420KBB – TP02

Table des matières

420KBB – TP02 ..... **3**
Forme du travail ***3***
Modalités de remise 3
Programme principal qui vous est imposé ***4***
Dans le détail… **10**
Code client ***10***
Bibliothèque de classes ***10***
Classe `Cadre` 10
Classe `CanalComm` 11
Énumération `ActionTy` 11
Classe `InfoCueilleur` 11
Énumération `Catégorie` 12
Classe statique `DimensionsÉcran` 12
Classe `Point2D` 12
Classe `Cercle` 13
Classe `Catalogue` 13
Classe `Case` 14
Interface `IProjetable` 14
Classe `Mutable` 15
Interface `IDétecteur` 15
Interface `IPositionnable` 15
Interface `IIconifiable` 15
Classe `Déchet` 16
Classe `Détecteur` 16
Classe `Robot` 17
Classe `Surface` 18
Classe `PipelineAffichage` 19
Classe `Messagerie` 20
Classe `ZoneTransit<T>` 21

Algorithmes – classe statique `Algos` 21
Avant de vous mettre au boulot… ***22***
Annexe – Rappels de vocabulaire **23**

420KBB – TP02

Nous sommes *toujours* 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. La première version de ce simulateur ayant été un succès, votre tâche est désormais d'implémenter un simulateur pour de multiples `WAL-1D` ayant pour mandat de collecter de multiples formes de déchets plus immondes les uns que les autres.

Une annexe nommée Annexe – Rappels de vocabulaire vous est proposée à la fin de cet énoncé et se veut un aide-mémoire terminologique si vous en ressentez le besoin.

Forme du travail

Votre travail prendra la forme d'un programme console accompagné d'une `.DLL`. Chacun pourra être composé de plusieurs fichiers sources. La `.DLL` 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 22 novembre 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-TP02.zip` (p. ex. : `05-TromblonGaetan-TP02.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 (note : ne le retranscrivez pas, prenez-le de h-deb!) :

```
using KBB_TP01_Wally;
using System;
using System.Diagnostics;
using System.Diagnostics.Metrics;

RobotInfo[] robotsInfo =
{
   new(null, 0, "Wallyd", ConsoleColor.Magenta, Catégorie.Métal),
   new(null, 1, "Bart", ConsoleColor.Blue, Catégorie.Plastique),
   new(null, 2, "Charles", ConsoleColor.Yellow, Catégorie.Organique),
   new(null, 3, "Lisa", ConsoleColor.Green, Catégorie.Nucléaire)
};

(char sym, Catégorie cat)[] symCatégorie =
{
    ('%', Catégorie.Métal),
    ('*', Catégorie.Plastique),
    ('&', Catégorie.Plastique),
    ('/', Catégorie.Organique),
    ('?', Catégorie.Nucléaire),
};
foreach (var symCat in symCatégorie)
    Catalogue.Get.Associer(symCat.sym, symCat.cat);

// préparer la surface d'affichage
const int HAUTEUR = 20,
          LARGEUR = 40;
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(LARGEUR, 0), HAUTEUR);
// préparer la zone d'informations
Messagerie information = new(new(0, HAUTEUR), 2);



Random dé = new();
List<(Robot r, Catégorie c)> robots =
   CréerRobots(robotsInfo, dé, surf, cadre.Exclure);

// placer les déchets sur la surface
const int NB_DÉCHETS = 3;
List<Déchet> déchets =
   CréerPollution(symCatégorie, dé, NB_DÉCHETS, surf, cadre.Exclure);
```

```
surf.Ajouter(déchets.ToArray());

CanalComm canal = new(robotsInfo.Length);

PipelineAffichage pipe = new(surf, cadre, canal, messagerie, information);

pipe.Démarrer();

CancellationTokenSource src = new();
Thread[] cueilleurs =
   CréerCueilleurs
   (
      robotsInfo, surf, canal, messagerie, information, src.Token
   );
Thread lireTouche = new(() =>
{
   Console.ReadKey(true);
   src.Cancel();
});
lireTouche.Start();
foreach (var c in cueilleurs) c.Start();
foreach (var c in cueilleurs) c.Join();
lireTouche.Join();
pipe.Arrêter();

static List<(Robot robot, Catégorie cat)>
   CréerRobots(RobotInfo [] robotsInfo, Random dé, Surface surf,
               Func<Point2D, bool> exclure)
{
   List<(Robot r, Catégorie c)> robots = new();
   // Trouver les cases libres
   var libres = surf.TrouverSi(exclure,c => c == default || c == ' ');
   for (int r = 0; r < robotsInfo.Length; r++)
   {
      RobotInfo info = robotsInfo[r];
      int n = dé.Next(libres.Count);
      Robot robot =
         CréerRobot(info.nom, libres[n], info.couleur, info.catégorie);
      info.robot = robot;
      libres.RemoveAt(n);
   }
   return robots;
}
static Robot
   CréerRobot(string nom, Point2D pos, ConsoleColor couleur, Catégorie cat)
{
   Robot robot = new(nom, pos, couleur);
   robot.Équiper(new Détecteur(robot, 1, cat));  // MB:
```

```
   return robot;
}

static List<Déchet>
   CréerPollution((char sym, Catégorie cat)[] symCatégorie, Random dé,
                  int nbDéchets,
                  Surface surf, Func<Point2D, bool> exclure)
{
   List<Déchet> déchets = new();
   var libres = surf.TrouverSi(exclure, c => c == default || c == ' ');
   if (libres.Count < nbDéchets * symCatégorie.Length)
      throw new SurfacePleineException();
   foreach (var symCat in symCatégorie)
   {
      for (int d = 0; d != nbDéchets; ++d)
      {
         int n = dé.Next(libres.Count);
         déchets.Add(new(symCat.sym, symCat.cat, libres[n]));
         libres.RemoveAt(n);
      }
   }
   return déchets;
}

static Thread[] CréerCueilleurs
   (RobotInfo[] robotsInfo, Surface surf, CanalComm canal,
    Messagerie messagerie, Messagerie information, CancellationToken jeton)
{
   Dictionary<char, int> collectés = new();
   var cueilleurs = new Thread[robotsInfo.Length];
   for (int r = 0; r < robotsInfo.Length; r++)
   {
      var (robot, noPort, _, _, cat) = robotsInfo[r];
      var catalogue = Catalogue.Get;
      cueilleurs[r] = new (() =>
      {
         // tant qu'il reste des déchets à ramasser
         while (!jeton.IsCancellationRequested &&
                surf.TrouverSi(c => catalogue.Est(c, cat)).Count > 0)  // MB::
         {
            bool trouvé = false;
            // Trouver et ramasser le déchet
            do
            {
               canal.PublierSur(noPort, new(robot.Zone, robot.Couleur, robot.Symbole));
               var pts = robot.Détecter(surf, cat);
               if (pts.Count > 0)
               {
```

```
                trouvé = true;
                if (pts[0] == robot.Position)
                {
                   char c = surf[pts[0]].Symbole;
                   lock (collectés)
                      if (collectés.ContainsKey(c))
                         collectés[c]++;
                      else
                         collectés.Add(c, 1);
                   var pos = robot.Position;
                   messagerie.Ajouter
                   (
                      robot.Couleur,
                      $"Déchet collecté à la position {pos}"
                   );
                   surf.Retirer(pos);
                }
                else
                {
                   messagerie.Ajouter
                   (
                      robot.Couleur,
                      $"Trouvé {pts.Count} déchet(s)",
                      $"Déplacement vers {pts[0]}"
                   );
                   robot.DéplacerVers(pts[0]);
                   robot.RéinitialiserPuissance();
                }
                string s = "Collectés : ";
                lock (collectés)
                {
                   foreach (var (sym, n) in collectés)
                      s += $"{sym} x {n}; ";
                   information.Ajouter
                   (
                       ConsoleColor.White, s, "Pressez une touche pour terminer"
                   );
                }
             }
             else
             {
                robot.AugmenterPuissance();
             }
             Thread.Sleep(500);
          }
          while (!trouvé);
          robot.RéinitialiserPuissance();
       }
```

```
      });
   }
   return cueilleurs;
}

class RobotInfo
{
   public Robot? robot;
   public int noPort;
   public string nom;
   public ConsoleColor couleur;
   public Catégorie catégorie;
   public RobotInfo(Robot? robot, int noPort, string nom,
                    ConsoleColor couleur, Catégorie catégorie)
   {
      this.robot = robot;
      this.noPort = noPort;
      this.nom = nom;
      this.couleur = couleur;
      this.catégorie = catégorie;
   }
   public void Deconstruct(out Robot? robot, out int noPort,
                           out string nom, out ConsoleColor couleur,
                           out Catégorie catégorie)
   {
      robot = this.robot;
      noPort = this.noPort;
      nom = this.nom;
      couleur = this.couleur;
      catégorie = this.catégorie;
   }
}

class SurfacePleineException : Exception;
```

En détail :

- On inscrit au catalogue que ce à quoi chaque symbole correspond
- On crée une surface d'une certaine dimension (vous pouvez jouer avec la hauteur et la largeur, mais avec prudence car cela influera sur l'affichage de l'écran dans son ensemble, incluant les messageries)
- On crée un cadre pour cette surface
- On crée deux messageries : une nommée `messagerie` qui sera placée à droite de la surface et affichera les opérations faites par les `WAL-1D`, et l'autre nommée `information` qui sera placée sous la surface et offrira des informations utiles sur la progression des travaux
- On crée et on positionne nos instances de `Robot`
- On génère diverses sortes de déchets sur la surface (vous pouvez jouer avec la quantité de déchets, dans la mesure du raisonnable)

- Ensuite, tant qu'il reste des déchets, nos robots cueilleurs utilisent leurs détecteurs et essaient de les trouver, accroissant graduellement le rayon dudit détecteur. Quand un déchet est trouvé, le `Robot` cueilleur se déplace dans sa direction puis reprend les opérations
- Pendant son travail, chaque fois qu'un `Robot` rejoint la position d'un déchet qu'il aura détecté, il le retire de la carte
- Quand la carte est exempte de déchets, les cueilleurs ont fini leur travail
- La fin de la simulation se produit suivant la pression d'une touche

Notez la présence d'un `CanalComm` servant de canal de communication. Chaque `Robot` sera associé à un numéro de port sur lequel il publiera de l'information. Le `PipelineAffichage` prendra en partie les informations utilisées pour fins d'affichage d'un tel canal.

Dans le détail…

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!

Il est probable qu'une partie du travail soit déjà fait, que ce soit parce que vous avez mené à terme le `TP01` ou parce que vous utiliserez celui fourni par vos chics professeurs à titre de base de travail. **Soyez toutefois très attentives et attentifs aux changements** car les consignes du `TP01` et du `TP02` ne sont pas toujours les mêmes!

Note : dans le texte qui suit, on verra parfois le terme « **cueilleur** » pour entité qui a pour rôle de collecter des déchets. Ce terme s'applique essentiellement aux robots dans ce travail pratique.

Code client

Le code client sera une application console faite pour l'essentiel du code imposé ci-dessus.

Bibliothèque de classes

La `.DLL` contenant l'essentiel du code du `TP02` se déclinera en un ensemble d'entités (fichiers, classes, énumérations) détaillé ci-dessous.

Note : certaines classes sont telles que leurs instances sont susceptibles d'être accédées concurremment par plusieurs fils d'exécution alors faites en sorte qu'elles soient correctement synchronisées.

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 `CanalComm`

La classe `CanalComm` représente un ensemble de canaux de communication. Chaque canal est modélisé par un entier (un numéro de port), et une `ZoneTransit<InfoCueilleur>` est associée à chacun des ports (voir Classe `InfoCueilleur` pour des détails).

Cette classe expose :

- Une propriété publique entière immuable `NbPorts`
- Une propriété privée `Ports` associant à chaque port (un port est représenté par un entier) un `ZoneTransit<InfoCueilleur>`
- Un constructeur public acceptant en paramètre un nombre de ports, puis instanciant chacun des ports de manière à ce que les ports aient les numéros de zéro (inclus) à `NbPorts` (exclu) et que les zones de transit associées soient initialement vides
- Une méthode d'instance privée `ObtenirPort`, acceptant en paramètre un port de type `int` et retournant un `ZoneTransit<InfoCueilleur>`. Si un numéro de port inconnu est passé, la méthode doit lever `PortInexistantException`
- Une méthode d'instance publique `PublierSur`, acceptant en paramètre un numéro de port et un `InfoCueilleur`, et ajoutant cet `InfoCueilleur` à la `ZoneTransit` appropriée
- Une méthode d'instance publique `LireSur`, acceptant en paramètre un numéro de port entier et extrayant la `List<InfoCueilleur>` de la `ZoneTransit` appropriée
- Une méthode d'instance publique `Balayer`, qui parcourra chaque port et retournera une `List<InfoCueilleur>` contenant le plus récent `InfoCueilleur` de chaque port pour lequel `LireSur` retournera une liste non-vide
- Une méthode d'instance publique `Vider` qui videra chacune des `ZoneTransit` associée au `CanalComm`

Énumération `ActionTy`

Cette énumération indiquera de manière symbolique un comportement adopté par un cueilleur. Les valeurs possibles seront `Collecte` et `Recherche`.

Classe `InfoCueilleur`

Une instance de la classe `InfoCueilleur` servira de mécanisme pour communiquer des informations à propos d'un cueilleur (nom qu'on donne parfois à un `Robot`). Cette classe immuable exposera publiquement :

- Une propriété `Zone` de type `Cercle`
- Une propriété `Couleur` de type `ConsoleColor`
- Une propriété `Symbole` de type `char`
- Une propriété `Action` de type `ActionTy`
- Un constructeur paramétrique acceptant en paramètre un `Cercle`, un `ConsoleColor`, un `char` et un `ActionTy` dans l'ordre. Si aucun `ActionTy` n'est passé en paramètre par l'appelant, la valeur `ActionTy.Recherche` devra être utilisée

Énumération `Catégorie`

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

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 susceptible d'être affichée 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 `Arrière` 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, mais il existe d'autres sortes de détecteurs : voir Classe `Détecteur`). 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 `Détecteur`

La classe `Détecteur` 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`), un rayon (de type `float`) et une catégorie (de type `Catégorie`) qui sera la catégorie que le `Détecteur` doit détecter. Un `Détecteur` 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 la catégorie qui lui a été passée à la construction
- 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 associé à la catégorie qu'il doit détecter (vous pouvez valider ceci à travers le `Catalogue`)
- La propriété `Id` doit être initialisée avec un identifiant provenant du `GénérateurPartagé` (voir le `TP00` pour les détails) avec un préfixe exprimé en majuscules et composé d'au plus trois lettres, soit les premières lettre du nom de la catégorie à détecter

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 d'instance immuable `Couleur` de type `ConsoleColor`
- 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, une position et une couleur.
- 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 » (celui qui prévalait avant qu'il n'y ait eu une première 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
- Deux propriétés immuables `Hauteur` et `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)`. Voir le `TP01` pour des détails.

Classe `PipelineAffichage`

La classe `PipelineAffichage` diffère de celle du `TP01` sur quelques aspects, alors soyez attentives et attentifs à ce qui suit :

- Elle possède une propriété privée `Transfos` représentant un `List` de fonctions applicables à un `Mutable` et retournant un `Mutable`
- Elle possède une propriété privée de type `Surface` représentant la surface sur laquelle afficher
- Elle possède une propriété privée de type `Cadre` représentant le cadre à placer autour de cette surface
- Elle possède une propriété privée `Afficheur` de type `Thread`
- Elle possède une propriété privée de type `CanalComm`
- Elle possède deux propriétés privées de type `Messagerie`, nommées respectivement `Mess` et `Info`
- Elle expose un constructeur paramétrique acceptant en paramètre une `Surface`, un `Cadre`, un `CanalComm` et deux `Messagerie` (dans l'ordre, celle qui sera associée à `Mess` et celle qui sera associée à `Info`) et initialisant les propriétés correspondantes. Ce constructeur ajoutera (méthode `Ajouter`) au pipeline le résultat de l'application (méthode `Appliquer`) du `Cadre` pour que les affichages subséquents soient bien encadrés. Ce constructeur sera aussi responsable de créer `Afficheur` et de lui associer la méthode d'instance `Exécuter` (plus bas), mais pas de démarrer ce fil d'exécution

Les services requis pour `PipelineAffichage` sont les suivants :

- Une méthode d'instance publique `Ajouter` acceptant en paramètre une fonction applicable à un `Mutable` et retournant un `Mutable`, et ajoutant celle-ci à `Transfos`
- Une méthode d'instance publique `Appliquer` acceptant en paramètre un `Point2D` nommé `pos` et un `Mutable` nommé `p`, et qui aura pour rôle d'appliquer chaque fonction dans `Transfos` à `p` (en s'assurant de conserver le cumul des transformations), puis retournant le résultat de `Afficher` appliqué aux paramètres `pos` et `p`. Le type de retour ce cette méthode sera `IProjetable`
- Une méthode d'instance publique `Appliquer` acceptant en paramètre un `Mutable` nommé `p`, et qui aura pour rôle d'appliquer chaque fonction dans `Transfos` à `p` (en s'assurant de conserver le cumul des transformations), puis retournant le résultat de `Afficher` appliqué à un `Point2D` par défaut et à `p`. Le type de retour ce cette méthode sera `IProjetable`
- Une méthode d'instance publique `Appliquer` recevant en paramètre en `Cadre` et retournant une fonction applicable à un `Mutable` et retournant un `Mutable`, donc une transformation. La transformation retournée pourra être ajoutée au pipeline pour que chaque affichage affiche aussi le cadre
- Une méthode d'instance publique `Afficher` acceptant en paramètre un `Point2D` nommé `pos`, représentant un point de référence (le coin en haut et à gauche où l'affichage doit se faire à la console), de même qu'un `Mutable` nommé `m`, et qui aura pour rôle d'afficher chaque case de `m` au bon endroit à l'écran (relatif au point de référence) et de la bonne couleur

- Une méthode d'instance publique `GénérerHalo` acceptant en paramètre un `InfoCueilleur` et un `Mutable`. Son rôle sera de `Dupliquer` le `Mutable`, puis d'utiliser la `Zone` et la `Couleur` décrites par le `InfoCueilleur` pour colorier de la couleur demandée les cases qui se trouvent à l'intérieur de cette zone, ce qui résultera en un joli halo. Enfin, elle ajoutera le symbole du cueilleur (indiqué par le `InfoCueilleur`) au centre du halo avant de retourner le `Mutable` ainsi modifié

L'affichage d'un `PipelineAffichage` sera pris en charge par son fil d'exécution `Afficheur`. En ce sens, `PipelineAffichage` exposera aussi les services suivants :

- La méthode publique `Démarrer`, qui s'assurera de démarrer le fil `Afficheur`
- La méthode publique `Arrêter`, qui s'assurera de demander la fin de l'exécution du fil `Afficheur` et attendra la fin de cette exécution avant que la fonction ne se termine
- La méthode publique `Exécuter` qui, tant qu'aucune demande d'arrêt n'aura été reçue, fera les tâches suivantes à chaque `200` millisecondes :
  - Appeler `Balayer` sur le `CanalComm` du `PipelineAffichage`, ce qui lui donnera l'information décrivant les halos des divers cueilleurs
  - Ensuite, s'il y a au moins un halo à afficher, elle dupliquera la `Surface`, puis cumulera des appels à `GénérerHalo` (plus bas) sur cette surface en prenant soin de conserver ce cumul
  - Enfin, elle appliquera `Appliquer` sur la surface résultante, puis affichera les deux `Messagerie` que sont `Mess` et `Info` en appelant leurs méthodes `Afficher` respectives

Classe `Messagerie`

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

- Un constructeur paramétrique acceptant en paramètre un `Point2D` et une hauteur maximale (un entier) en paramètre. Le `Point2D` représentera le point de référence de la messagerie, donc l'endroit à l'écran où l'affichage de la messagerie débutera, alors que la hauteur maximale représentera nombre maximal de lignes qu'on pourra afficher sur la messagerie : si par exemple une messagerie est de hauteur maximale `10`, alors seuls les dix plus récents messages y seront affichés (pas besoin de conserver les plus anciens)
- Une méthode `Ajouter` acceptant en paramètre une couleur (de type `ConsoleColor`) et un nombre arbitrairement grand de messages (chacun étant de type `string`). Elle ajoutera ces messages à ceux entreposés dans la messagerie (ne conservant pas plus que la hauteur maximale possible) et leur associera la couleur demandée
- Une méthode `Afficher` ne prenant pas de paramètre et affichant les messages du plus récent au plus ancien, avec la couleur appropriée pour chaque message. Cette méthode affichera chaque `string` (une par ligne) à partir du point de référence de la messagerie

Cette classe est susceptible d'être utilisée concurremment par plusieurs fils d'exécution, alors assurez-vous que les accès aux états partagés y seront correctement synchronisés.

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 mettre à jour l'affichage à l'écran.

Classe `ZoneTransit<T>`

Une instance de la classe `ZoneTransit<T>` modélisera un canal synchronisé de transfert d'informations de type `T`. Elle exposera les méthodes suivantes :

- Une méthode d'instance publique `Ajouter` acceptant en paramètre un `List<T>` et ajoutant les éléments de cette collection à la fin de celles contenues dans la `ZoneTransit<T>`
- Une méthode d'instance publique `Extraire` ne prenant pas de paramètre et retournant une `List<T>` qui contiendra les éléments qui étaient au préalable contenus dans la `ZoneTransit<T>`, prenant soin de les retirer de la `ZoneTransit<T>` par le fait-même
- Une méthode d'instance `Vider` qui vide le contenu de la `ZoneTransit<T>` sans retourner quoi que ce soit

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.