420KBB – TP01

Table des matières

420KBB – TP01

Nous sommes dans un futur semi-lointain. Les ordures, déchets et autres rebuts sont devenus l'un des principaux enjeux pour la survie de l'humanité; en effet, notre espèce a choisi, depuis des décennies, d'expédier ses ordures, déchets et autres rebuts au-delà des cieux, et a pollué de nombreux autres astres tout comme elle polluait autrefois la Terre.

La situation ne peut plus durer. Une nouvelle gamme de robots d'entretien haut-de-gamme nommée `WAL-1D` a été conçue pour collecter ces ordures, déchets et autres rebuts, et aider à graduellement réduire la pollution environnementale pour éviter la disparition de l'espèce humaine (et de la plupart des autres espèces, on peut le craindre).

Vous faites partie des éminent(e)s scientifiques œuvrant à la mise au point de l'intelligence artificielle des robots de la gamme `WAL-1D`. Votre rôle est de mettre au point un simulateur de collecte d'ordures sur surface plane.

Ce document utilise des termes que vous avez entendu vos professeurs 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, à ceci près qu'il devra utiliser le système développé au `TP00` pour générer des identifiants. Vous n'aurez pas à modifier ce générateur 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.

Un exécutable de démonstration, montrant comment le programme est supposé se comporter, sera déposé sur h-deb pour vous divertir. Vos chics profs vous donneront le lien à utiliser.

Modalités de remise

Organisation :	Travail individuel ou en équipe de deux
Date de remise :	**Vendredi le 11 octobre 2024 à 23 h 59**
Code source imprimé?	Oui (pour les groupes de Patrice)
Remise par Colnet?	Oui, dans une archive zip nommée[1] 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) **Pour les groupes de Patrice**, 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)

[1] http://h-deb.ca/CLG/Cours/demander-aide.html#remise_travaux

Programme principal qui vous est imposé

Le programme principal imposé sera le suivant :

```
// ...
// remplir le catalogue
const char SYMBOLE_MÉTAL = '%';
Catalogue.Get.Associer(SYMBOLE_MÉTAL, Catégorie.Métal);
// préparer la surface d'affichage
const int HAUTEUR = 10,
          LARGEUR = 20;
Surface surf = new(HAUTEUR, LARGEUR);
// préparer le cadre de la surface
Cadre cadre = new(HAUTEUR, LARGEUR, ConsoleColor.Cyan);

// préparer la zone de messagerie
Messagerie messagerie = new(new(0, HAUTEUR));

// positionner Wallyd
Point2D centre = new(3, 3);
Robot wallyd = new("Wallyd", centre);
wallyd.Équiper(new DétecteurMétal(wallyd, 1.0f));
surf.Ajouter(wallyd);

// placer les déchets sur la surface
const int NB_DÉCHETS = 3;
Random dé = new();
List<Déchet> déchets = new();
var libres = surf.TrouverSi
(
   cadre.Exclure,
   c => c == default || c == ' '
);
if (libres.Count < NB_DÉCHETS)
   throw new SurfacePleineException();
for (int i = 0; i != NB_DÉCHETS; ++i)
{
   int n = dé.Next(libres.Count);
   déchets.Add(new(SYMBOLE_MÉTAL, Catégorie.Métal, libres[n]));
   libres.RemoveAt(n);
}
surf.Ajouter(déchets.ToArray());

while (surf.TrouverSi(c => c == SYMBOLE_MÉTAL).Count > 0)
{
   bool trouvé = false;
   do
   {
      PipelineAffichage pipeline = new();
```

```
      pipeline.Ajouter(Appliquer(cadre));
      pipeline.Appliquer
      (
         GénérerHalo(wallyd.Zone, surf.Dupliquer())
      );
      var pts = wallyd.Détecter(surf, Catégorie.Métal);
      if (pts.Count > 0)
      {
         trouvé = true;
         if (pts[0] == wallyd.Pos)
         {
            messagerie.Effacer();
            messagerie.Afficher
            (
               $"Déchet collecté à la position {pts[0]}"
            );
         }
         else
         {
            messagerie.Afficher
            (
               $"Trouvé {pts.Count} déchet(s)",
               $"Déplacement vers {pts[0]}"
            );
            surf.Retirer(wallyd);
            wallyd.DéplacerVers(pts[0]);
            surf.Ajouter(wallyd);
            wallyd.RéinitialiserPuissance();
         }
      }
      else
      {
         wallyd.AugmenterPuissance();
         messagerie.Effacer();
      }
      Thread.Sleep(500);
   }
   while (!trouvé);
   wallyd.RéinitialiserPuissance();
}
{
   PipelineAffichage pipeline = new();
   pipeline.Ajouter(Appliquer(cadre));
   pipeline.Appliquer
   (
      GénérerHalo(wallyd.Zone, surf.Dupliquer())
   );
}
```

```
static Mutable GénérerHalo(Cercle c, Mutable p)
{
   Mutable res = p.Dupliquer();
   for (int ligne = 0; ligne != res.Hauteur; ++ligne)
      for (int col = 0; col != res.Largeur; ++col)
      {
         Point2D pt = new(col, ligne);
         if (c.Centre.Distance(pt) <= c.Rayon)
            res[pt] = new(res[pt].Symbole, res[pt].Avant, ConsoleColor.Green);
      }
   return res;
}

static Func<Mutable, Mutable> Appliquer(Cadre p)
{
   return m =>
   {
      Mutable res = m.Dupliquer();
      for (int i = 0; i != p.Hauteur; ++i)
         for (int j = 0; j != p.Largeur; ++j)
         {
            var c = p[i, j];
            if (c.Symbole != default)
               res[i, j] = c;
         }
      return res;
   };
}
static Func<Mutable, Mutable> AppliquerSurface(Surface surf)
{
   return m =>
   {
      Mutable res = m.Dupliquer();
      for (int i = 0; i != surf.Hauteur; ++i)
         for (int j = 0; j != surf.Largeur; ++j)
         {
            var c = surf[i, j];
            if (c.Symbole != default)
               res[i, j] = c;
         }
      return res;
   };
}

class SurfacePleineException : Exception { }
```

En détail :

- On inscrit au catalogue que le symbole `'%'` correspond à du métal
- On crée une surface d'une certaine dimension (vous pouvez jouer avec la hauteur et le largeur)
- On crée un cadre pour cette surface
- On place notre `Robot`, nommé « wallyd », quelque part sur la surface (vous pouvez jouer avec cette position)
- On génère des déchets métalliques sur la surface (vous pouvez jouer avec la quantité de déchets, dans la mesure du raisonnable)
- Ensuite, tant qu'il reste des déchets, wallyd utilise son détecteur et essaie de les trouver, accroissant graduellement le rayon dudit détecteur. Quand un déchet est trouvé, il se déplace dans sa direction puis reprend les opérations
- Chaque fois que wallyd rejoint la position d'un déchet, il le retire de la carte
- Quand la carte est exempte de déchets, la simulation se termine

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!

Énumération `Catégorie`

Cette énumération indiquera de manière symbolique les catégories de déchets auxquels notre protagoniste pourra être confronté. Pour le moment, les deux seules valeurs possibles seront `Vide` et `Métal`.

Classe statique `DimensionsÉcran`

La classe statique `DimensionsÉcran` exposera au minimum les services suivants :

- Une constante `NB_LIGNES` valant `25` (la hauteur classique d'un écran console)
- Une constante `NB_COLS` valant `80` (la largeur classique d'un écran console)
- Un prédicat `EstDans` acceptant en paramètre un `Point2D` et retournant `true` seulement si ce point est (inclusivement) est dans un écran console décrit par `DimensionsÉcran`

Classe `Point2D`

Le type `Point2D`, une classe immuable modélisant une coordonnée `2D` (`X` et `Y`), devra être implémenté. On s'attend au minimum à ce que les services suivants soient implémentés :

- Des propriétés entières `X` et `Y`
- Un constructeur par défaut, positionnant le point à la l'origine (à la position `0,0`)
- Un constructeur paramétrique
- Une spécialisation de `ToString` exprimant un point sous forme `"(`*X*`,`*Y*`)"` où *X* est la valeur de la propriété `X` et *Y* est la valeur de la propriété `Y`
- Une méthode `Distance` permettant d'évaluer la distance entre deux instances de `Point2D`. Notez qu'on entend ici la distance euclidienne, donc étant donné deux instances de `Point2D` nommées p_0 et p_1, la distance doit s'exprimer sous la forme $\sqrt{(p_0.X - p_1.X)^2 + (p_0.Y - p_1.Y)^2}$

Vous pouvez ajouter d'autres services si vous l'estimez utile et pertinent.

Classe `Cercle`

Une instance de la classe immuable `Cercle` modélise… un cercle! On s'attend au minimum à ce que les services suivants soient implémentés :

- Une propriété `Centre`, de type `Point2D`
- Une propriété `Rayon` de type `float`. Note : le rayon doit être strictement positif; levez `RayonIllégalException` si cette contrainte n'est pas respectée

- Un constructeur par défaut, modélisant un cercle unitaire (centré à l'origine avec un rayon de longueur `1`)
- Un constructeur paramétrique acceptant un rayon, modélisant un cercle centré à l'origine avec le rayon demandé
- Un constructeur paramétrique acceptant un rayon et un centre, modélisant un cercle centré à l'endroit demandé avec le rayon demandé
- Un prédicat d'instance `Contient` acceptant en paramètre un `Point2D` et retournant `true` seulement si le `Cercle` contient ce point. Note : les bordures du cercle sont incluses dans le calcul

Classe `Catalogue`

La classe `Catalogue` sera un singleton qui entreposera des associations, chacune se faisant entre un symbole (de type `char`) et une `Catégorie`. On s'attend au minimum à ce que les services suivants soient implémentés :

- Une propriété de classe `Get` de type `Catalogue`, instanciée au démarrage et se limitant à sa partie `get`
- Un constructeur par défaut privé (car il s'agit d'un singleton)
- Une méthode d'instance `Associer` acceptant en paramètre un `symbole` (de type `char`) et une `Catégorie`, et insérant dans le catalogue un lien entre les deux s'il n'existe pas encore
- Un prédicat d'instance `Est` acceptant en paramètre un `symbole` (de type `char`) et une `Catégorie`, et retournant `true` seulement s'il existe un lien entre les deux

Suggestion : utilisez à l'interne un `Dictionary<char,List<Catégorie>>` pour qu'il soit possible d'associer plusieurs catégories à un même symbole.

Classe `Case`

Une instance de la classe immuable `Case` représentera une case à l'écran. On s'attend au minimum à ce que les services suivants soient implémentés :

- Une propriété d'instance `Symbole` de type `char` (le symbole associé à la case)
- Une propriété d'instance `Avant` de type `ConsoleColor` (la couleur du texte lors d'un affichage)
- Une propriété d'instance `Arrière` de type `ConsoleColor` (la couleur du fond lors d'un affichage)
- Un constructeur paramétrique acceptant un `char` (le symbole) en paramètre. Ce constructeur initialisera `Avant` avec la valeur de la propriété `ForegroundColor` de la classe statique `Console` et `Après` avec la valeur de la propriété `BackgroundColor` de la classe statique `Console`
- Un constructeur paramétrique acceptant un `char` (le symbole) et un `ConsoleColor` (la couleur du texte) en paramètre. Ce constructeur initialisera `Après` avec la valeur de la propriété `BackgroundColor` de la classe statique `Console`
- Un constructeur paramétrique acceptant un `char` (le symbole), un `ConsoleColor` (la couleur du texte) et un autre `ConsoleColor` (la couleur du fond) en paramètre

Interface `IProjetable`

L'interface `IProjetable` exprime le contrat minimal pour une surface susceptible d'être projetée à l'écran. Ce contrat est :

- Une propriété `Hauteur` de type `int`. Le contrat ne doit exiger que la partie `get` de la propriété
- Une propriété `Largeur` de type `int`. Le contrat ne doit exiger que la partie `get` de la propriété
- Un indexeur immuable de type `Case` acceptant en paramètre une ligne et une colonne (dans l'ordre)
- Un indexeur immuable de type `Case` acceptant en paramètre un `Point2D`
- Une méthode `Dupliquer` de type `Mutable` (voir Classe `Mutable`)

Classe `Mutable`

Note : cette classe joue un rôle spécial dans le programme, étant la contrepartie modifiable d'un `IProjetable`, mais ne la confondez pas avec le concept plus général de mutabilité au sens de classe dont les instances peuvent voir leurs états être modifiés même après construction.

La classe `Mutable` implémente `IProjetable`. Elle contient un `Case[,]` privé et expose en plus les services suivants :

- Un constructeur paramétrique acceptant en paramètre un `IProjetable` et copiant les `Case` de cet objet dans son propre `Case[,]`
- Une méthode d'instance `Dupliquer` de type `Mutable` retournant un `Mutable` dont le contenu est une copie de celui de `this`
- Les indexeurs d'un `Mutable` sont mutables (alors que le contrat de `IProjetable` n'exige que des indexeurs immuables), mais c'est légal : une classe implémentant une interface peut faire plus que ce que l'interface exige minimalement, après tout

Interface `IDétecteur`

L'interface `IDétecteur` exprime le contrat minimal pour les opérations d'un détecteur (par exemple un détecteur de métal : voir Classe `DétecteurMétal`). Ce contrat est :

- Une propriété d'instance mutable `Rayon` de type `float`
- Une méthode d'instance `Détecter` acceptant en paramètre un `IProjetable` et retournant un `List<Point2D>`
- Un prédicat d'instance `PeutDétecter` acceptant en paramètre une `Catégorie`
- Une propriété d'instance `Zone` de type `Cercle`. Le contrat ne doit exiger que la partie `get` de la propriété
- Une propriété d'instance `Id` de type `GénérateurId.Identifiant` (voir le `TP00`). Le contrat ne doit exiger que la partie `get` de la propriété

Interface `IPositionnable`

L'interface `IPositionnable` exprime le contrat minimal pour une entité pouvant être placée sur une surface `2D`. Ce contrat est :

- Une propriété d'instance `Pos` de type `Point2D`. Le contrat ne doit exiger que la partie `get` de la propriété

Interface `IIconifiable`

L'interface `IIconifiable` exprime le contrat minimal pour une entité affichable à une position donnée sur une surface `2D`. Tout `IIconifiable` est aussi `IPositionnable`. Ce contrat est :

- Une propriété d'instance `Symbole` de type `char`. Le contrat ne doit exiger que la partie `get` de la propriété

Classe `Déchet`

La classe immuable `Déchet` implémente `IIconifiable` et expose en plus :

- Une propriété d'instance `Famille` de type `Catégorie`, indiquant à quelle catégorie de déchet un `Déchet` appartient
- Un constructeur paramétrique acceptant (dans l'ordre) un symbole (de type `char`), une famille (de type `Catégorie`) et une position (de type `Point2D`)

Classe `Cadre`

La classe immuable `Cadre` implémente `IProjetable` et expose en plus :

- Un constructeur paramétrique acceptant en paramètre (dans l'ordre) une hauteur, une largeur et une couleur (de type `ConsoleColor`)
- Un prédicat d'instance `Exclure` acceptant en paramètre un `Point2D` et retournant `true` seulement si ce point correspond à un coin du `Cadre` ou à un mur du `Cadre`
- Un indexeur de type `Case` acceptant en paramètre une ligne et une colonne (dans l'ordre) et retournant une case de symbole `'+'` s'il s'agit d'un coin, `'-'` s'il s'agit d'un mur horizontal et `'|'` s'il s'agit d'un mur vertical. Si la position décrite n'est pas sur le contour du cadre, le symbole de la case retournée doit être `default`
- Un indexeur de type `Case` acceptant en paramètre un `Point2D` et retournant une case de symbole `'+'` s'il s'agit d'un coin, `'-'` s'il s'agit d'un mur horizontal et `'|'` s'il s'agit d'un mur vertical. Si la position décrite n'est pas sur le contour du cadre, le symbole de la case retournée doit être `default`

Classe `DétecteurMétal`

La classe `DétecteurMétal` implémente `IDétecteur` et expose en plus :

- Une propriété d'instance mutable `Rayon` de type `float`. Note : le rayon doit être au moins `1`; levez `RayonIllégalException` si cette contrainte n'est pas respectée

- Un constructeur acceptant en paramètre une source (de type `IPositionnable`) et un rayon (de type `float`). Un `DétecteurMétal` occupera la même position que sa source, même si cette dernière se déplace
- Une propriété de second ordre `Zone` de type `Cercle` retournant la zone dans laquelle le détecteur fonctionne
- Un prédicat d'instance `PeutDétecter` acceptant une `Catégorie` en paramètre et retournant `true` seulement s'il s'agit de métal
- Une méthode d'instance `Détecter` qui retournera la liste des points de la surface passée en paramètre qui sont (a) contenus dans `Zone` et (b) dont le symbole est considéré comme du métal par le `Catalogue`
- La propriété Id doit être initialisée avec un identifiant provenant du `GénérateurPartagé` avec préfixe `"MET"` (voir le `TP00` pour les détails)

Classe `Robot`

La classe `Robot` implémente `IIconifiable` et expose en plus :

- Une propriété d'instance immuable `Nom` de type `string`. Les règles de validité pour un `Nom` sont qu'il s'agisse d'une référence non-nulle, que sa longueur soit non-nulle et que son caractère à l'indice zéro ne soit pas un blanc (des méthodes du type `char` pourront vous être utiles ici)
- Une propriété de second ordre `Symbole` correspondant à l'élément à l'indice zéro du `Nom` du `Robot`
- Une méthode d'instance `Équiper` acceptant en paramètre un `IDétecteur` (voir Interface `IDétecteur`) Elle aura pour rôle d'informer le `Robot` qu'il possède un détecteur. Pour ce travail, tout `Robot` aura zéro ou un détecteur (truc : gardez cette information dans une propriété mutable privée qui sera `null` par défaut, et considérez qu'un `Robot` est équipé d'un détecteur si cette propriété est non-nulle)
- Une propriété `IdDétecteur` retournant l'identifiant du détecteur ou levant `AucunDétecteurException` si le détecteur est `null`
- Une propriété d'instance `Pos` de type `Point2D` exposant un accesseur public (comme spécifié par l'interface `IIconifiable`) et un mutateur (`set`) privé.
- Un constructeur paramétrique acceptant en paramètre un nom et une position.
- Une méthode d'instance `Détecter` acceptant en paramètre un `IProjetable` et une `Catégorie`, et retournant un `List<Point2D>`. Si le `Robot` n'a pas de détecteur, cette méthode lève `AucunDétecteurException`. Si le détecteur équipé par le `Robot` ne peut pas détecter cette catégorie, elle retourne une liste vide. Sinon, elle retourne le résultat d'un appel à la méthode `Détecter` du détecteur en lui passant le `IProjetable` en paramètre
- Une méthode d'instance `AugmenterPuissance` qui incrémente (*ajoute un à*) le `Rayon` du détecteur équipé par le `Robot`, mais seulement si ce dernier est équipé d'un détecteur (elle n'a pas d'effet dans le cas contraire)
- Une méthode d'instance `RéinitialiserPuissance` qui ramène le `Rayon` du détecteur équipé par le `Robot` à son état « normal » (sans qu'il n'y ait eu d'augmentation), mais seulement si ce dernier est équipé d'un détecteur (elle n'a pas d'effet dans le cas contraire)

- Une propriété de second ordre `Zone` qui expose la `Zone` du détecteur équipé par le `Robot`, et lève `AucunDétecteurException` si le `Robot` n'est pas équipé d'un détecteur
- Une méthode d'instance `DéplacerVers` acceptant en paramètre un `Point2D` représentant la destination vers laquelle le `Robot` se dirige. Cette méthode change la position du `Robot` d'une case en direction de ce point (diagonales incluses) et n'a pas d'effet si le `Robot` est déjà à l'endroit visé

Classe `Surface`

La classe `Surface` implémente `IProjetable` et expose en plus :

- Un indexeur de type `Case` acceptant en paramètre une ligne et une colonne (dans l'ordre) dont l'accesseur sera public (en conformité avec `IProjetable`) et qui implémentera un mutateur (`set`) privé
- Un indexeur de type `Case` acceptant en paramètre un `Point2D` dont l'accesseur sera public (en conformité avec `IProjetable`) et qui implémentera un mutateur (`set`) privé
- Une propriété immuable `Hauteur` de type `int`
- Une propriété immuable `Largeur` de type `int`
- Un constructeur paramétrique acceptant en paramètre une hauteur et une largeur (dans l'ordre) et initialisant les propriétés correspondantes. Ce constructeur doit aussi s'assurer qu'un `Case[,]` dans l'instance de `Surface` soit correctement initialisé de manière à ce que chaque `Case` ait pour symbole un espace (`' '`)
- Une méthode d'instance `Ajouter` acceptant en paramètre autant d'objets implémentant `IIconifiable` que souhaité et modifiant le symbole correspondant à chaque `IIconifiable` à sa position dans la `Surface`, mais ne modifiant ni `Avant`, ni `Après` (les couleurs de texte et de fond). **Précondition** : tous les `IIconifiables` passés en paramètre ont des positions distinctes les unes des autres
- Une méthode d'instance `Retirer` acceptant en paramètre autant d'objets implémentant `IIconifiable` que souhaité et modifiant le symbole correspondant à chaque `IIconifiable` à sa position dans la `Surface` pour le remettre à `default`, mais ne modifiant ni `Avant`, ni `Après` (les couleurs de texte et de fond)
- Une méthode d'instance `Retirer` acceptant en paramètre un `Point2D` et modifiant le symbole à cette position dans la `Surface` pour le remettre à `default`, mais ne modifiant ni `Avant`, ni `Après` (les couleurs de texte et de fond)
- Une méthode d'instance `Ajouter` acceptant en paramètre un `IProjetable` et modifiant le symbole de tous les éléments dans `this` correspondant à une position du `IProjetable` pour lequel le symbole n'est pas `default` par ce symbole. Les propriétés `Avant` et `Arrière` des cases ne sont pas impactés par cette méthode. **Précondition** : si le `IProjetable` passé en paramètre se nomme `p`, alors `p.Hauteur==Hauteur` et `p.Largeur==Largeur`
- Une méthode d'instance `Projeter` acceptant en paramètre un `IProjetable` puis créant un `Mutable` qui soit une copie de `this` et modifiant le symbole de tous les éléments dans ce `Mutable` correspondant à une position du `IProjetable` pour lequel le symbole n'est pas `default` par ce symbole. Cette méthode retourne le `Mutable` résultant.

Précondition : si le `IProjetable` passé en paramètre se nomme `p`, alors `p.Hauteur==Hauteur` et `p.Largeur==Largeur`

- Une méthode d'instance `Projeter` acceptant en paramètre un `IIconifiable` puis créant un `Mutable` qui soit une copie de `this` et modifiant la `Case` à la position de ce `IIconifiable` par une `Case` contenant le symbole, la couleur de texte et la couleur de fond de ce `IIconifiable`. Cette méthode retourne le `Mutable` résultant

Vous devrez aussi ajouter deux méthodes d'instance à cette classe, soit les méthodes `TrouverSi(exclure,pred)` et `TrouverSi(pred)`. Ces deux méthodes utilisent des techniques qui n'ont pas encore été couvertes dans le cours (mais qui le seront, ne vous en faites pas!) alors le code vous est fourni :

```
// ...
      public List<Point2D> TrouverSi
      (
         Func<Point2D, bool> exclure,
         Func<char, bool> pred
      )
      {
         List<Point2D> pts = new();
         for (int ligne = 0; ligne != Hauteur; ++ligne)
            for (int col = 0; col != Largeur; ++col)
            {
               Point2D pt = new(col, ligne);
               if (!exclure(pt) && pred(this[pt].Symbole))
                  pts.Add(pt);
            }
         return pts;
      }
      public List<Point2D> TrouverSi(Func<char, bool> pred) =>
         TrouverSi(pt => false, pred);
   }
// ...
```

Vous trouverez aussi ce code sur https://h-deb.ca/CLG/Cours/420KBB/index.html#tp pour faciliter le copier / coller qui est parfois laborieux avec un document `.pdf`.

Classe `PipelineAffichage`

La classe `PipelineAffichage` est presque entièrement implémentée (pour ce travail pratique, du moins), alors une partie du code vous est fourni :

```
// ...
   internal class PipelineAffichage
   {
      List<Func<Mutable, Mutable>> Transfos = new();
      public void Ajouter(Func<Mutable, Mutable> transfo) =>
         Transfos.Add(transfo);
      public IProjetable Appliquer(Point2D pos, Mutable p)
```

```
      {
         foreach (var transfo in Transfos)
            p = transfo(p);
         return Afficher(pos, p);
      }
      public IProjetable Appliquer(Mutable p) =>
         Appliquer(new(), p);
// ...
```

Vous trouverez aussi ce code sur https://h-deb.ca/CLG/Cours/420KBB/index.html#tp pour faciliter le copier / coller qui est parfois laborieux avec un document `.pdf`.

Il vous faudra toutefois implémenter la méthode d'instance `Afficher`. Cette méthode acceptera en paramètre un point de référence (un `Point2D`) et un `Mutable` (la surface à afficher). Son travail sera :

- De parcourir toutes les `Case` du `Mutable`
- Pour chacune de ces `Case` :
  - Positionner le curseur à la position demandée (la ligne et la colonne de cette position, à laquelle seront ajoutés le `X` du point de référence pour la largeur et le `Y` du point de référence pour la hauteur)
  - À cet endroit, faire une copie des couleurs de texte et de fond, puis écrire le symbole (avec la bonne couleur de texte et de fond) de la `Case`, et enfin remettre en place les couleurs de texte et de fond copiées (autrement dit : remettre les conditions initiales d'affichage en place). **Note** : si le symbole à afficher est `default`, n'affichez rien
- Cette méthode retourne un `IProjetable`. Retournez donc le `Mutable` que vous aurez affiché, qui est aussi un `IProjetable` comme vous le savez

Classe `Messagerie`

Une instance de la classe `Messagerie` exposera les services suivants :

- Un constructeur paramétrique acceptant en paramètre un `Point2D` en paramètre. Ce point représentera le point de référence de la messagerie, donc l'endroit à l'écran où l'affichage de la messagerie débutera
- Une méthode `Afficher` acceptant en paramètre autant de `string` que souhaité. Cette méthode affichera chaque `string` (une par ligne) à partir du point de référence de la messagerie
- Une méthode `Effacer`. Cette méthode remplacera le texte affiché à la console par la `Messagerie` par des blancs

Truc : retenez (à l'interne) la plus grande hauteur et la plus grande largeur d'affichage fait par la `Messagerie`. Ceci vous permettra plus aisément de réaliser `Effacer`.

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.

Avant de vous mettre au boulot…

Ce travail est plus costaud, et surtout moins directif, que les précédents. 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 {}
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 EstPlusGrandQue(Carré autre) => Aire > autre.Aire;
   // ...
}
```

Une **propriété de second ordre** est une propriété calculée ou synthétisée. 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.

Un **prédicat** est une fonction booléenne. Dans la classe `Carré`, `EstCôtéValide` est un prédicat de classe et `EstPlusGrandQue` 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.