# 420KBB – TP01b

## *Table des matières*

# 420KBB – TP01b

Votre travail au `TP01a` a porté fruit et vous avez désormais une plateforme opérationnelle de simulation du comportement d'un robot de la série `WAL-1D`. Il semble y avoir de l'espoir pour le futur environnemental, et peut-être même pour l'humanité!

La prochaine tâche attribuée à votre équipe et à vous-même (étape `TP01b`) est d'apporter un certain nombre de raffinements au logiciel existant. On vous explique que cette courte étape de développement constitue un jalon en vue d'une prochaine mise à jour majeure impliquant plusieurs robots. Le nom de code de cette future mise à jour sera, vous dit-on, `TP02`; ça semble excitant!

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. 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 24 octobre 2025 à 23 h 59** |
| **Code source imprimé?** | Non, pas cette fois |
| **Remise par Colnet?** | Oui, dans une archive zip nommée[1] comme suit :<br>`Groupe-NomPrénom-TP01b.zip` (p. ex. : `05-TromblonGaetan-TP01b.zip`)<br>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) |

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

## *Remarques générales*

**Note** : vous aurez un nouveau programme principal pour `TP01b`. Le code suit, mais prenez-le du site du cours (ce sera plus facile de le copier / coller du site que d'un `PDF`) :

```
ConfigInfo config = Config.LireConfig("../../../config_tp1.json");
// préparer la surface d'affichage
Surface surf = FabriqueSurface.Créer(config);
// préparer la zone de messagerie
Messagerie messagerie = new(new(surf.Largeur, 0));
// positionner Wallyd (on se limite à lui pour le moment)
List<Robot> robots = FabriqueSurface.CréerRobots(config.Robot, surf);
Robot wallyd = robots[0];
PipelineAffichage pipeline = new(surf);
// tant qu'il reste des déchets à ramasser
while (surf.TrouverSi(c => wallyd.PeutDétecter(c)).Count > 0)
{
   bool trouvé = false;
   // Trouver et ramasser le déchet
   do
   {
      // 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,
                     CatalogueCouleurs.Get.ObtenirCouleur(wallyd.Symbole,
                                                          ConsoleColor.Green),
                     surf.Dupliquer())
      );
      var pts = wallyd.Détecter(surf, Catégorie.Métal);
      if (pts.Count > 0)
      {
         trouvé = true;
         Point2D nouvellePosi = wallyd.CalculerDéplacementVers(pts[0]);
         if (pts[0] == nouvellePosi)
         {
            messagerie.Effacer();
            messagerie.Afficher
            (
               $"Déchet collecté à la position {pts[0]}"
            );
            surf.Retirer(pts[0]);
         }
         else
         {
            if (surf[nouvellePosi].EstVide())
               messagerie.Afficher
```

```
                (
                   $"Trouvé {pts.Count} déchet(s)",
                   $"Déplacement vers {nouvellePosi}"
                );
             else
             {
                nouvellePosi = wallyd.TrouverPassage(surf);
                messagerie.Effacer();
                messagerie.Afficher
                   (
                     $"Déplacement impossible vers {pts[0]}",
                     $"Contournement vers {nouvellePosi}"
                   );
             }
          }
          wallyd.DéplacerVers(nouvellePosi, surf);
       }
       else
       {
          wallyd.AugmenterPuissance();
          //messagerie.Effacer();
       }
       Thread.Sleep(500);
    }
    while (!trouvé);
    wallyd.RéinitialiserPuissance();
}
//
// dernier affichage une fois les déchets collectés
//
{
   pipeline.Appliquer
   (
       GénérerHalo(wallyd.Zone,
                   CatalogueCouleurs.Get.ObtenirCouleur(wallyd.Symbole, ConsoleColor.Green),
                   surf.Dupliquer())
   );
}
```

Portez attention aux changements suivants, en caractères gras ci-dessus :

- L'association d'une couleur et d'un symbole à un robot a été déplacée (elle se trouve maintenant dans `Config.CréerRobots`, voir Classe `Config` et classes associées pour des détails)
- La `Messagerie` a été déplacée : plutôt que d'être sous la surface d'affichage, elle se trouve désormais à sa droite
- Le `PipelineAffichage` est désormais créé hors de la boucle, et l'ajout du `Cadre` de la `Surface` est fait dans son constructeur (voir Classe `PipelineAffichage`)

- Le test pour savoir s'il reste des déchets à ramasser est désormais sous la responsabilité de notre robot.
- La génération du halo passe désormais en partie par les services du `CatalogueCouleurs`
- La mécanique de déplacement a été modifiée (plus de détails dans Classe `Robot`)

**Note** : vous aurez une nouvelle `DLL` d'affichage pour `TP01b`, et cette `DLL` sera différente à certains égards de celle fournie pour `TP01a`. Ne vous trompez pas! Prenez la nouvelle `DLL` d'affichage du site du cours.

**Note** : vous aurez un nouveau fichier `JSON` pour `TP01b`, et un exemple simplifié est donné dans Classe `Config` et classes associées. Prenez le nouveau fichier `JSON` du site du cours,

### *Tâches à réaliser*

Les tâches suivantes sont celles qui doivent être réalisées dans le cadre de ce petit travail pratique.

#### *Classe statique `AlgosWAL1D`*

Ajoutez une classe statique `AlgosWAL1D`.

Déplacez-y la fonction `GénérerHalo`. Assurez-vous que le code compile et fonctionne encore une fois ce changement apporté (ajoutez des `using static` aux endroits opportuns si nécessaire).

Ajoutez-y une méthode d'extension `EstDans` pour `Surface`. Ce prédicat doit accepter en paramètre un `Point2D` et retourner `true` seulement si ce `Point2D` est dans la `Surface` (excluez les bordures de la `Surface` pour les besoins du calcul, celles-ci étant réservées pour son `Cadre`).

#### *Énumération `Catégorie`*

De nouvelles catégories de déchets s'ajoutent au système. Ajoutez les catégories `Plastique`, `Organique` et `Nucléaire` à l'énumération `Catégorie`.

#### *Classe `Config` et classes associées*

Nous travaillerons à partir d'un nouveau fichier `JSON` ayant la forme suivante :

```
{
  "Surface": {
    "Hauteur": 10,
    "Largeur": 20
  },
  "Robot": [
    {
      "Nom": "Wallyd",
      "Symbole": "W",
      "Couleur": "Magenta",
      "Catégorie": "Métal"
    },
    {
```

```
      "Nom": "Bart",
      "Symbole": "B",
      "Couleur": "Blue",
      "Catégorie": "Plastique"
    }
  ],
  "CatégorieDéchet": [
    {
      "NbDéchetsParCat": 5,
      "Symbole": "%",
      "Catégorie": "Métal"
    },
    {
      "NbDéchetsParCat": 10,
      "Symbole": "*",
      "Catégorie": "Métal"
    }
  ]
}
```

Comme pour le `TP01a`, vous pourrez faire des modifications créatives à certains aspects de ce fichier de configuration (modifier le nombre de déchets de chaque catégorie, les noms de robots, les dimensions de la surface, etc.) dans la mesure où vous respectez l'esprit du travail pratique.

Les changements principaux que vous remarquerez sont :

- Il peut y avoir plusieurs robots (ceci deviendra très important au `TP02`).
- Il peut y avoir plusieurs catégories de déchets.
- Il faut donc que `ConfigInfo` soit ajustée pour contenir une `List<InfoRobot>` de même qu'une `List<CatégorieDéchet>`.

### Classe *Détecteur*

Adaptez cette classe pour tenir compte des changements apportés à l'interface `IDétecteur` (voir Interface `IDétecteur`)

### Classe *FabriqueSurface*

Cette classe subira quelques changements :

- Ajoutez une méthode `InitialiserPollution` acceptant en paramètre une `List<CatégorieDéchet>` et une `Surface`. Cette fonction devra (a) créer une `List<Déchet>`, (b) trouver les cases libres sur la surface, puis (c) pour chaque `CatégorieDéchet`, appellera `InitialiserDéchets` pour qu'elle ajoute les déchets en question dans la `List<Déchet>`. Une fois tous les déchets créés, cette fonction les ajoutera en bloc à la `Surface`.
- Modifiez la méthode `InitialiserDéchets` pour qu'elle accepte en paramètre une `CatégorieDéchet`, une `List<Déchet>` et une `List<Point2D>` indiquant les cases libres au moment de l'appel. Cette fonction devra lever une exception si le nombre de cases

libres est insuffisant pour loger tous les déchets de cette catégorie, puis placera chaque déchet à une case libre distincte de celles des autres déchets tout en prenant soin de retirer la case utilisée des cases libres.

- Ajoutez une méthode `CréerRobots` acceptant en paramètre une `List<InfoRobot>` et une `Surface`. Cette fonction devra (a) trouver les cases libres sur la surface, levant `SurfacePleineException` si le nombre de cases vides est insuffisant pour loger tous les robots, puis (b) créera chaque robot en le plaçant à un endroit libre sur la surface, (c) équipera chaque robot d'un détecteur approprié pour le type de déchets qu'il doit détecter, et (d) s'assurera que le `CatalogueCouleurs` sache quelle couleur est associée au robot. Cette fonction retournera une `List<Robot>` contenant chaque `Robot` créé.

### *Interface `IDétecteur`*

Enrichissez le contrat décrit par cette interface en ajoutant une méthode `PeutDétecter(char)` (rappel : l'interface exposait déjà `PeutDétecter(Catégorie)`). L'intention est de simplifier le code client en lui permettant de passer un symbole en paramètre et de laisser les détecteurs faire le lien avec la catégorie correspondante par eux-mêmes.

### *Classe `PipelineAffichage`*

Il a été décidé que cette classe quitterait la `DLL` d'affichage et serait désormais logée dans le code client. L'implémentation de cette classe va comme suit :

```
class PipelineAffichage
{
   List<Func<Mutable, Mutable>> Transfos { get; } = new();
   // NOTE : IL MANQUE UN CONSTRUCTEUR ICI
   public void Ajouter(Func<Mutable, Mutable> transfo) =>
      Transfos.Add(transfo);
   public IProjetable Appliquer(Point2D pos, Mutable p)
   {
      foreach (var transfo in Transfos)
         p = transfo(p);
      return Afficher(pos, p);
   }
   public IProjetable Appliquer(Mutable p) =>
      Appliquer(new(), p);
   public IProjetable Afficher(Point2D pos, Mutable p)
   {
      for (int ligne = 0; ligne != p.Hauteur; ++ligne)
         for (int col = 0; col != p.Largeur; ++col)
         {
            Console.SetCursorPosition(col + pos.X, ligne + pos.Y);
            var préF = Console.ForegroundColor;
            var préB = Console.BackgroundColor;
            Console.ForegroundColor = p[ligne, col].Avant;
            Console.BackgroundColor = p[ligne, col].Arrière;
            Console.Write
```

```
            (
               p[ligne, col].Symbole == default ?
                  ' ' : p[ligne, col].Symbole
            );
            Console.BackgroundColor = préB;
            Console.ForegroundColor = préF;
         }
      return p;
   }
   // retourne une fonction qui transforme un mutable pour
   // y ajouter un cadre
   public Func<Mutable, Mutable> Appliquer(Cadre p)
   {
      return m =>
      {
         Mutable res = m.Dupliquer();
         for (int i = 0; i != p.Hauteur; ++i)
            for (int j = 0; j != p.Largeur; ++j)
            {
               var c = p[i, j];
               if (c.Symbole != default)
                  res[i, j] = c;
            }
         return res;
      };
   }
}
```

On vous indique qu'il manque présentement à cette classe un constructeur de `Pipeline` acceptant une `Surface` en paramètre. Ce constructeur doit ajouter aux transformations du `Pipeline` le résultat de l'application de la fonction `Appliquer` sur le `Cadre` de la surface. On vous fait confiance pour trouver le bon moyen d'y arriver.

## *Classe `Robot`*

Ajoutez à `Robot` une méthode `PeutDétecter(char)` et faites en sorte que cette méthode délègue le travail au `Détecteur` installé sur le `Robot`.

Modifiez la méthode `DéplacerVers` pour qu'elle prenne en paramètre la nouvelle position du `Robot` de même qu'une `Surface`, et qu'elle se limite à (a) retirer le `Robot` de sa position actuelle, (b) déplacer le `Robot` à sa nouvelle position, (c) ajouter le `Robot` à sa nouvelle position et (d) réinitialiser la puissance du `Robot`.

Ajoutez la méthode `TrouverPassage` acceptant en paramètre une `Surface` et retournant un `Point2D`. Cette méthode doit :

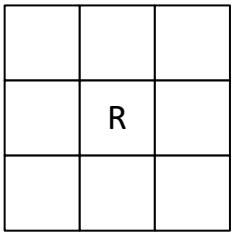
- Faire une `List<Point2D>` de tous les points immédiatement avoisinant au `Robot` (il y en a huit au maximum) et qui (a) sont dans la `Surface` et (b) sont vides.

- Par la suite, si cette liste est vide, la fonction doit retourner un point équivalent à la position actuelle du `Robot` (pas de place pour bouger, on reste sur place!), sinon elle doit retourner une position choisie de manière pseudoaléatoire dans la liste.

Ajoutez la méthode `CalculerDéplacementVers` acceptant en paramètre un `Point2D` représentant le point de destination (qui peut être éloigné), et retournant un `Point2D` dont le rôle sera de retourner le point immédiatement avoisinant du `Robot` qui le rapprochera de sa destination. Évidemment, si le `Robot` est déjà à destination, retournez sa position actuelle.

**Note** : les points immédiatement avoisinants d'un `Robot` sont visibles à droite. Le `Robot` représenté par la case `R` est au centre du voisinage.

Les directions possibles sont les cases voisines (Est, Nord-Est, Nord, Nord-Ouest, Ouest, Sud-Ouest, Sud, Sud-Est)




### *Ajustements apportés à la `DLL` d'affichage*

Ce qui suit liste sommairement les ajustements apportés à la `DLL` d'affichage. Vous pouvez vous référer à la description donnée au `TP01a` pour plus d'informations :

- La classe `PipelineAffichage` et déplacée dans le code client (voir plus haut).
- La classe `Case` expose désormais un prédicat `EstVide`
- La classe `Point2D` implémente `IEquatable<Point2D>` et expose désormais aussi :
  - `operator==(Point2D, Point2D)`
  - `operator!=(Point2D, Point2D)`
  - `operator+(Point2D, Point2D)` retournant un `Point2D`

### *Avant de vous mettre au boulot…*

Ce travail est beaucoup plus petit que le précédent, mais nous continuons à vous donner de plus en plus de liberté. Comme à l'habitude :

- 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.