }

    private string _color;

}
```

Maintenant, pour notre voiture, nous avons besoin de son moteur. Ici, ce dernier ne peut appartenir qu'à une voiture et si elle est détruite, son moteur l'est également. (La classe « Engine » devient vitale pour la classe « Car »).

Nous créons une classe pour le moteur :

```
class Engine
{
    /// Constructor by default
    public Engine()
    {  }
}
```

Maintenant, nous souhaitons ajouter le moteur à la construction de la voiture.

```
class Car
{
    /// <summary>
    /// Constructor
    /// </summary>
    /// <param name="color"> color of car </param> >
    public Car(string color)
    {
        _color = color;
        _engine = new Engine();
    }

    private string _color;
    private Engine _engine;

}
```

Lors de la création d'une voiture, notre code ressemblera à cela :

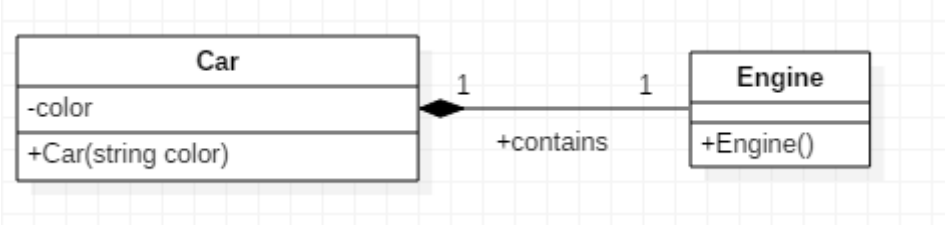
```
class Program
{
    /// <summary>
    /// Main
    /// </summary>
    /// <param name="args"></param>
    static void Main(string[] args)
    {
        // Create a new car
        Car porsche = new Car("green");

        Console.ReadLine();
    }
}
```

Puisque le moteur ne vit qu'au travers de sa voiture, il n'est pas présent dans le programme principal mais créé dans la classe 'Car'.

Représentation UML

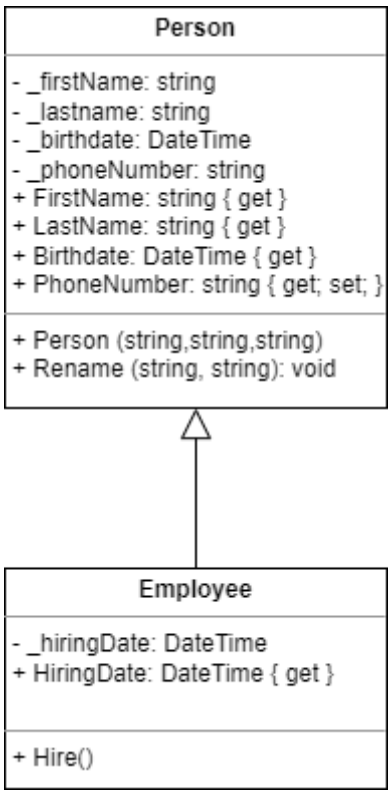
Tout comme une classe, la composition est représentée en UML. Elle se compose d'une ligne qui du côté de la classe « qui a besoin de l'autre pour vivre » à un losange rempli. Les multiplicités sont également représentées sur la ligne.

Exemple pour notre voiture et notre moteur



Héritage

Lorsqu'une classe hérite d'une autre, cela se traduit graphiquement par une flèche dont la pointe est un triangle :



Remarque : très souvent, la lisibilité d'un diagramme de classe contenant à la fois des relations d'agrégation/composition **et** d'héritage en même temps n'est pas bonne. Une bonne pratique consiste à faire des diagrammes séparés.

Interface

L'implémentation d'interface se montre avec une flèche traitillée avec une pointe en triangle.