420KBB – TP02b**Table des matières**

420KBB – TP02b	2
<i>Forme du travail</i>	2
Modalités de remise	2
<i>Programme principal qui vous est imposé</i>	3
Dans le détail	7
Classe CanalComm<T>	7
Classe PipelineAffichage	7
Classe Robot	7
Classe Détecteur	8
Classe Catalogue	8
Programme principal	8
Annexe – Rappels de vocabulaire	9

420KBB – TP02b

La fin de session approche, alors nous vous proposons une petite activité pour améliorer votre note (et idéalement votre bien-être... ou encore votre bien-être si vous favorisez la nouvelle orthographe) et mettre en pratique les fonctions asynchrones.

Forme du travail

Votre travail sera une modification au TP02 et 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 6 décembre 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)
Impact :	Si ce travail est correctement fait, la note de votre plus faible travail pratique de la session sera bonifiée de 20 points de pourcentage (20% si le travail était noté sur 100)

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!) :

```
// ...
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),
    ('*', Catégorie.Recyclable),
};

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 = 10;
List<Déchet> déchets =
    CréerPollution(symCatégorie, dé, NB_DÉCHETS, surf, cadre.Exclure);
surf.Ajouter(déchets.ToArray());

CanalComm<InfoCanalComm> canal = new(robotsInfo.Length);
PipelineAffichage pipe = new(surf, cadre, canal);
```

```
CancellationTokenSource cts0 = new();
CancellationTokenSource cts1 = new();

Task collectes = Task.WhenAny
(
    ExécuterCollecteAsync
    (
        robotsInfo, surf, canal, messagerie, information, cts0.Token, cts1
    ), ArrêterAsync(cts0)
);

while (!cts1.Token.IsCancellationRequested)
{
    pipe.Rafraichir();
    messagerie.Afficher();
    information.Afficher();
}
pipe.Rafraichir();
messagerie.Afficher();
information.Afficher();

// Ici la méthode ExécuterCollecteAsync

// Ici la méthode ArrêterAsync

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));
}
```

```
    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;
}
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 des déchets de différentes catégories sur la surface (vous pouvez jouer avec la quantité de déchets, dans la mesure du raisonnable)
- On crée un `CanalComm` sur lequel les robots publieront les informations nécessaires au `PipelineAffichage` pour bien rafraichir l’écran
- 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
- Il est possible de mettre fin à la simulation en pressant n’importe quel toucher touche

Le travail du TP02b consiste à modifier le programme du TP02 pour remplacer les fils d’exécution (les instances de `Thread`) par la programmation asynchrone basée sur le type `Task` et les opérateurs `async` et `await`.

Dans le détail...

La grande majorité des classes du TP02 demeure inchangée, sauf pour les classes suivantes qui devront être retouchées pour respecter les exigences du programme principales et ajouter les mécanismes de programmation asynchrone.

Classe *CanalComm*<T>

La classe `CanalComm` est modifiée pour devenir générique. Vous devez faire toutes les modifications nécessaires.

Classe *PipelineAffichage*

La classe `PipelineAffichage` subit une cure de minceur. Voici, grosso modo en quoi consiste les modifications :

- L’affichage ne se fait plus dans un fil secondaire mais plutôt dans le fil principal au moment d’appeler la méthode `Rafraichir`
- Un `PipelineAffichage` n’est plus responsable d’afficher les deux messageries
- La méthode publique `Rafraichir` devient responsable de rafraichir l’écran avec les changements obtenus du `CanalComm`, ce qui implique :
 - Appeler `Balayer` sur le `CanalComm` du `PipelineAffichage` pour obtenir l’information décrivant les halos des divers cueilleurs
 - Ensuite, s’il y a au moins un halo à afficher, dupliquer la `Surface`, puis cumuler des appels à `GénétrerHalo` sur cette surface en prenant soin de conserver ce cumul
 - Enfin, appliquer `Appliquer` sur la surface résultante

Classe *Robot*

La classe `Robot` offre maintenant deux méthodes asynchrones permettant aux instances de cette classe de prendre en charge la collecte des déchets pour lesquels ils sont compétents.

- Une méthode asynchrone d’instance privée `RechercherDéchetsAsync` acceptant en paramètre un `CanalComm`, une `Surface` et un `noPort`. La méthode retourne une `Task` qui contiendra éventuellement une `List<Point2D>` non-vide représentant la position des déchets trouvés.
 - Précondition : il existe au moins un déchet sur la surface que le détecteur peut détecter.
 - La méthode procède en incrémentant systématiquement le rayon d’action du détecteur tant que ce dernier ne retourne pas une `List<Point2D>` non-vide.
- Une méthode asynchrone d’instance publique `CueillirDéchetsAsync` acceptant en paramètre un `CanalComm`, un `noPort`, une `Surface`, un `CancellationToken` et deux `Message`. La méthode retourne une `Task` (rappel : `Task` tout court signifie, conceptuellement, `Task<void>`).
 - Cette méthode est responsable de ramasser tous les déchets que le détecteur du robot peut détecter.
 - Le ramassage des déchets pourrait être interrompu par l’appelant à travers le `CancellationToken` passé en paramètre.

- Cette méthode délègue à la méthode `RechercherDéchetsAsync` la recherche des déchets.
- Une fois la position des déchets obtenue, `CueillirDéchetsAsync` déplace le robot vers le premier déchet de la liste. Si la position du robot coïncide avec la position du déchet, on ramasse le déchet, sinon on poursuit la recherche. Dans les deux cas, on met à jour les messageries.

Classe Détecteur

La classe `Détecteur` implémente `IDétecteur` et se voit doté de la méthode d'instance `PeutDétecter` qui accepte en paramètre un `symbole`. Cette méthode est un prédicat qui retourne `true` seulement si le `symbole` correspond à un déchet que le détecteur peut détecter.

Classe Catalogue

La classe `Catalogue` est un singleton entreposant des associations, chacune se faisant entre un `symbole` (de type `char`) et une `Catégorie`. Cette classe peut maintenant aussi comptabiliser les déchets ramassés (à vous de déterminer comment, et de vous assurer que ce soit fait sans provoquer de *Data Race*).

Programme principal

Vous remarquerez que, dans le programme principal, il reste deux méthodes à écrire, soit `ExécuterCollecteAsync` et `ArrêterAsync`.

Sans grande surprise, l'une exécute la collecte de déchets et l'autre s'assure de l'arrêt de cette collecte. On s'attend à ce que ces fonctions s'intègrent bien au reste du programme principal, mais nous ne vous donnons pas plus d'indications, pour vous donner l'occasion de réfléchir et de figurer les principaux enjeux de ce problème.

Sentez-vous tout à fait à l'aise de discuter avec votre chic prof pour voir si vos hypothèses tiennent la route : verbaliser nos idées aide à leur donner forme!

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.