

420KBB – TP01**Table des matières**

420KBB – TP01a.....	3
<i>Forme du travail.....</i>	<i>3</i>
Modalités de remise	3
<i>Programme principal qui vous est imposé.....</i>	<i>4</i>
<i>Vue aérienne.....</i>	<i>7</i>
Classe CatalogueDéchets.....	7
Énumération Catégorie	7
Classe Cercle.....	8
Classe statique Config et autres classes associées	9
Classe Déchet.....	11
Classe Détecteur	11
Classe statique DimensionsÉcran.....	11
Classe statique FabriqueSurface.....	12
Interface IDétecteur	13
Classe Messagerie	13
Classe Robot	14
Algorithmes – classe statique Algos.....	15
<i>Services de la .DLL fournie pour ce travail (TP01a-Wallyd-Affichage)</i>	<i>16</i>
Classe Cadre	16
Classe Case	16
Classe CatalogueCouleurs	16
Interface IIconifiable.....	16
Interface IPositionnable.....	17
Interface IProjetable.....	17
Classe Mutable	17
Classe PipelineAffichage	18
Classe Point2D.....	18
Classe Surface	18

<i>Avant de vous mettre au boulot.....</i>	20
Annexe – Rappels de vocabulaire	21

420KBB – TP01a

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. Ce programme devra utiliser la bibliothèque de classes développée au TP00 pour générer des identifiants. Il devra aussi utiliser une bibliothèque de classes fournie par vos chics profs pour gérer une partie de l'affichage.

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!). Vous n'aurez pas accès au code source de la .DLL fournie par vos chics profs.

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.

Modalités de remise

Organisation :	Travail individuel ou en équipe de deux
Date de remise :	Vendredi le 10 octobre 2025 à 23 h 59
Code source imprimé?	Oui (pour les groupes de Patrice)
Remise par Colnet?	Oui, dans une archive zip nommée¹ 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 :

```
// ...
using TP01_Wallyd;
using TP01a_Wallyd_Affichage;

ConfigInfo config = Config.LireConfig("../../../config_tpl.json");

char symboleDéchet = config.CatégorieDéchet.Symbole;

// associer les couleurs connues au catalogue de couleurs
CatalogueCouleurs.Get.Associer(config.Robot.Symbole, config.Robot.Couleur);

// préparer la surface d'affichage
Surface surf = FabriqueSurface.Créer(config);

// préparer la zone de messagerie
Messagerie messagerie = new(new(0, surf.Hauteur));

// positionner Wallyd
Robot wallyd = FabriqueSurface.CréerRobot(config.Robot, surf);

// tant qu'il reste des déchets à ramasser
while (surf.TrouverSi(c => c == symboleDéchet).Count > 0)
{
    bool trouvé = false;
    // Trouver et ramasser le déchet
    do
    {
        PipelineAffichage pipeline = new();
        // Ajouter une fonction de transformation au pipeline
        pipeline.Ajouter(pipeline.Appliquer(surf.Cadre));
        // Appliquer le pipeline de transformation au mutable
        // (la surface avec son halo) l'afficher à la position pos(0,0)
        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]}"
    );
    wallyd.DéplacerVers(pts[0], surf);
}
}
else
{
    wallyd.AugmenterPuissance();
    messagerie.Effacer();
}
Thread.Sleep(25);
}
while (!trouvé);
wallyd.RéinitialiserPuissance();
}
//
// dernier affichage une fois les déchets collectés
//
{
    PipelineAffichage pipeline = new();
    pipeline.Ajouter(pipeline.Appliquer(surf.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;
}
}

```

En détail :

- On lit du fichier `config_tp1.json` l'état initial du programme (appel à `Config.LireConfig`)
- On associe le symbole et la couleur du robot, ces informations provenant du fichier de configuration préalablement lu
- On crée une surface d'une certaine dimension (vous pouvez jouer avec la hauteur et la largeur en modifiant le fichier `.json`). Cela mène au positionnement d'un certain nombre de déchets à collecter (ce nombre provient aussi du fichier `.json`)
- On crée une `Messagerie`, zone destinée à accueillir les messages destinés aux usagers
- On place notre Robot, nommé `wallyd`, quelque part sur la surface
- 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 surface
- Quand la surface 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!

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.

Classe *CatalogueDéchets*

La classe `CatalogueDéchets` 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 `CatalogueDéchets`, 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.

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 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 statique *Config* et autres classes associées

La classe statique `Config` servira à consommer un fichier `.json` correctement formulé² 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;
using TP01a_Wallyd_Affichage;
namespace TP01_Wallyd;
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
// classe CatégorieDéchet
// class InfoRobot
```

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.

² 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? CatégorieDéchet, de type CatégorieDéchet? Robot, de type InfoRobot?
InfoSurface	Hauteur, de type int Largeur, de type int
CatégorieDéchet	NbDéchetsParCat, de type int Symbole, de type char Catégorie, de type Catégorie
InfoRobot	Nom, de type string Symbole, de type char Couleur, de type ConsoleColor Catégorie, de type Catégorie

Note importante : pour les propriétés qui sont d'un type énuméré (dans les classes du tableau ci-dessus, on parle du type `Catégorie` de même que 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 Catégorie Catégorie { get; set; }
// ...
```

Classe Déchet

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

- Une propriété d'instance `Id` de type `Identifiant`, initialisée à l'aide d'un générateur partagé avec le préfixe `"MET"` pour le moment (nous élargirons ceci dans un travail pratique ultérieur)
- Une propriété d'instance `Symbole` de type `char`, indiquant le symbole qui servira pour afficher ce déchet
- Une propriété d'instance `Famille` de type `Catégorie`, indiquant à quelle catégorie de déchet un `Déchet` appartient
- Une propriété d'instance `Pos` de type `Point2D`, indiquant l'emplacement sur la surface de ce déchet
- 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
- Une propriété d'instance immuable `Cat` de type `Catégorie`
- Un constructeur acceptant en paramètre une source (de type `IPositionnable`), un rayon (de type `float`) et une `Catégorie`. Un `Détecteur` occupera la même position que sa source, même si cette dernière se déplace
- Une propriété calculée `Zone` de type `Cercle` retournant la zone dans laquelle le détecteur fonctionne. Ce `Cercle` est centré sur la position `Pos` du `Détecteur` et a pour rayon son `Rayon`
- Un prédicat d'instance `PeutDétecter` acceptant une `Catégorie` en paramètre et retournant `true` seulement s'il s'agit de la catégorie `Cat`
- 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 `Cat` par le `CatalogueDéchets`
- Une propriété de classe `Gen` de type `IGénérateurId` qui sera immuable et initialisée avec un générateur séquentiel de votre `TP00` et avec le préfixe `"DET"`. Cela permettra à toutes les instances de `Détecteur` d'avoir des identifiants distincts les uns des autres
- La propriété d'instance immuable `Id` de type `Identifiant`, initialisée avec un identifiant provenant du générateur `Gen` ci-dessus

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 statique *FabriqueSurface*

La classe statique `FabriqueSurface` sert à fabriquer une `Surface` à partir d'informations provenant d'une configuration (voir Classe statique `Config` et autres classes associées, plus haut). On s'attend au minimum à ce que les services suivants soient implémentés :

- Une méthode de `Créer` acceptant en paramètre un `ConfigInfo` et retournant une `Surface`. Cette méthode doit, sur la base du membre `Surface` d'un `ConfigInfo` :
 - instancier une `Surface` de la bonne hauteur et de la bonne largeur
 - instancier le membre `Cadre` de cette `Surface` avec la même hauteur et la même largeur
 - positionner les déchets sur la surface (voir `InitialiserDéchets` ci-dessous; notez que la catégorie de déchets à utiliser est indiquée par le `ConfigInfo`), et
 - retourner la `Surface` ainsi initialisée
- Une méthode `InitialiserDéchets`, acceptant en paramètre une `CatégorieDéchet` et une `Surface`. Cette méthode doit :
 - trouver toutes les cases libres sur la `Surface` (en excluant son `Cadre`)
 - s'il y a moins de cases libres que de déchets à placer (le nombre de déchets à placer est indiqué par la propriété `NbDéchetsParCat` d'une `Catégorie`), lever `SurfacePleineException`
 - à travers `CatalogueDéchets` (voir Classe `CatalogueDéchets`), associer le symbole de la `Catégorie` à la valeur énumérée correspondante; puis
 - créer autant d'instances de `Déchet` que requis (voir `NbDéchetsParCat` dans la `Catégorie`) avec une position unique pour chaque déchet à créer (truc : faites une `List<Déchet>`, puis pour chaque déchet à construire pigez une position parmi les positions libres et retirez cette position de la liste des positions libres pour éviter de la piger deux fois)
 - quand tous les déchets sont construits, ajoutez-les à la `Surface` (cette classe offre justement une méthode `Ajouter` pour cela, méthode qui accepte en paramètre un tableau et nous savons comment convertir une `List<Déchet>` en `Déchet[]` n'est-ce pas?)
- Une méthode `CréerRobot`, acceptant en paramètre un `InfoRobot` et une `Surface`. Cette méthode doit :
 - trouver toutes les cases libres sur le `Surface` (en excluant son `Cadre`)
 - s'il n'y a plus de cases libres, lever `SurfacePleineException`
 - à une position libre prise au hasard, instanciez un `Robot` dont le nom est pris du `InfoRobot` reçu en paramètre et à la position choisie, puis
 - équipez (méthode `Équiper` du `Robot`) ce `Robot` d'un `Détecteur`. La source du détecteur sera le robot, le détecteur aura un rayon de 1 et la catégorie à détecter sera celle indiquée par le `InfoRobot` reçu en paramètre, pour enfin
 - ajouter le `Robot` à la `Surface` (cette classe offre justement une méthode `Ajouter` pour cela) et retourner le `Robot`

Note : pour trouver les cases libres sur une Surface nommée surf (en excluant son cadre), utilisez la méthode `TrouverSi` de surf comme suit :

```
List<Point2D> libres = surf.TrouverSi
(
    c => c == default || c == ' ',
    surf.Cadre.Exclure
);
```

Ne vous en faites pas, cette syntaxe sera expliquée en temps et lieu.

Interface IDéTECTEUR

L'interface `IDéTECTEUR` exprime le contrat minimal pour les opérations d'un détecteur (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 une `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

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 (votre chic prof vous montrera comment faire ceci). 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

Note : pour écrire un caractère à un point choisi sur la Console, utilisez la méthode `Console.SetCursorPosition` qui accepte en paramètre une position `x,y` où le 0,0 est en haut et à gauche de l'écran, et les coordonnées sont toutes positives.

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`.

Classe Robot

La classe Robot implémente IIconifiable et expose en plus :

- Une propriété d'instance immuable publique `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). Levez `NomInvalidException` le cas échéant
- Une propriété calculée publique `Symbole` correspondant à l'élément à l'indice zéro du `Nom` du Robot
- Une propriété privée immuable `Puissance` de type `Stack<float>`
- Un constructeur paramétrique acceptant en paramètre un nom et une position
- 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.
- Une méthode d'instance `Détecter` acceptant en paramètre un `IProjetable` et une `Catégorie`, et retournant une `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, la méthode 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). Utilisez la pile `Puissance` pour ce faire
- 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 et une `Surface` qui est présumée être celle où se trouve le Robot. 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 (note : utilisez les méthodes `Retirer` et `Ajouter` de la `Surface` pour, respectivement, retirer le Robot de son ancienne position et le placer à sa nouvelle position. À la suite d'un déplacement, réinitialisez la puissance du Robot (méthode `RéinitialiserPuissance` ci-dessus))

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.

Services de la .DLL fournie pour ce travail (TP01a-Wallyd-Affichage)

Vos chics profs vous offrent une .DLL nommée TP01a-Wallyd-Affichage prenant en charge un ensemble de services destinés à vous simplifier l'existence. **Vous n'avez pas à coder ce qui suit, mais vous pourrez (dans bien des cas, vous devrez) l'utiliser.**

Parmi les classes et autres mécanismes de cette .DLL, voici ce qui pourrait vous intéresser.

Classe Cadre

La classe immuable `Cadre` implémente `IProjetable` et expose entre autres 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`

Les autres services de cette classe ne sont pas pertinents pour ce travail pratique.

Classe Case

Certains services retournent ou acceptent en paramètre des instances de `Case`. Vous n'aurez pas à interagir avec cette classe pour le moment, mais l'important est qu'elle combine un symbole et deux couleurs (la couleur « avant » pour le texte et la couleur « arrière » pour le fond).

Classe CatalogueCouleurs

La classe `CatalogueCouleurs` est un singleton offrant deux services clés :

- `Associer`, qui accepte en paramètre un `char` et un `ConsoleColor` et associe le symbole dénoté par le `char` à cette couleur
- `Obtenir`, qui accepte en paramètre un `char` et un `ConsoleColor`. Si le symbole dénoté par le `char` passé en paramètre est connu du catalogue, alors la couleur qui lui est associée est retournée par cette fonction, sinon la couleur passée en paramètre (une sorte de « plan B ») est retournée

Les autres services de cette classe ne sont pas pertinents pour ce travail pratique.

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

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 *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 **Note** : un indexeur est une propriété un peu spéciale qui permet d'utiliser les crochets `[ et ]` sur un objet comme dans le cas d'une `List` ou d'un dictionnaire.

- Classe `Mutable`)

Note : un indexeur est une propriété un peu spéciale qui permet d'utiliser les crochets `[ et ]` sur un objet comme dans le cas d'une `List` ou d'un dictionnaire.

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

Note : ceci vous est offert à titre informatif car le code du programme principal qui interagit avec la classe `Mutable` vous est fourni.

Classe PipelineAffichage

Cette classe est importante pour l’affichage de ce qui apparaît à la console pour ce programme, mais le code qui interagit avec elle vous est fourni. Mentionnons pour le moment que le pipeline est constructible, qu’il est possible de lui ajouter des transformations (méthode `Ajouter`) et qu’il est possible que lui appliquer des effets (méthode `Appliquer`).

Classe Point2D

Le type `Point2D` est une classe immuable modélisant une coordonnée 2D (`X` et `Y`) offrant entre autres les services suivants :

- 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 $\sqrt{(p_0.X - p_1.X)^2 + (p_0.Y - p_1.Y)^2}$

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’accessor 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’accessor 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
- Deux méthodes d'instance nommées `TrouverSi(exclure,pred)` et `TrouverSi(pred)` qui utilisent des techniques qui n'ont pas encore été couvertes dans le cours (mais qui le seront, ne vous en faites pas!).

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 : 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 EstPlusGrandQue(Carré autre) => Aire > autre.Aire;
    // ...
}
```

Une **propriété calculée** (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.