

420KBB – TP01**Table des matières**

420KBB – TP01	2
<i>Forme du travail.....</i>	<i>2</i>
<i>Modalités de remise</i>	<i>3</i>
<i>Programme principal qui vous est imposé.....</i>	<i>4</i>
Fonctions auxiliaires du programme principal.....	6
<i>Vue aérienne.....</i>	<i>7</i>
Classe Abeille	8
Classe Alvéole	8
Classe Apiculteur	9
Classe Case	9
Classe Config.....	10
Classe Décoration	12
Classe Écran.....	12
Interface IObsMellivore.....	12
Classe Miel	13
Classe Peintre	13
Classe Point2D.....	13
Classe RegistreAbeilles.....	14
Classe RegistreÉvénements.....	14
Classe Surface	14
Classe Vide	15
Algorithmes – classe statique Algos.....	15
<i>Avant de vous mettre au boulot.....</i>	<i>16</i>
Annexe – Rappels de vocabulaire	17

420KBB – TP01

Dans notre monde contemporain, quoiqu'en pensent certains dirigeants que nous omettrons de nommer mais dont la pensée est... confuse, l'environnement et la crise climatique sont des sujets d'une extrême importance. Les actions concrètes de votre Collège en font d'ailleurs foi¹.

L'un des signes que l'environnement est en péril est la réduction de la population d'abeilles. Pourtant, ces chics insectes sont d'une importance immense pour la survie de plusieurs espèces, dont les humains, car elles contribuent directement à la production d'une part considérable de notre alimentation². La possible disparition des abeilles est donc plus qu'une tragédie : il s'agit d'une menace à notre survie collective.

Pour cette raison, l'observation des abeilles et de leur comportement prend une importance non-négligeable dans nos vies. On sollicite donc votre contribution pour la réalisation d'un projet qui, à terme, vise à mieux comprendre (et à protéger!) ces précieuses petites bestioles.

Ce document utilise des termes que vous avez entendu votre chic professeur utiliser depuis le début de votre formation, mais qui sont parfois mal compris. Une annexe nommée Annexe – Rappels de vocabulaire vous est proposée à la fin de cet énoncé et se veut un aide-mémoire si vous en ressentez le besoin.

Forme du travail

Votre travail prendra la forme d'un seul programme, fait de plusieurs fichiers sources. Ce programme devra utiliser la bibliothèque de classes développée au TP00 pour générer des identifiants.

Vous n'aurez pas à modifier le code du TP00 s'il fonctionnait (s'il ne fonctionnait pas, bien sûr, corrigez les bogues de toute urgence!).

Le programme principal sera imposé (voir la section Programme principal qui vous est imposé, plus bas). Pour le reste, vos chics profs vous proposeront des approches pour résoudre les problèmes posés par ce travail pratique, mais vous aurez aussi de la liberté dans la réalisation de plusieurs aspects du travail.

Note : le namespace pour ce programme devrait être `MayoAbeille`, un hommage (un peu oblique) à **Maya l'abeille**³, un personnage de dessin animé des années '70.



¹ <https://clg.qc.ca/milieu-de-vie/environnement/>

² <https://nature-action.qc.ca/pourquoi-il-est-important-de-protger-les-abeilles/>

³ [https://fr.wikipedia.org/wiki/Maya_l'abeille_\(s%C3%A9rie_t%C3%A9l%C3%A9vis%C3%A9e_d'animation\)](https://fr.wikipedia.org/wiki/Maya_l'abeille_(s%C3%A9rie_t%C3%A9l%C3%A9vis%C3%A9e_d'animation))

Modalités de remise

Organisation :	Travail individuel ou en équipe de deux
Date de remise :	Au début de la séance S11 (voir https://h-deb.ca/CLG/Cours/420KBB/index.html#calendrier_par_groupe pour la date exacte pour votre groupe)
Code source imprimé?	Oui
Remise par Colnet?	Oui, dans une archive zip nommée⁴ comme suit : Groupe-NomPrénom-TP01.zip (p. ex. : 05-TromblonGaetan-TP01.zip) Cette archive doit contenir votre projet une fois celui-ci nettoyé (demandez à votre professeur si vous ne comprenez pas cette partie de la consigne) Il faudra aussi livrer une version imprimée au début du cours suivant la remise. Ce sera celle que je corrigerai (la version en ligne servira à fins de tests si j'ai des préoccupations)

⁴ http://h-deb.ca/CLG/Cours/demander-aide.html#remise_travaux

Programme principal qui vous est imposé

Le programme principal imposé sera le suivant (vous le trouverez intégralement sur le site du cours, avec plusieurs fonctions en supplément) :

```
using MayoAbeille;
using GénérateurId;
ConfigInfo? cfg = Config.LireConfig("../../config.json");
Abeille.SYMBOLE = cfg.Abeille.Symbole; Abeille.COULEUR = cfg.Abeille.Couleur;
Alvéole.SYMBOLE = cfg.Alvéole.Symbole; Alvéole.COULEUR = cfg.Alvéole.Couleur;
Miel.SYMBOLE = cfg.Miel.Symbole;      Miel.COULEUR = cfg.Miel.Couleur;
Vide.SYMBOLE = cfg.Vide.Symbole;      Vide.COULEUR = cfg.Vide.Couleur;
Surface surface = new(cfg.Surface.Hauteur, cfg.Surface.Largeur);
try
{
    surface = Populer( Décorer(surface, GénérerDécorations(
        surface, cfg.Surface.Décorations
    ) ), cfg.Surface.Abeilles );
    Apiculteur bill = new("Bill", surface);
}
catch(DécorationsIncompatiblesException)
{
    Environment.Exit(-1); // insérez du code si vous le jugez pertinent
}
catch(PopulationTropNombreuseException)
{
    Environment.Exit(-2); // insérez du code si vous le jugez pertinent
}
catch (SurfacePleineException)
{
    Environment.Exit(-3); // insérez du code si vous le jugez pertinent
}
catch (PositionConflictuelleException)
{
    Environment.Exit(-4); // insérez du code si vous le jugez pertinent
}
Teindre();
while (!(surface.Compter(Miel.SYMBOLE) == 0))
{
    Projeter(surface);
    foreach(Abeille ab in RegistreAbeilles.Instance.Population)
        ab.Agir(surface);
    Décrire(RegistreAbeilles.Instance.Population, new(0, surface.Hauteur));
}
Projeter(surface);
Décrire(RegistreAbeilles.Instance.Population, new(0, surface.Hauteur));
```

En détail :

- On lit du fichier `config.json` l'état initial du programme (appel à `Config.LireConfig`) puis on initialise certains états du programme (note : les mots `COULEUR` et `SYMBOLE` ici sont des attributs de classe mutables qui doivent être initialisés dès la configuration lue).
- On crée une surface d'une certaine dimension (vous pouvez jouer avec la hauteur et la largeur en modifiant le fichier `.json`). Initialement, cette surface sera telle que toutes ses cases seront vides (pas d'abeille, pas de miel, pas d'alvéole)
- On ajoute à cette surface des abeilles, des alvéoles et du miel à l'aide des fonctions `Populer` et `GénérerDécorations`. Ceci est susceptible de lever des exceptions, que le programme proposé ne traite pas mais « avale » pour terminer son exécution avec des codes d'erreur reconnaissables
- On crée ensuite un `Apiculteur` (nommé "Bill") qui surveillera le comportement des abeilles et produira un rapport décrivant certaines de leurs actions
- Par la suite, on trouve un appel à `Teindre` (voir plus bas) pour associer des couleurs à certains symboles, puis on s'engage dans une boucle qui se répétera tant qu'il restera au moins un symbole de `Miel` sur la surface. Cette boucle :
 - Projette (`Projeter`) la surface à la console
 - S'assure que chaque `Abeille` puisse `Agir`
 - Décrit (méthode `Décrire`) les abeilles et ce qu'elles font
- À la fin de la boucle, un dernier affichage est fait pour assurer l'absence de « bruit » sur la console. Ceci conclut l'exécution du programme

Fonctions auxiliaires du programme principal

Dans le programme principal laissé pour vous sur le site du cours, vous trouverez aussi quelques fonctions déjà implémentées (méthodes de classe de la classe `Program`), soit :

- La méthode `Projeter(surf)` qui affichera le contenu de `surf` à la console en tenant compte de la couleur associée à chaque symbole.
- La méthode `Décrire(abeilles, pos)` qui, à partir de la position `pos` à l'écran, affichera le descriptif (`nom`, `position`) de chaque Abeille dans `abeilles`, puis affichera chaque élément obtenu par la méthode `ObtenirÉvénements` de `RegistreÉvénements`.
- La méthode `GénérerPositions(hau, lar)` qui génère tous les points possibles pour une surface 2D débutant à (0,0) et ayant la hauteur `hau` et la largeur `lar`.
- La méthode `Piger(pts, dé)` qui pige un `Point2D` dans `pts` de manière pseudoaléatoire à l'aide de `dé`.
- La méthode `ChoisirDécoration(dé)` qui génère de manière pseudoaléatoire une « décoration » pour une case donnée de la surface.
- La méthode `Teindre` qui associe des couleurs à certains symboles affichables (voir `Classe Peintre`, plus loin).
- La méthode `GénérerDécorations(surf, nbDécos)` qui placera `nbDécos` décorations choisies de manière pseudoaléatoire sur la surface `surf`.
- La méthode `Répertorier(surf, sym)` qui retourne une `List<Point2D>` contenant la position de chaque `Point2D` dans `surf` dont le symbole est `sym`.
- La méthode `Décorer(surface, décorations)` qui place chaque décoration dans `décorations` à l'endroit désiré sur `surface`.
- La méthode `Populer(surface, nbAbeilles)` qui instancie `nbAbeilles` fois la classe `Abeille` et place chacune d'elles sur `surface`.

Ces fonctions sont nécessaires au bon fonctionnement du programme, et vous n'avez pas le droit de les modifier. Vous pouvez toutefois les lire car certaines utilisent des techniques que nous n'avons pas encore couvert en classe.

Vue aérienne

Les attentes minimales pour ce travail vont comme suit. Vous pouvez faire plus que ce qui est demandé, mais vous ne pouvez pas faire moins... en fait, vous *devrez* faire plus que ce qui est demandé car ce document ne mentionne essentiellement que ce qui est public dans les classes à implémenter, laissant tout le reste à vos bons soins et à votre créativité.

N'hésitez pas à utiliser la classe statique `Algos` et à y loger des services généraux qui vous aideront à être plus productives et plus productifs!

Dans ce qui suit, les entités (classes, interfaces et énumérations) se présentent en ordre lexicographique, pas nécessairement dans l'ordre selon lequel vous devriez les écrire. Nous vous suggérons d'analyser le problème avant de programmer. Une exception notable est la classe `Algos`, logée à la toute fin.

Une fois cette analyse faite, nous vous suggérons de commencer par les entités les plus indépendantes, donc par celles qui peuvent être écrites sans dépendre de l'écriture des autres, puis de progresser une étape à la fois vers les entités dépendant graduellement de plus en plus des autres.

Dans chaque classe, vous pouvez ajouter tout membre vous semblant pertinent dans la mesure où ces membres sont privés.

Classe Abeille

Une instance de la classe `Abeille` représentera une abeille. Ses membres publics seront :

- Un attribut de classe mutable `SYMBOLE` de type `char`, destiné à être initialisé une fois la configuration du programme lue.
- Un attribut de classe mutable `COULEUR` de type `ConsoleColor`, destiné à être initialisé une fois la configuration du programme lue.
- Une propriété d'instance mutable `Pos` de type `Point2D`.
- Une propriété d'instance immuable `Id` de type `GénérateurId.Identifiant`.
- Une propriété d'instance calculée `Nom` de type `string` qui sera synthétisée à partir de la méthode `ToString` de `Id`.
- Un constructeur acceptant un `Identifiant` et un `Point2D`, et dont le rôle sera d'initialiser les propriétés d'instance correspondantes.
- Une spécialisation de `ToString` retournant la valeur de la propriété `Nom`.
- Une méthode d'instance `Agir(Surface)` qui (a) examinera les options de déplacement possible sur la `Surface` passée en paramètre à partir de la position `Pos` (note : une `Abeille` peut se déplacer d'une case à la fois, à la verticale ou à l'horizontale, mais pas en diagonale), et s'il y a au moins une option de déplacement qui s'offre à elle (b) se déplacera (en appelant la méthode `Déplacer` de la `Surface`) et mettre à jour `Pos`.

De plus, je vous recommande au moins les membres privés suivants :

- Une propriété d'instance de type `Random`, qui aidera à bien implémenter `Agir`.
- Une méthode de classe `GénérerOptions` acceptant en paramètre une `Surface` et un `Point2D`, dont le rôle sera de générer et de retourner une `List<Point2D>` contenant les positions de déplacement possibles sur la `Surface` à partir du `Point2D`. Note : une `Abeille` peut se déplacer sur une case valide de la surface, mais seulement si cette case est vide ou si elle contient du miel.

Classe Alvéole

La classe `Alvéole` décrira les paramètres d'affichage d'une alvéole. Ce ne sera pas une classe `static` même si ses membres sont pour le moment tous `static` car il est possible que cette classe évolue de manière à devenir instanciable dans le futur. Ses membres publics sont :

- Un attribut de classe mutable `SYMBOLE` de type `char`, destiné à être initialisé une fois la configuration du programme lue.
- Un attribut de classe mutable `COULEUR` de type `ConsoleColor`, destiné à être initialisé une fois la configuration du programme lue.

Classe *Apiculteur*

Une instance de la classe `Apiculteur` représentera un apiculteur, et sera un observateur de mellivores donc cette classe implémentera `IObsMellivore`. Ses membres publics seront :

- Une propriété immuable `Nom` de type `string`.
- Un constructeur acceptant en paramètre une `string` (le nom de l'apiculteur) et une `Surface`. En plus d'initialiser le `Nom` de l'apiculteur, ce constructeur s'assurera d'abonner l'apiculteur à la `Surface` pour être en mesure de réagir aux événements de consommation de miel qui y surviendront.
- Une méthode `MielConsommé` acceptant en paramètre une `Abeille` (qui a mangé du miel) et un `Point2D` (à quel endroit le miel fut consommé). Son rôle sera d'informer le `RegistreÉvénements` de ce qui s'est produit.

Classe *Case*

Une instance de la classe `Case` représentera une case sujette à faire partie d'une `Surface`. Ses membres publics seront :

- Une propriété d'instance mutable `Symbole` de type `char`.
- Un constructeur acceptant en paramètre un symbole et initialisant la propriété correspondante.
- Un constructeur par défaut représentant une case vide (dont le symbole est celui du `Vide`).
- Une propriété d'instance calculée `EstVide` qui s'évaluera à `true` seulement si la case est vide.
- Une propriété d'instance calculée `EstOccupée` qui s'évaluera à `true` seulement si la case n'est pas vide.

Classe Config

La classe `static` nommée `Config` offrira des services pour charger la configuration initiale du système. Elle permettra de consommer un fichier `.json` correctement formulé⁵ et à charger en mémoire l'état initial du programme. Nous vous suggérons de mettre dans un même fichier `Config.cs` la classe `Config` et quelques classes qui lui sont associées (ci-dessous).

Nous vous donnons une partie de la classe `Config` pour vous aider à démarrer :

```
using System.Text.Json;
using System.Text.Json.Serialization;
namespace MayoAbeille;
public static class Config
{
    static public ConfigInfo? LireConfig(string nomFichier)
    {
        string jsonString = LireFichier(nomFichier);
        var options = new JsonSerializerOptions
        {
            Converters = { new JsonStringEnumConverter() }
        };
        return JsonSerializer.Deserialize<ConfigInfo>(jsonString, options);
    }
    // ... votre code pour la classe Config suit
}
// les autres classes de ce fichier suivent :
// classe ConfigInfo
// classe InfoSurface
// class InfoSymbole
```

Portez attention aux lignes en caractères gras : ce sont celles qui réalisent la consommation du fichier `.json` dont le nom est passé en paramètre à `Config.LireConfig`. Certaines données à consommer du fichier sont des `enum`, ce qui explique que le `JsonSerializer` utilise l'option `Converters` initialisée à un `JsonStringEnumConverter`.

Pour la classe `Config`, il vous reste à coder la méthode `LireFichier` qui accepte en paramètre une `string` représentant le nom du fichier à lire et retourne le texte entier de ce fichier. Assurez-vous bien sûr de faire en sorte que le fichier soit bien fermé une fois la lecture terminée, et ce quoiqu'il advienne!

La désérialisation du fichier `.json` implique que le code C# offre des classes correspondant aux éléments qu'on pense trouver dans ce fichier. Vous devrez donc coder ce qui suit; toutes les propriétés seront mutables et publiques (à la fois pour le `get` et le `set`) car elles ne serviront qu'à représenter ce qui est lu du fichier dans le code et non pas à encapsuler d'information.

⁵ Si le fichier `.json` est malformé pour nos fins, le programme plantera *et c'est correct comme ça*.

Classe	Propriétés
ConfigInfo	Surface, de type InfoSurface? Abeille, de type InfoSymbole? Alvéole, de type InfoSymbole? Miel, de type InfoSymbole? Vide, de type InfoSymbole?
InfoSurface	Hauteur, de type int Largeur, de type int Abeilles, de type int Décorations, de type int
InfoSymbole	Symbole, de type char Couleur, de type ConsoleColor

Note importante : pour les propriétés qui sont d'un type énuméré (dans les classes du tableau ci-dessus, on parle seulement du type ConsoleColor), placez l'annotation `[JsonConverter(typeof(JsonStringEnumConverter))]` juste avant la propriété. Cela générera le code requis pour consommer et convertir cette énumération.

Par exemple :

```
// ...
[JsonConverter(typeof(JsonStringEnumConverter))]
public ConsoleColor Couleur{ get; set; }
// ...
```

Classe Décoration

La classe Décoration sera un enregistrement (un record class en termes C#). Voici la totalité du code pour cette classe :

```
internal record Décoration(Point2D Pos, char Symbole);
```

Ceci est équivalent à la classe suivante :

```
internal class Décoration
{
    public Point2D Pos{ get; init; }
    public char Symbole{ get; init; }
    public Décoration(Point2D pos, char symbole)
    {
        Pos = pos;
        Symbole = symbole;
    }
    // on a aussi un ToString() autogénéré, et on peut avoir une sérialisation
    // JSON autogénérée si on ajoute une annotation à cet effet
    // on a aussi Equals(object?), Equals(T?) de IEquatable<T>, GetHashCode(),
    // et les opérateurs ==, !=
}
```

Chouette n'est-ce pas? Vous pouvez lire plus sur le sujet si vous êtes curieuse ou curieux⁶. C'est pour nous un premier contact avec cette écriture concise pour une classe simpliste pour laquelle aucun invariant n'existe.

Classe Écran

La classe static nommée Écran modélisera les dimensions d'un écran console classique. Elle exposera au minimum les services suivants :

- Une constante LIGNES valant 25 (la hauteur classique d'un écran console).
- Une constante COLS valant 80 (la largeur classique d'un écran console).
- Un prédicat Contient acceptant en paramètre un Point2D et retournant true seulement si les coordonnées de ce sont (inclusivement) contenues dans un écran console.

Interface IObsMellivore

L'interface IObsMellivore permettra à des objets intéressés d'être informés de la consommation de miel par des abeilles.

Son contrat se limite à l'implémentation d'une méthode MielConsommé acceptant en paramètre un Abeille? (qui a mangé du miel) et un Point2D? (où cela s'est-il produit).

⁶ <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record>

Classe Miel

La classe `Miel` décrira les paramètres d’affichage du miel. Ce ne sera pas une classe `static` même si ses membres sont pour le moment tous `static` car il est possible que cette classe évolue de manière à devenir instanciable dans le futur. Ses membres publics sont :

- Un attribut de classe mutable `SYMBOLE` de type `char`, destiné à être initialisé une fois la configuration du programme lue.
- Un attribut de classe mutable `COULEUR` de type `ConsoleColor`, destiné à être initialisé une fois la configuration du programme lue.

Classe Peintre

La classe `static` nommée `Peintre` offrira des services de coloration de symboles à l’écran.

- Elle contiendra un dictionnaire permettant d’associer à un symbole (de type `char`) une couleur (de type `ConsoleColor`).
- Elle exposera la méthode publique `Associer(symbole, couleur)` qui ajoutera une association dans le dictionnaire susmentionné.
- Elle exposera aussi la méthode publique `Peindre(pt, c)` où `pt` est un `Point2d` et `c` la `Case` dont il faut peindre le symbole. Son rôle sera :
 - de se positionner à l’endroit demandé à l’écran (je vous recommande d’utiliser la méthode `Console.SetCursorPosition` pour ce faire), puis
 - d’afficher le symbole à cet endroit.
- La couleur utilisée pour l’affichage doit être prise du dictionnaire pour le symbole de `c` (note : si ce symbole n’est pas au dictionnaire, utilisez la couleur blanche pour l’affichage).
- Note : la propriété qui contrôle la couleur de l’affichage du texte à l’écran est `Console.ForegroundColor`.
- Il va de soi que la modification de couleur d’affichage faite par cette méthode est transitoire et que la méthode doit remettre en place la couleur d’affichage qui était en vigueur lors de l’appel avant de compléter son exécution.

Classe Point2D

Une instance de la classe `Point2D` représentera une position sur une surface. Ses membres publics sont :

- Une propriété d’instance immuable `X` de type `int`.
- Une propriété d’instance immuable `Y` de type `int`.
- Un constructeur paramétrique acceptant en paramètre deux `int` (`x` et `y`) et initialisant les propriétés correspondantes.
- Un constructeur par défaut modélisant un point à l’origine.
- Une spécialisation de `ToString` retournant une `string` faite du `X` et du `Y` de ce point, séparés par une virgule.

Classe *RegistreAbeilles*

La classe `RegistreAbeilles` implémentera le schéma de conception Singleton. Ses membres incluront :

- Une propriété de classe publique immuable `Instance` référant au singleton en tant que tel.
- Une propriété d'instance privée `Abeilles` de type `List<Abeille>` destinée à contenir une référence sur chaque `Abeille` du programme.
- Une propriété d'instance publique calculée `Population` qui retournera une copie de `Abeilles`.
- Une méthode d'instance publique `Contient(ab)` où `ab` est de type `Abeille`. Cette méthode retournera `true` seulement si une `Abeille` du nom de `ab` est inscrite au registre.
- Une méthode d'instance publique `Ajouter(ab)` où `ab` est de type `Abeille`. Cette méthode ajoutera `ab` au registre, mais seulement si le registre ne la `Contient` pas encore (elle lèvera `NomExistantException` dans le cas contraire).
- Une méthode d'instance publique `Trouver(pos)` où `pos` est de type `Point2D`. Cette méthode retournera une référence sur l'`Abeille` dont la position est `pos` dans le registre, ou `null` si aucune `Abeille` n'est à cet endroit.

Classe *RegistreÉvénements*

La classe `RegistreÉvénements` implémentera le schéma de conception Singleton. Ses membres incluront :

- Une propriété de classe publique immuable `Instance` référant au singleton en tant que tel.
- Une propriété d'instance privée `Événements` de type `List<string>` qui contiendra le texte de chaque événement signalé pendant l'exécution du programme.
- Une méthode d'instance publique `Ajouter(qui, événement)` où `qui` et `événement` sont tous deux de type `string`, et qui devra ajouter une `string` à `Événements`. Si `qui` vaut "Roger" et si `événement` vaut "fait du vélo", alors la `string` devra prendre la forme "Roger : fait du vélo".
- Une méthode d'instance publique `ObtenirÉvénements(nb)` où `nb` est un entier, qui retournera une `List<string>` contenant au plus les `nb` plus récents événements notés dans `Événements`.

Classe *Surface*

Une instance de la classe `Surface` représente une surface 2D de cases (voir Classe `Case`). Ses membres incluront :

- Une propriété d'instance privée `Abonnés` contenant une liste de `IObsMellivore`.
- Une méthode d'instance publique `Abonner(p)` où `p` est un `IObsMellivore`, qui ajoutera `p` à la liste des abonnés.
- Une propriété privée `Cases` représentant un tableau 2D de `Case`. Cette propriété doit être initialisée à la construction d'une `Surface` et ne peut plus changer de taille par la suite.

- Une propriété d'instance publique et calculée `Hauteur` retournant le nombre de lignes de `Cases`.
- Une propriété d'instance publique et calculée `Largeur` retournant le nombre de colonnes de `Cases`.
- Une propriété publique et calculée `Aire` retournant l'aire de la `Surface`.
- Une méthode d'instance publique `Contient(pos)` où `pos` est de type `Point2D`, qui retournera `true` seulement si `pos` est inclusivement dans les dimensions de `Cases`.
- Un constructeur paramétrique acceptant en paramètre une hauteur et une largeur (deux entiers), puis initialisant `Cases` entièrement avec des instances par défaut de `Case`.
- Une méthode publique `Déplacer(avant, après)` où `avant` et `après` sont de type `Point2D`. Cette méthode doit déplacer le symbole à la position `avant` dans `Cases` vers la position `après` dans `Cases`, plaçant un symbole `Vide` à la position `avant` par la suite. Note : si la position `après` contient du `Miel`, alors il faut informer les `Abonnés` que du miel a été consommé.
- Un indexeur public implémentant les volets `get` et `set` sur la base d'un paramètre de type `Point2D` et donnant accès à la `Case` correspondante dans `Cases`.
- Une méthode publique `Compter(sym)` où `sym` est de type `char`, qui retournera le nombre d'occurrences du symbole `sym` dans `Cases`.

Classe `Vide`

La classe `Vide` décrira les paramètres d'affichage d'une case vide. Ce ne sera pas une classe `static` même si ses membres sont pour le moment tous `static` car il est possible que cette classe évolue de manière à devenir instanciable dans le futur. Ses membres publics sont :

- Un attribut de classe mutable `SYMBOLE` de type `char`, destiné à être initialisé une fois la configuration du programme lue.
- Un attribut de classe mutable `COULEUR` de type `ConsoleColor`, destiné à être initialisé une fois la configuration du programme lue.

Algorithmes – classe statique `Algos`

Il se peut que vous remarquiez, en réalisant ce travail, que certaines fonctions sont particulièrement utiles, mais difficiles à placer dans une classe ou l'autre dû à leur caractère quelque peu général. Nous vous suggérons donc de rédiger une classe statique `Algos` pour loger des services réutilisables mais non-spécifiques... ou encore mieux : d'utiliser celle que nous avons commencé à écrire en classe et de l'enrichir!

Avant de vous mettre au boulot...

Ce travail est plus costaud, et surtout moins directif, que le précédent. Nous vous invitons donc fortement à suivre une démarche de travail inspirée de ce qui suit :

- Ne commencez pas à coder aveuglément. **Ayez un plan**
- Pour avoir un plan, il faut d'abord **lire les consignes avec attention, et se les approprier**
- À plus forte partie, pour ce travail, vous voudrez assurément ajouter des classes et des interfaces à celles listées dans l'énoncé, et ajouter des états et des services aux classes imposées
- Essayez de vous assurer de bien comprendre les relations entre les différents éléments du travail. Ceci peut se faire par un schéma UML, par exemple
- N'oubliez pas les principes vus en classe : DRY, YAGNI, viser une forte cohésion mais un faible couplage, préférer les qualifications d'accès les plus restrictives possibles dans chaque cas, etc.

Amusez-vous bien!

Annexe – Rappels de vocabulaire

Ce document utilise des termes que vous connaissez, mais que certaines et certains confondent parfois, alors juste au cas... Vous trouverez un exemple de chacun de ces termes (ou presque) dans la classe `Carré` ci-dessous :

```
class CôtéInvalideException : Exception;
class Carré
{
    static bool EstCôtéValide(int candidat) => candidat > 0;
    static int ValiderCôté(int candidat) =>
        EstCôtéValide(candidat)? candidat : throw new CôtéInvalideException();
    public Carré(int côté, ConsoleColor couleur)
    {
        Côté = côté;
        Couleur = couleur;
    }
    int côté;
    public int Côté { get => côté; private init { côté = ValiderCôté(value); } }
    public ConsoleColor Couleur { get; set; }
    public int Périmètre => Côté * 4;
    public int Aire => Côté * Côté;
    public bool EstPlusVasteQue(Carré autre) => Aire > autre.Aire;
    // ...
}
```

Une **propriété calculée** (propriété de second ordre) est une propriété calculée ou synthétisée, à laquelle ne correspond pas un attribut. Dans la classe `Carré`, les propriétés `Périmètre` et `Aire` sont des propriétés de second ordre alors que les propriétés `Côté` et `Couleur`, qui reposent chacune directement sur un attribut (même si on ne voit pas ce dernier dans le cas de `Couleur` car il s'agit d'une propriété automatique) sont quant à elles des propriétés de premier ordre (auxquelles correspond une variable).

Un **prédicat** est une fonction booléenne. Dans la classe `Carré`, `EstCôtéValide` est un prédicat de classe et `EstPlusVasteQue` est un prédicat d'instance.

Une classe est **immuable** quand une instance de cette classe n'est pas modifiable une fois construite, et une propriété est **immuable** quand elle n'est pas modifiable une fois construite. Quelque chose qui n'est pas immuable est dit **mutable**. Dans la classe `Carré`, la propriété d'instance `Côté` est immuable, mais la propriété d'instance `Couleur` est mutable, donc `Carré` est mutable.

Note : le caractère immuable ou mutable d'une classe tient à ses propriétés et à ses attributs d'instance (on ne tient pas compte de ses attributs ou de ses propriétés de classe).

Note : qu'une propriété de second ordre n'a essentiellement jamais de mutateur (`set` ou `init`) et est donc essentiellement toujours immuable.

Un **accesseur** est une méthode donnant accès à un état d'un objet ou d'une classe sans permettre de le modifier. Les parties `get` de propriétés sont habituellement des accesseurs, mais d'autres méthodes peuvent aussi jouer ce rôle.

Un **mutateur** est une méthode permettant de modifier un ou plusieurs états d'un objet ou d'une classe. Les parties `set` et `init` de propriétés sont des mutateurs, mais d'autres méthodes peuvent aussi jouer ce rôle.

Une **précondition** est une responsabilité qui incombe à l'appelant d'une fonction. Si l'appelant ne respecte pas les préconditions d'une fonction, cette dernière ne garantit pas qu'elle fera ce qui lui est demandé.

Une **postcondition** est une responsabilité qui incombe à la fonction appelée. Si un appelant d'une fonction respecte les préconditions de cette fonction, alors la fonction appelée s'engage à respecter ses postconditions.