

# Chapitre 4

## Algorithmique

### 1. Bases de l'algorithmique

Jusqu'à présent, nous nous sommes contentés de développer des applications n'incluant aucune "logique" : elles se limitaient à afficher des données. Il n'y avait aucune notion de condition, de répétition ou même de logique de code. En effet, le code d'une application est souvent complexe et les embranchements sont multiples en fonction de diverses conditions. Dans ce chapitre, nous allons découvrir la logique algorithmique, qui vous permettra de créer du code plus proche de ce que l'on peut retrouver dans les applications répondant à des problématiques plus complexes.

#### 1.1 La logique conditionnelle

Indéniablement, il s'agit ici d'une brique que vous allez utiliser de façon systématique. Une condition implique l'exécution ou non d'une partie du code en fonction de l'évaluation d'un test logique.

## 1.1.1 Test simple : le if/else

La logique conditionnelle se traduit en pseudocode de la façon suivante :

```
SI une condition ALORS
  Je fais quelque chose
SINON
  Je fais autre chose
```

En C#, les mots-clés pour réaliser une instruction conditionnelle sont `if` et `else` :

```
if(condition)
{
    ....
}
else
{
    ....
}
```

La condition testée par une instruction `if` doit renvoyer un booléen. Ce dernier peut être stocké dans une variable mais il est également possible que l'instruction `if` évalue directement la condition, sans variable intermédiaire.

Si on reprend l'exemple de la fin du chapitre précédent, on pourrait améliorer notre classe `Voiture` pour rajouter un booléen qui indique si l'instance de la voiture est fonctionnelle. Si la valeur est égale à "oui", il est inutile de réparer la voiture. Cependant, si la voiture n'est pas fonctionnelle, il faut la réparer :

```
public class Voiture
{
    public bool Fonctionnelle { get; set; }
    ...
}
public class Garage
{
    public void Repare(Voiture voiture)
    {
        if(voiture.Fonctionnelle)
        {
            Console.WriteLine("La voiture n'a pas besoin d'être
réparée car elle est fonctionnelle");
        }
    }
}
```

```
        else
        {
            Console.WriteLine("Réparation de la voiture");
            voiture.Fonctionnelle = true;
        }
    }
}
```

Comme on le voit dans le code ci-dessus, l'instruction `if` se base sur la valeur booléenne stockée dans la propriété `Fonctionnelle` de la classe `voiture` pour évaluer si oui ou non la réparation est nécessaire. Ici, le test a été fait de telle sorte que l'on vérifie si la condition est vraie, et dans le cas inverse, on effectue la réparation. On peut très bien inverser la condition initiale, en comparant le booléen à la valeur `false`. De ce fait, on peut même se passer du `else`, qui n'apporte pas réellement de plus-value :

```
public void Repare(Voiture voiture)
{
    if(voiture.Fonctionnelle == false)
    {
        Console.WriteLine("Réparation de la voiture");
        voiture.Fonctionnelle = true;
    }
}
```

À noter également qu'il est possible d'inverser la valeur d'un booléen en mettant un point d'exclamation en préfixe. Ainsi, `!true` est égal à `false`, et `!false` est égal à `true`. Même si cela peut sembler compliqué de prime abord, vous verrez que c'est une façon d'écrire qui deviendra rapidement automatique à l'utilisation. Si l'on reprend l'exemple précédent, le code qui utilise l'inversion de valeur avec le point d'exclamation serait le suivant :

```
public void Repare(Voiture voiture)
{
    if(!voiture.Fonctionnelle)
    {
        Console.WriteLine("Réparation de la voiture");
        voiture.Fonctionnelle = true;
    }
}
```

Même si à première vue l'instruction `else` est utilisée pour définir le cas inverse de celui du `if` principal, elle peut également servir de base pour une autre instruction `if` à suivre afin de faire une instruction ayant pour sémantique "sinon si". Il suffit dans ce cas d'ajouter une condition `if` après le `else`. Par exemple :

```
public void DecrireVoiture(Voiture voiture)
{
    if(voiture.Marque == "Ferrari")
    {
        Console.WriteLine("Voiture chère");
    }
    else if(voiture.Marque == "Peugeot")
    {
        Console.WriteLine("Voiture standard");
    }
    else
    {
        Console.WriteLine("Marque de voiture non reconnue");
    }
}
```

À noter que la structure du code conditionnel est très flexible : on peut avoir uniquement une seule instruction `if`, une instruction `if` et son `else` associé, ou un enchaînement de `if` et `else if` (avec ou sans `else final`). La seule impossibilité : avoir une instruction `else` seule, car cette dernière indique forcément l'inverse d'une condition donnée.

#### ■ Remarque

*Pour que ces embranchements soient possibles, il faut bien sûr qu'il y ait des conditions pouvant donner plusieurs résultats. À ce titre, il n'est pas utile de faire un `if`, `else if`, `else` avec un simple booléen, car ce dernier ne pouvant avoir que deux états, un `if` avec un `else` est suffisant.*



Afin d'effectuer un quelconque test sur une donnée d'une classe ou de faire un appel de méthode, il est recommandé de tester si la valeur est bien différente de null. Cela peut se faire de façon "classique" ou grâce au nouvel apport du mot-clé not de C# 9 (comme cela est décrit dans la section à venir Pattern matching) :

```
public void DecrireVoiture(Voiture voiture)
{
    if (voiture != null) // avant C# 9
    {
        ...
    }
    if (voiture is not null) // depuis C# 9
    {
        ...
    }
}
```

Pour éviter ce genre de problèmes, un opérateur de navigation sécurisé a été ajouté en C# 6. Ce dernier permet de n'accéder à une méthode ou de lire une donnée que si la variable n'est pas null. On utilise le point d'interrogation juste après la variable, avant l'appel, et cela permet de se passer de tester la nullité :

```
public void DecrireVoiture(Voiture voiture)
{
    if(voiture?.Marque == "Ferrari")
    {
        Console.WriteLine("Voiture chère");
    }
    else if(voiture?.Marque == "Peugeot")
    {
        Console.WriteLine("Voiture standard");
    }
    else
    {
        Console.WriteLine("Marque de voiture non reconnue");
    }
}
```

Le fonctionnement de cet opérateur est le suivant :

- Si la variable n'est pas `null`, on accède à la propriété ou à la méthode concernée normalement.
- Si la variable est `null` :
  - S'il s'agit d'un appel d'une méthode, et que la méthode ne renvoie rien, elle ne sera pas invoquée ;
  - S'il s'agit d'un appel d'une méthode et que la méthode renvoie une valeur, ou qu'il s'agit d'un appel à une propriété, il faudrait tester si la valeur est différente de `null`. S'il s'agit d'un type référence (d'une classe, comme un `string`), alors il faudrait tester si c'est `null` ou pas, afin de voir si l'appel a été fait. S'il s'agit d'un type valeur, alors le type sera encadré d'un nullable. Par exemple, si le type de retour était un `int`, on obtiendrait un `int?` lors de l'appel, qui serait égal à `null` si la variable était à `null`, ou qui aurait la valeur le cas contraire.

```
public class TestClass
{
    public int Valeur { get; set; }
    public string ValeurString { get; set; }
    public void Methode() { }
    public int MethodeInt()
    {
        return 42;
    }
    public string MethodeString()
    {
        return "valeur";
    }
}

TestClass c = null;
int? valeur = c?.Valeur;
string valeurStr = c?.ValeurString;
c?.Methode();
int? retour = c?.MethodeInt();
string retourStr = c?.MethodeString();
```

## Chapitre 4-2

# Le pattern Chain of Responsibility

### 1. Description

Le pattern Chain of Responsibility construit une chaîne d'objets telle que si un objet de la chaîne ne peut pas répondre à une requête, il puisse la transmettre à son successeur et ainsi de suite jusqu'à ce que l'un des objets de la chaîne y réponde.

### 2. Exemple

Nous nous plaçons dans le cadre de la vente de véhicules d'occasion. Lorsque le catalogue de ces véhicules est affiché, l'utilisateur peut demander une description de l'un des véhicules mis en vente. Si une telle description n'a pas été fournie, le système doit alors renvoyer la description associée au modèle de ce véhicule. Si à nouveau, cette description n'a pas été fournie, il convient de renvoyer la description associée à la marque du véhicule. Une description par défaut est renvoyée s'il n'y a pas non plus de description associée à la marque.

Ainsi, l'utilisateur reçoit la description la plus précise qui est disponible dans le système.

Le pattern Chain of Responsibility fournit une solution pour mettre en œuvre ce mécanisme. Celle-ci consiste à lier les objets entre eux du plus spécifique (le véhicule) au plus général (la marque) pour former la chaîne de responsabilité. La requête de description est transmise le long de cette chaîne jusqu'à ce qu'un objet puisse la traiter et renvoyer la description.

Le diagramme d'objets UML de la figure 4-2.1 illustre cette situation et montre les différentes chaînes de responsabilité (de la gauche vers la droite).

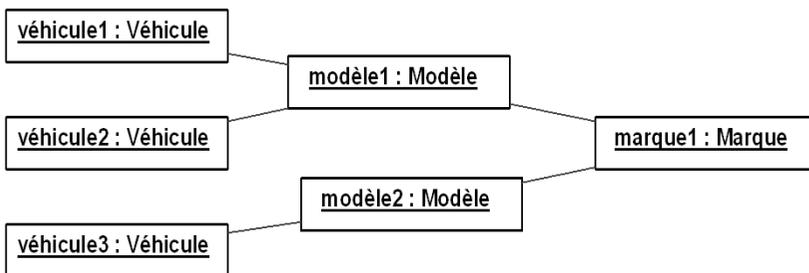


Figure 4-2.1 - Diagramme d'objets de véhicules, modèles et marques avec les liens de la chaîne de responsabilité

La figure 4-2.2 représente le diagramme des classes du pattern Chain of Responsibility appliqué à l'exemple. Les véhicules, modèles et marques sont décrits par des sous-classes concrètes de la classe `ObjetBase`. Cette classe abstraite introduit l'association suivant qui implante la chaîne de responsabilité. Elle introduit également trois méthodes :

- `getDescription` est une méthode abstraite. Elle est implantée dans les sous-classes concrètes. Cette implantation doit retourner soit la description si elle existe soit la valeur `null` dans le cas contraire.
- `descriptionParDéfaut` retourne une valeur de description par défaut, valable pour tous les véhicules du catalogue.

- `donneDescription` est la méthode publique destinée à l'utilisateur. Elle invoque la méthode `getDescription`. Si le résultat est `null`, alors s'il y a un objet suivant, sa méthode `donneDescription` est invoquée à son tour sinon c'est la méthode `descriptionParDéfaut` qui est utilisée.

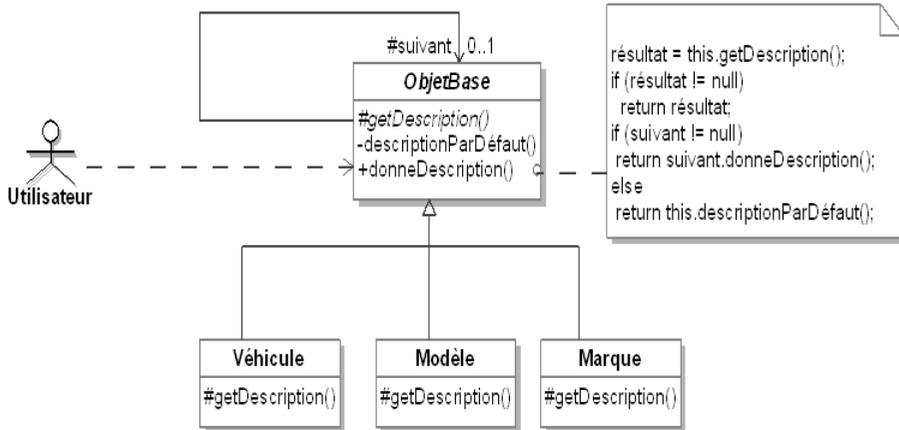


Figure 4-2.2 - Le pattern Chain of Responsibility pour organiser la description de véhicules d'occasion

La figure 4-2.3 montre un diagramme de séquence qui est un exemple de requête d'une description basée sur le diagramme d'objets de la figure 4-2.1.

Dans cet exemple, ni le `véhicule1`, ni le `modèle1` ne possèdent de description. Seule `marque1` possède une description qui est donc utilisée pour le `véhicule1`.

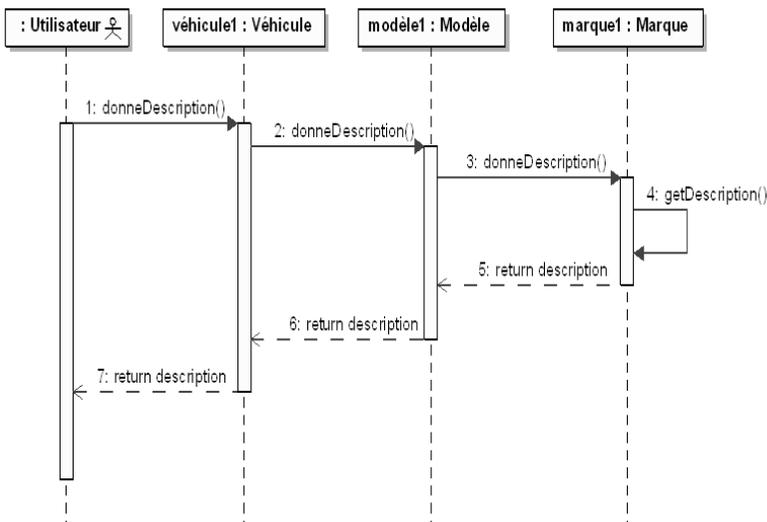


Figure 4-2.3 - Diagramme de séquence illustrant sur un exemple le pattern Chain of Responsibility

## 3. Structure

### 3.1 Diagramme de classes

La figure 4-2.4 détaille la structure générique du pattern.

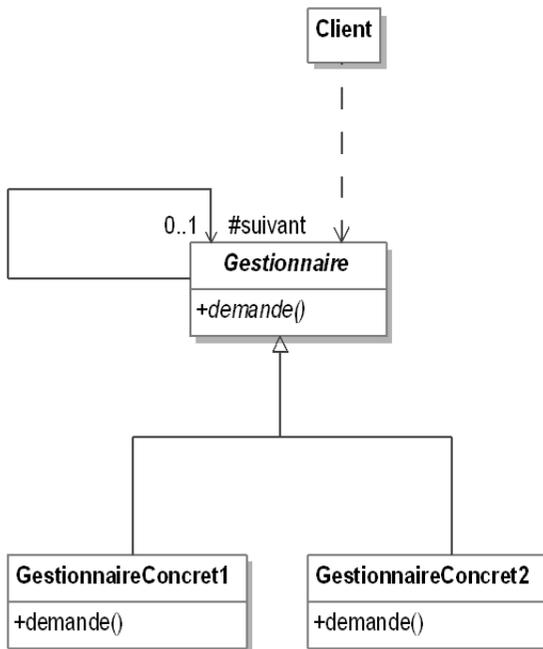


Figure 4-2.4 - Structure du pattern Chain of Responsibility

## 3.2 Participants

Les participants au pattern sont les suivants :

- Gestionnaire (ObjetBase) est une classe abstraite qui implante sous forme d'une association la chaîne de responsabilité ainsi que l'interface des requêtes.
- GestionnaireConcret1 et GestionnaireConcret2 (Véhicule, Modèle et Marque) sont les classes concrètes qui implantent le traitement des requêtes en utilisant la chaîne de responsabilité si elles ne peuvent pas les traiter.
- Client (Utilisateur) initie la requête initiale auprès d'un objet de l'une des classes GestionnaireConcret1 ou GestionnaireConcret2.

## 3.3 Collaborations

Le client effectue la requête initiale auprès d'un gestionnaire. Cette requête est propagée le long de la chaîne de responsabilité jusqu'au moment où l'un des gestionnaires la traite.

## 4. Domaines d'application

Le pattern est utilisé dans les cas suivants :

- Une chaîne d'objets gère une requête selon un ordre qui est défini dynamiquement.
- La façon dont une chaîne d'objets gère une requête ne doit pas être connue de ses clients.

## 5. Exemple en C#

Nous introduisons maintenant l'exemple en langage C#. La classe `ObjetBase` est décrite à la suite. Elle implante la chaîne de responsabilité par la propriété `suisvant`. Les autres méthodes correspondent aux spécifications introduites dans la figure 4-2.2.

```
using System;

public abstract class ObjetBase
{
    public ObjetBase suisvant { protected get; set; }

    private string descriptionParDefaut()
    {
        return "description par défaut";
    }

    protected abstract string description { get; }

    public string donneDescription()
    {
        string resultat;
        resultat = this.description;
        if (resultat != null)
            return resultat;
        if (suisvant != null)
            return suisvant.donneDescription();
        else
            return this.descriptionParDefaut();
    }
}
```

Les trois sous-classes concrètes de la classe `ObjetBase` sont `Vehicule`, `Modele` et `Marque`, leur code source C# est présenté à la suite. La classe `Vehicule` gère une description simple fournie au moment de sa construction (le paramètre `null` est utilisé en cas d'absence de description).

```
using System;

public class Vehicule : ObjetBase
{
    protected string laDescription;
```