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.

Une instruction `if` peut être "compressée" en l'exprimant sous une forme réduite appelée ternaire. Généralement, on utilise cette approche afin d'écrire en ligne un test pour éviter une lourdeur syntaxique, et ce afin d'affecter le contenu d'une variable. La syntaxe est la suivante : on définit en première partie le test à évaluer, séparant d'un point d'interrogation le test des résultats. Ensuite, les cas vrai et faux sont tous deux séparés par un deux-points. La syntaxe est la suivante :

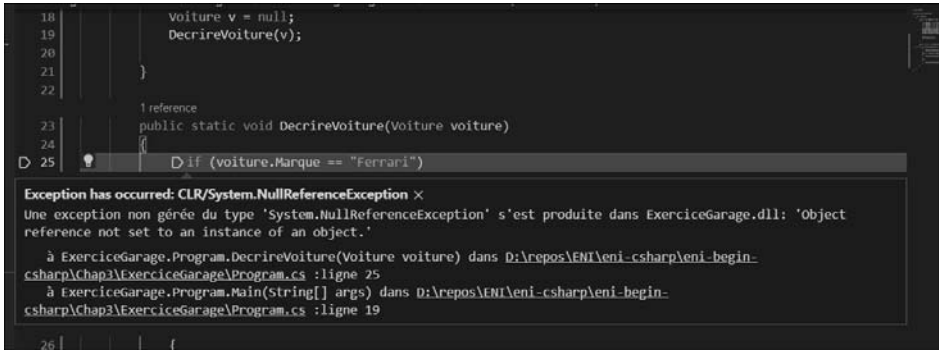
```
test ? cas si vrai : cas si faux
```

Par exemple :

```
string voiture = voiture.Marque == "Ferrari" ? "Voiture chère" :
"Voiture peu chère";
```

Il est possible d'enchaîner les ternaires avec l'usage des parenthèses (en refaisant une autre ternaire dans un cas ou l'autre) mais il est recommandé d'agir avec parcimonie afin de conserver une lisibilité de code optimale.

Il est souvent intéressant de tester si un objet a été affecté avant d'accéder à ses données ou à ses méthodes. En l'absence de ce test, cela peut provoquer une erreur à l'exécution (appelée exception, que nous détaillerons dans ce chapitre à la section La gestion des erreurs). Si on reprend le code précédent, étant donné que `Voiture` est une classe, elle peut donc valoir la valeur `null`. La fonction `DecrireVoiture` tenterait dès lors d'accéder à une variable qui n'a pas de valeur, provoquant une erreur à l'exécution :



Erreur à l'exécution

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 3
Profilage d'une application .NET

1. Gestion de la mémoire par .NET

1.1 Principes de base

La plate-forme .NET comprend un gestionnaire de mémoire en charge de l'affectation et de la libération de la mémoire. Ces deux opérations fonctionnent bien sûr de concert, mais dans un mode très différent de C++ ou d'autres langages où la mémoire est gérée manuellement par le développeur.

En .NET, au lieu de laisser au développeur le soin de gérer ces deux opérations, le runtime prend en charge la totalité de la seconde opération, à savoir le nettoyage de la mémoire. Le développeur réserve de la mémoire en créant une instance d'une classe grâce au mot-clé **new**, mais ne se trouve pas dans l'obligation de la libérer explicitement par la suite (même s'il reste possible de mettre en place des mécanismes avancés de nettoyage de la mémoire). C'est le rôle d'un module de .NET appelé ramasse-miettes (ou *Garbage Collector*, parfois abrégé en GC, en anglais). Ce dernier se chargera, lorsque le besoin s'en fait sentir, de collecter la mémoire désormais inutile et de la remettre à disposition du programme. Nous allons détailler son fonctionnement dans les chapitres suivants.

La majorité des développeurs .NET ne prennent pas garde à la façon dont la mémoire est gérée. Il s'agit ici d'un bel accomplissement de la plate-forme, car si cela est autant transparent, c'est que le processus est efficace. Néanmoins, connaître le fonctionnement de cette gestion permet, dans les cas les plus avancés, de pouvoir répondre et traiter d'éventuels problèmes, que nous détaillerons plus en détail au fil du chapitre.

1.2 Gestion de mémoire automatisée et performances

Comme pour toutes les techniques simplifiant le travail de programmation, des polémiques sont nées sur la perte de performance liée à l'utilisation des ramasse-miettes. Pour couper court à une discussion potentiellement stérile, revenons simplement aux principes d'amélioration de la performance d'une application, tels qu'exposés au début du présent ouvrage. Effectivement, gérer la mémoire manuellement peut permettre de gagner quelques pourcents sur le temps passé à l'exécution dans une fonctionnalité. Mais la mise en œuvre de la gestion manuelle est complexe : même des développeurs expérimentés font des erreurs de gestion de mémoire en C++, et cette gestion peut représenter une part non négligeable du temps affecté au développement.

En résumé, le ratio « temps de codage/performance gagnée » est très mauvais, et dans la majorité des cas, nous aurons avantage à utiliser une technologie de collecte automatisée. D'autant plus que les algorithmes de nettoyage de la mémoire ont été améliorés au fil du temps et disposent de modes d'exécution différents selon l'environnement de destination, ce qui laisse tout de même une petite place à la configuration et l'optimisation des performances à ce niveau.

1.3 Le cas particulier du temps réel

1.3.1 Lever un malentendu

Il existe un champ d'application où les ramasse-miettes posent de réels problèmes, à savoir les applications temps réel. Il est important d'en parler, car la confusion entre les notions de rapidité et de temps réel a amplement participé au mythe des mauvaises performances des langages à gestion de mémoire automatique.

La programmation dite temps réel consiste à proposer des fonctions dont le temps d'exécution est déterministe, c'est-à-dire qu'au moment où le développeur crée une fonctionnalité en appelant des fonctions du langage, il peut calculer à l'avance la durée d'exécution de celle-ci, en additionnant les temps unitaires connus. La notion de temps réel s'arrête là et n'englobe aucune notion de rapidité dans l'exécution.

Remarque

La programmation temps réel est très liée à la théorie de la gestion des risques. Un risque, en termes mathématiques, est décrit par l'équation Risque = Probabilité x Gravité. Appliquons ceci sur le cas d'un logiciel qui doit réagir dans un temps imparti, en supposant qu'il y a une chance sur mille qu'au cours d'une journée d'utilisation, ce ne soit pas le cas (c'est la probabilité). Si la gravité est minime, par exemple, simplement faire attendre une demi-seconde un opérateur, le risque est acceptable. Si à l'inverse la vie de centaines de personnes est en jeu, il est évident que le risque est alors beaucoup trop élevé.

Encore une fois, c'est un malentendu qui est à l'origine de l'association entre "temps réel" et "rapidité d'exécution". Le temps réel est utilisé dans l'aéronautique et l'industrie des satellites, mais pas pour des raisons de performance. C'est d'ailleurs un motif de surprise pour beaucoup de gens de découvrir que les processeurs industriels sont cadencés bien moins haut que les processeurs grand public. Dans un ordinateur portable, le processeur effectue plusieurs milliards d'opérations par seconde (notion de gigahertz), mais au prix d'un système de refroidissement nécessitant l'apport d'air frais extérieur, voire plus (comme du watercooling, ou pour les cas les plus extrêmes, un refroidissement à l'azote). En effet, l'enveloppe thermique, c'est-à-dire la chaleur diffusée par le processeur, augmente exponentiellement avec la fréquence d'opération. Dans un environnement industriel, où des robots sont exposés à la poussière et aux projections, il est évidemment hors de question que le processeur ne soit pas abrité par une protection étanche. Du coup, ces processeurs fonctionnent à une cadence exprimée en millions d'opérations par seconde (ou mégahertz) « seulement », de façon à dégager moins de chaleur et ainsi pouvoir opérer enfermés dans leur coque de protection.

Le risque de surchauffe est bien sûr associé à la panne ; les processeurs modernes sont capables de mettre automatiquement le système en veille, ou au minimum de se couper, avant que la chaleur ne les endommage de manière définitive. Il est donc aisé de comprendre que ce genre de comportement n'est pas tolérable pour certains cas d'usage, comme dans un avion par exemple.

Les processeurs utilisés ne présentent évidemment pas ce genre de comportement. Vu leur fréquence d'utilisation, une surchauffe ne peut provenir que d'un évènement extérieur grave, comme un incendie à bord. Et dans ce cas gravissime, il vaut bien sûr mieux que le processeur (et donc certaines fonctions vitales de l'avion) continue de fonctionner plutôt que de s'arrêter, ce qui ne l'empêchera de toute façon pas de brûler.

Nous touchons là au cœur de la problématique des applications temps réel : quel que soit le contexte, elles doivent continuer à fonctionner de manière déterministe, comme prévu. Mais à nouveau, la performance n'est pas le problème. Un avion ne sera pas plus sûr avec un processeur opérant à 1 GHz qu'avec un processeur opérant à 1 MHz, bien au contraire. En termes de rapidité de réaction, en supposant par exemple que l'envoi d'une commande de vol nécessite quelques dizaines d'opérations, la différence entre les deux systèmes ne sera que de quelques millionièmes de seconde. Ce delta est négligeable. En revanche, même s'il n'y a qu'une chance sur un million que le premier processeur se mette en sécurité parce que, par exemple, le soleil tape un peu plus fort que d'habitude, voudrions-nous monter dans un avion qui en est équipé, sachant qu'un avion moderne vole autour de 100 000 heures sur l'ensemble de sa vie ? Certainement pas...

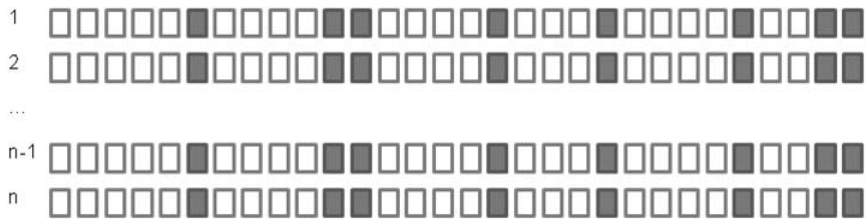
1.3.2 Non-déterminisme des ramasse-miettes

Voilà pourquoi la technologie du ramasse-miettes est incompatible avec les applications temps réel et que nous ne verrons jamais une étiquette "Powered by .NET" sur les commandes de vol d'un avion.

Comme la gestion de la mémoire est prise en charge par le système et non le développeur, il est impossible de savoir à l'avance quand le ramasse-miettes va être déclenché. Le runtime peut choisir de le déclencher parce que le système d'exploitation lui signale qu'il a besoin de mémoire pour une autre application, ou parce qu'il considère grâce à ses propres algorithmes que la mémoire nécessite un nettoyage.

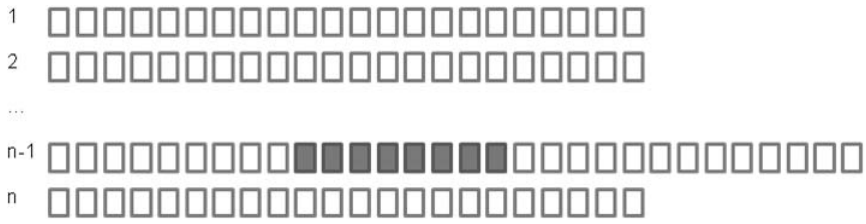
Lorsqu'un développeur utilise un langage où il possède la maîtrise de l'affectation et du nettoyage de la mémoire (comme le C ou le C++), il peut agir de façon régulière, de telle sorte que les opérations d'affectation et de nettoyage se produisent à des moments prévisibles. De la sorte, à chaque lancement de l'application ou à chaque exécution d'un traitement donné, la position de ces opérations ne change pas.

Pour illustrer cela par une image, on peut considérer que chaque rectangle plein constitue une opération de nettoyage de la mémoire, alors que chaque rectangle blanc constitue une lecture ou une affectation de la mémoire :



À chaque lancement de l'application ou du traitement, la position de ces opérations ne change pas.

À l'inverse, le schéma ci-dessous montre le même traitement codé dans un environnement de développement où le moteur d'exécution est désormais responsable du recyclage mémoire. Les rectangles pleins correspondent à l'activation du ramasse-miettes.



Le recyclage de la mémoire est alors aléatoire, le ramasse-miettes se déclenchant au bon vouloir du moteur d'exécution. Pendant certains traitements, il ne se déclenche jamais. Sur d'autres, il est actif à la fin du traitement, voire au début ou même en plein milieu s'il le faut.

Le mécanisme de ramasse-miettes est bien plus complexe qu'il n'y paraît. Bien qu'il ne soit pas complètement prédictible, il cherche toujours à s'exécuter de la meilleure manière possible : en attendant au maximum qu'une interaction utilisateur soit terminée, ou que le processeur ne soit plus trop sollicité. Il va même jusqu'à se lancer dans un thread parallèle à celui du traitement courant de l'application pour la gêner le moins possible. Enfin, des mécanismes sophistiqués, que nous détaillons plus loin, lui permettent de faire son travail le plus vite possible, en ne nettoyant que la partie de la mémoire qui a le plus de chance de contenir des espaces recyclables.

Malgré tout cela, il reste un risque minime que le ramasse-miettes se déclenche en plein milieu d'une utilisation du logiciel et provoque un ralentissement notable. L'utilisateur d'une application de gestion ne se rend, la plupart du temps, même pas compte que le serveur a pris une seconde de plus. À l'inverse, il est impossible de prendre le risque, même ténu, qu'un avion ne réagisse plus aux commandes de vol pendant cette même seconde.

Clairement, les applications temps réel sont extrêmement différentes des logiciels standards. Le niveau de test et de validation qui leur est appliqué est bien évidemment beaucoup plus poussé, les plates-formes de développement et/les matériels utilisés sont spécifiques, et surtout leur développement est énormément plus long et coûteux.

Pour revenir aux applications habituelles (c'est-à-dire ne nécessitant pas de temps réel), le lecteur peut être assuré que la perte de performance liée à une gestion automatique de la mémoire est un mythe. Les ralentissements effectifs sont presque toujours invisibles à l'utilisateur. Mais surtout, l'utilisation de cette technique réduit très fortement les fuites de mémoire, alors que seuls un excellent niveau de développement ou une expertise supplémentaire peuvent nous garantir la même chose en C++, au prix dans un cas comme dans l'autre d'un important surcoût.

Remarque

L'analyse du graphique ci-dessus peut amener à la remarque suivante : vu que l'application ne gère normalement pas le recyclage mémoire pendant l'exécution d'une fonction, serait-il possible qu'une application .NET soit alors plus rapide qu'une application C++ ? C'est effectivement le cas : des benchmarks ont montré que des opérations limitées dans le temps et très consommatrices de mémoire pouvaient être traitées ponctuellement plus rapidement en .NET qu'en C++. Et si l'application a des plages de moindre utilisation que le ramasse-miettes peut utiliser, le même résultat peut être atteint de manière globale. À l'inverse, il sera en faveur de C++ si l'application est utilisée de manière continuellement intensive, à condition bien sûr que le codage soit d'une qualité parfaite en termes de gestion des fuites mémoire.

1.4 Affectation de la mémoire

Cette justification du recyclage automatique de la mémoire étant posée, nous allons dans un premier temps nous intéresser à l'opération inverse, à savoir l'affectation de la mémoire. Comment un développeur C# demande-t-il de la mémoire à .NET ?

Le mot-clé **`new`**, bien que son fonctionnement interne soit relativement complexe comme nous allons le voir ci-dessous, est extrêmement simple à utiliser. Considérons deux classes **`Animal`** et **`Chien`**, la seconde héritant de la première :