420KBB – TP02

Table des matières

420KBB – TP02

Après des mois de travail sur le simulateur de collecte automatisée de déchets à travers des robots de la série `WAL-1D`, le temps est venu de déployer votre équipe de robots et de collecter ces déchets qui polluent l'environnement sur les diverses planètes visitées par l'humanité. Enfin!

Pour cette version officielle du programme de nettoyage, il vous faudra toutefois résoudre des problèmes plus sophistiqués que par le passé, soit :

- Assurer le mouvement concurrent de multiples robots équipés de détecteurs diversifiés et collectant des déchets d'une variété de catégories
- Contourner les obstacles lors des mouvements des robots
- Conserver une trace écrite plus élaborée des actions posées par les robots

La rumeur veut que les industries polluantes (oui, il en reste!) souhaitent que leurs déchets demeurent jonchés ici et là sur la surface des planètes à nettoyer. Les robots de la série `WAL-1D` ne sont pas des unités de combat, et sont donc vulnérables aux bombes placées sur leur route. Il se peut que certains robots ne survivent pas aux opérations, mais croisons nos doigts et espérons que la plupart d'entre aux parviennent à mener à bien leur mission!

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. Il est construit sur les fondations des `TP01a` et `TP01b`, mais comprend de nombreuses modifications.

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 21 novembre 2025 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)

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

Remarques générales

Note : vous aurez un nouveau programme principal pour `TP02`. Le code suit, mais prenez-le du site du cours (ce sera plus facile de le copier / coller du site que d'un `PDF`) :

```
// ... using ...
ConfigInfo config = Config.LireConfig("../../../config_tp2.json");

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

// préparer la zone de messagerie
Messagerie messagerie = new(new(surf.Largeur, 0), surf.Hauteur);
// préparer la zone d'informations
Messagerie information = new(new(0, surf.Hauteur), 2);

List<Robot> robots = FabriqueSurface.CréerRobots(config.Robot, surf);
CancellationTokenSource src = new();
Thread saboteur = CréerSaboteur(robots, surf, src.Token);
saboteur.Start();

CanalComm canal = new(robots.Count);
Afficheur pipeline = new(surf, canal, messagerie, information);

pipeline.Démarrer();
Thread[] threads = CréerFils
(
   robots, surf, canal, messagerie, information, src.Token
);

Thread lireTouche = new(() =>
{
   Console.ReadKey(true);
   src.Cancel();
});
lireTouche.Start();
foreach (var c in threads) c.Start();
foreach (var c in threads) c.Join();
lireTouche.Join();
pipeline.Arrêter();
```

Le programme est simple, en fait :

- Il consomme le fichier de configuration comme dans les versions précédentes
- Il crée deux messageries, chacune ayant son propre rôle
- Il crée les robots, comme précédemment
- Il crée un `Saboteur` (voir plus bas) voué à nuire à nos robots dans l'exercice de leurs tâches, et le met immédiatement en marche

- Il crée un `CanalComm` (voir plus bas) et un `Afficheur` (voir plus bas), démarrant ce dernier
- Il crée un fil d'exécution par robot, à travers `CréerFils`, car chaque robot fera son travail indépendamment des autres
- Il crée un fil d'exécution responsable de lire une touche pour (éventuellement) terminer le programme
- Il lance tous les fils qui ne sont pas encore démarrés, puis
- Il attend la fin de tous ces fils (cette fin ne surviendra qu'une fois une touche pressée

Pour `CréerFils`, le code imposé est :

```
static Thread[] CréerFils
(
   List<Robot> robots, Surface surf, CanalComm canal,
   Messagerie messagerie, Messagerie information, CancellationToken jeton
)
{
   Dictionary<char, int> collectés = new();
   Thread[] fils = new Thread[robots.Count];
   for (int r = 0; r < robots.Count; r++)
   {
      var robot = robots[r];
      int portNo = r;
      fils[r] = new(() =>
      {
         bool décédé = false;
         // tant qu'il reste des déchets à ramasser
         while (!décédé && !jeton.IsCancellationRequested &&
               surf.TrouverSi(c => robot.PeutDétecter(c)).Count > 0)
         {
            bool trouvé = false;
            // Trouver et ramasser le déchet
            do
            {
               canal.PublierSur(
                  portNo,
                  new(robot.Zone,
                      CatalogueCouleurs.Get.ObtenirCouleur(robot.Symbole,
                                                           ConsoleColor.Green),
                      robot.Symbole));
               var pts = robot.Détecter(surf);
               if (pts.Count > 0)
               {
                  trouvé = true;
                  Point2D nouvellePosi = robot.CalculerDéplacementVers(pts[0]);
                  if (pts[0] == nouvellePosi)
                  {
                     char c = surf[pts[0]].Symbole;
```

```
            lock (collectés)
               if (collectés.ContainsKey(c))
                  collectés[c]++;
               else
                  collectés.Add(c, 1);
            messagerie.Ajouter
            (
               CatalogueCouleurs.Get.ObtenirCouleur(robot.Symbole,
                                                    ConsoleColor.Green),
               $"Déchet collecté à la position {nouvellePosi}"
            );
            surf.Retirer(pts[0]);
         }
         else
         {
            if (surf[nouvellePosi].EstVide)
               messagerie.Ajouter
               (
                  CatalogueCouleurs.Get.ObtenirCouleur(robot.Symbole,
                                                       ConsoleColor.Green),
                  $"Trouvé {pts.Count} déchet(s)",
                  $"Déplacement vers {pts[0]}"
               );
            else
            {
               nouvellePosi = robot.TrouverPassage(surf);
               messagerie.Ajouter
               (
                 CatalogueCouleurs.Get.ObtenirCouleur(robot.Symbole,
                                                      ConsoleColor.Green),
                 $"Déplacement impossible vers {pts[0]}",
                 $"Contournement par {nouvellePosi}"
               );
            }
            Thread.Sleep(100);
         }
         try
         {
            robot.DéplacerVers(nouvellePosi, surf);
         }
         catch(DéplacementFatalException ex)
         {
            messagerie.Ajouter(ConsoleColor.White, $"{robot.Nom} : {ex.Message}");
            décédé = true;
         }
         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(400);
           }
           while (!trouvé && !décédé);
           robot.RéinitialiserPuissance();
        }
        if (!décédé && surf.TrouverSi(c => robot.PeutDétecter(c)).Count == 0)
           messagerie.Ajouter(ConsoleColor.White,
                              $"{robot.Nom} : nettoyage complété, mise au repos");
     });
  }
  return fils;
}
```

Pour `CréerSaboteur`, le code imposé est :

```
static Thread CréerSaboteur
   (List<Robot> robots, Surface surf, CancellationToken jeton)
{
   Thread fils = new(() =>
   {
      Random rnd = new();
      bool terminé = false;
      Bombe[] zeBombes;
      List<char> symRobots = new();
      foreach (var r in robots)
         symRobots.Add(r.Symbole);
      while (!terminé)
      {
         lock (surf)
         {
            // Sélectionner un robot au hasard
            int idxRobot = rnd.Next(robots.Count);
            var posiLibres = surf.TrouverSi
            (
               c => c == default || c == ' ',
               surf.Cadre.Exclure
            );
            List<Bombe> bombes = new();
```

```
            foreach ( var pl in posiLibres)
               if (pl.Distance(robots[idxRobot].Pos) < 2)
                  bombes.Add(new('.', pl));
            zeBombes = bombes.ToArray();
            surf.Ajouter(zeBombes);
         }
         // Laisser les bombes un temps aléatoire
         Thread.Sleep(rnd.Next(100, 201));
         lock (surf)
         {
            surf.Retirer(zeBombes);
         }
         if (jeton.IsCancellationRequested)
            terminé = true;
         else
            Thread.Sleep(2000); // Répéter à toutes les 2 secondes
      }
   });
   return fils;
}
```

Prenez soin de lire et d'analyser le tout, et de poser des questions à votre chic prof!

Tâches à réaliser

Les tâches suivantes sont celles qui doivent être réalisées dans le cadre de ce travail pratique. Nous ne poserons pas notre regard sur le code fait dans les travaux pratiques précédents et qui ne change pas, préférant nous concentrer sur les modifications à apporter à l'existant et sur les nouveaux ajouts (classes, autres types, algorithmes, etc.).

Classe `Afficheur`

Une nouvelle classe nommée `Afficheur` prendra en charge le volet « affichage » de la surface sur laquelle se déplacent les robots. Ceci tiendra compte du fait que ce programme sera beaucoup plus dynamique que ses prédécesseurs, et permettra de déterminer une fréquence de rafraîchissement de l'affichage qui soit indépendante de la vitesse de déplacement des robots.

Les services à implémenter sont :

- Une propriété privée `Pipeline` de type `PipelineAffichage`, qui sera instanciée seulement une fois, à la construction
- Une propriété privée `Surface` de type `Surface`, qui sera instanciée seulement une fois, à la construction
- Un attribut privé `terminé` de type `int`, initialisé à zéro. **Ceci est important** : cette variable servira à titre de « booléen » (`0` pour faux, `1` pour vrai) et sera manipulée à partir de fonctions spécialisées permettant de modifier ou de tester un `int` de manière synchronisée
- Une propriété privée `FilAfficheur` de type `Thread`, qui sera instanciée seulement une fois, à la construction
- Une propriété privée `CanalComm` de type `CanalComm`, qui sera instanciée seulement une fois, à la construction
- Une propriété privée `Mess` de type `Messagerie`, qui sera instanciée seulement une fois, à la construction
- Une propriété privée `Info` de type `Messagerie`, qui sera instanciée seulement une fois, à la construction
- Un constructeur paramétrique privé qui acceptera une `Surface` en paramètre, et utilisera ce paramètre pour initialiser la propriété `Surface` et la propriété `Pipeline` (rappel : un `PipelineAffichage` est initialisé à l'aide d'une `Surface`)
- Un constructeur paramétrique public qui acceptera en paramètre une `Surface`, un `CanalComm` de même que deux `Messagerie` (`mess` et `info`, dans l'ordre) et initialisera les propriétés correspondantes. Ce constructeur instanciera aussi `FilAfficheur` en lui passant en paramètre la méthode `Exécuter` (plus bas dans la présente classe)
- Une méthode publique `Appliquer` acceptant en paramètre un `Mutable` et retournant un `IProjetable`, dont le rôle est d'appeler `Appliquer` au `Pipeline` en lui passant un `Point2D` par défaut et le `Mutable` en question
- Une méthode publique `Démarrer` qui ne prend pas de paramètre et ne retourne rien. Cette méthode doit modifier `terminé` de manière synchronisée (utilisez `Interlocked.Echange` pour ce faire) pour que cet attribut prenne la valeur `0`, puis démarrer `FilAfficheur`

- Une méthode publique `Arrêter` qui ne prend pas de paramètre et ne retourne rien. Cette méthode doit modifier `terminé` de manière synchronisée (utilisez `Interlocked.Echange` pour ce faire) pour que cet attribut prenne la valeur `1`, puis attendre la fin de l'exécution de `FilAfficheur`
- Une méthode privée `Exécuter` qui ne prend pas de paramètre et ne retourne rien. C'est cette méthode qui réalisera l'affichage à proprement dit. Son travail sera :
  - de boucler tant que l'exécution n'est pas terminée (tant que `terminé` sera différent de `1`), en dormant (`Thread.Sleep`) pour `500` ms après chaque itération
  - à chaque itération, `Balayer` le `CanalComm` pour collecter les informations les plus récentes
  - s'il y a au moins un élément dans la `List` retournée par le balayage, alors `Dupliquer` la `Surface`, puis appliquer `GénérerHalo` sur chaque élément de la `List` à l'aide de cette nouvelle surface pour que cette surface comprenne éventuellement tous les halos, pour enfin `Appliquer` cette surface. Par la suite, appeler `Afficher` sur `Mess` et `Info` respectivement
  - une fois que l'on aura constaté que `terminé` est devenu `1`, il faut faire un dernier affichage pour traiter les données résiduelles, comme c'est souvent le cas dans une telle situation

Classe statique `Algos`

Assurez-vous qu'en plus des autres outils qui s'y trouvent déjà, votre classe `Algos` contienne l'algorithme `Cumuler<T,U>` que nous avons fait en classe à titre d'exercice.

Assurez-vous qu'en plus des autres outils qui s'y trouvent déjà, votre classe `Algos` contienne l'algorithme `Permuter<T>` que nous avons fait en classe à titre d'exercice.

Classe statique `AlgosWAL1D`

Assurez-vous qu'en plus des autres outils qui s'y trouvent déjà, votre classe `AlgosWAL1D` contienne la méthode d'extension `Distance(this Point2D p0,Point2D p1)` calculant et retournant la distance euclidienne entre `p0` et `p1`.

Assurez-vous qu'en plus des autres outils qui s'y trouvent déjà, votre classe `AlgosWAL1D` contienne la méthode d'extension `Dupliquer(this Surface)` qui (a) verrouillera la `Surface` (utilisant l'objet de type `Surface` lui-même à titre de mutex), puis (b) retournera une copie de cette `Surface` en tant que `Mutable` en appelant le constructeur de `Mutable` prenant cette `Surface` (qui est aussi un `IProjetable`) en paramètre.

Classe `CanalComm`

Une nouvelle classe, `CanalComm`, modélisera un canal de communication synchronisé pour alimenter le pipeline d'affichage. Un `CanalComm` représentera un certain nombre de « ports » (des `int`), et à chaque « port » sera associée une zone de transit (des `ZoneTransit<T>`, voir plus bas). Il sera possible de publier sur un canal à l'aide de son « port », de lire d'un canal à l'aide de son « port », ou encore de balayer l'ensemble des canaux pour en collecter les informations les plus récentes.

Note : à travers cette classe, assurez-vous en tout temps que les accès à la propriété `Ports` soient synchronisés.

Les services à implémenter sont :

- Une propriété publique immuable `NbPorts` (un entier)
- Une propriété privée `Ports` qui sera un dictionnaire associant une clé entière (un numéro de port) à une valeur de type `ZoneTransit<PipelineInfo>` (voir plus bas pour ces deux types)
- Un constructeur public acceptant en paramètre un entier (le nombre de ports du `CanalComm`). Son rôle sera d'initialiser la propriété `NbPorts` et d'associer aux ports $0,1,2,\ldots,NbPorts-1$ des zones de transit nouvellement construites
- Une méthode publique `ObtenirPort` acceptant en paramètre un numéro de port et retournant la zone de transit qui lui est associée. Levez `PortInexistantException` si ce port n'existe pas
- Une méthode publique `PublierSur` acceptant en paramètre un numéro de port et un `PipelineInfo`, dont le rôle est d'ajouter le `PipelineInfo` à la zone de transit du port en question. Levez `PortInexistantException` si ce port n'existe pas
- Une méthode publique `LireSur` acceptant en paramètre un numéro de port, dont le rôle est d'extraire la `List<PipelineInfo>` de la zone de transit du port en question. Levez `PortInexistantException` si ce port n'existe pas
- Une méthode publique `Balayer` ne prenant pas de paramètre et retournant une `List<PipelineInfo>`. Cette méthode doit extraire la `List<PipelineInfo>` de la zone de transit associée à chaque port, et faire une `List<PipelineInfo>` contenant le dernier élément de chacune de ces `List` qui ne sera pas vide pour ensuite retourner cette `List`
- Une méthode publique `Vider` ne prenant pas de paramètre et vidant la zone de transit associée à chacun des ports du `CanalComm`

Énumération `Catégorie`

Une nouvelle catégorie de déchets s'ajoute au système. Ajoutez la catégorie `Explosif` à l'énumération `Catégorie`.

Classe `Messagerie`

La classe `Messagerie` vivra d'importants changements, pour tenir compte du fait que ce programme sera beaucoup plus dynamique que ses prédécesseurs.

Note : à travers cette classe, assurez-vous en tout temps que les accès à la propriété `Messages` soient synchronisés.

Les services à implémenter sont :

- Une propriété privée `Messages` dont le type sera une `List` d'uplets faits d'une `string` et d'un `ConsoleColor`
- Une propriété privée immuable `Pos` de type `Point2D` qui représentera la position à l'écran de la messagerie

- Une propriété privée entière mutable `Hauteur`
- Une propriété privée entière mutable `Largeur`
- Une propriété privée entière immuable `MaxHauteur` qui représentera la hauteur maximale de la messagerie
- Un constructeur paramétrique acceptant en paramètre la position de la messagerie et sa hauteur maximale, et initialisant les propriétés correspondantes
- Une méthode `Ajouter` acceptant en paramètre une `ConsoleColor` et un nombre arbitrairement grand de messages de type `string`. Cette méthode doit ajouter à `Messages` des paires faites de chaque `string` (dans l'ordre) et de la couleur en question, tout en s'assurant que le nombre d'éléments de `Messages` ne dépasse pas `MaxHauteur` (ne conservez que les `MaxHauteur` plus récents « messages colorés »)
- Une méthode `Afficher` ne prenant pas de paramètre et qui remplacera l'ancienne méthode du même nom. Cette méthode devra :
  - faire une copie locale (un *Snapshot*) de `Messages`
  - calculer la largeur de l'affichage. Cette largeur sera le maximum de la largeur actuelle et de la longueur de la plus longue `string` de ce *Snapshot* (vous **devez** utiliser `Algos.Cumuler` pour calculer cette dernière)
  - calculer la hauteur de l'affichage. Cette hauteur ne devra pas dépasser le nombre d'éléments du *Snapshot* et ne devra pas non plus dépasser `MaxHauteur`
  - enfin, afficher chaque string du Snapshot en s'assurant de (a) modifier la couleur de l'affichage pour respecter la couleur associée à cette `string` et (b) de remettre la couleur d'affichage initiale en place par la suite
- Une méthode `Effacer` ne prenant pas de paramètre et effaçant les affichages précédents de la messagerie (jusqu'à `Hauteur`, bien sûr)

Classe `NullDétecteur`

La classe `NullDétecteur` sera un `IDétecteur` qui ne sert qu'à représenter… l'absence d'un détecteur. Il s'agit d'une implémentation du schéma de conception *Null Object*, par lequel on évite d'utiliser des références `null` et de la validation en utilisant plutôt un objet qui représente un cas par défaut.

Chaque propriété de `NullDétecteur` lèvera `AucunDétecteurException`.

Chaque méthode de `NullDétecteur` lèvera `AucunDétecteurException`.

Structure `PipelineInfo`

Le `struct` nommé `PipelineInfo` représente les entités qui seront communiquées à travers le `CanalComm` (voir plus haut). Il s'agit d'un type simple se limitant à :

- Une propriété publique immuable `Zone` de type `Cercle`
- Une propriété publique immuable `Couleur` de type `ConsoleColor`
- Une propriété publique immuable `Symbole` de type `char`
- Un constructeur paramétrique acceptant en paramètre un `Cercle`, une `ConsoleColor` et un `char`, et initialisant les propriétés correspondantes.

Classe `Robot`

La classe `Robot` gagne aussi en sophistication :

- Plutôt que d'être initialisée à `null`, la propriété `Détecteur` doit être initialisée de manière à référer à un `NullDétecteur`. Allégez la validation des accès à la propriété `Détecteur` en conséquence
- Ajoutez une méthode `Détecter` acceptant en paramètre un `IProjetable` et retournant une `List<Point2D>`. Cette méthode déléguera son travail au `Détecteur`
- Modifiez `TrouverPassage` pour que cette méthode duplique tout d'abord la `Surface` passée en paramètre puis fasse ses tests sur cette copie locale de la surface plutôt que sur la surface originale

Classe `ZoneTransit<T>`

Implémentez une classe `ZoneTransit<T>` semblable à celle faite en exercice en classe. Faites en sorte que cette classe offre les services suivants :

- Un attribut de type `List<T>` nommée `données` et initialement vide
- Une propriété `Mutex` de type `object` initialisé à la construction
- Une méthode publique `Ajouter` acceptant en paramètre une `List<T>` et ne retournant rien. Son rôle est d'ajouter de manière synchronisée tous les éléments de cette `List` à la fin de `données`
- Une méthode publique `Extraire` ne prenant pas de paramètre et retournant une `List<T>`. Son rôle est de vider `données` et de retourner ce que données contenait, le tout de manière synchronisée. Vous **devez** utiliser `Algos.Permuter` pour y arriver
- Une méthode publique `Vider` ne prenant pas de paramètre et ne retournant rien. Son rôle est de vider `données` de manière synchronisée

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 `TP` précédents pour plus d'informations :

- Le prédicat `EstVide` exposé par la classe `Case` est désormais une propriété calculée plutôt qu'une méthode

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.