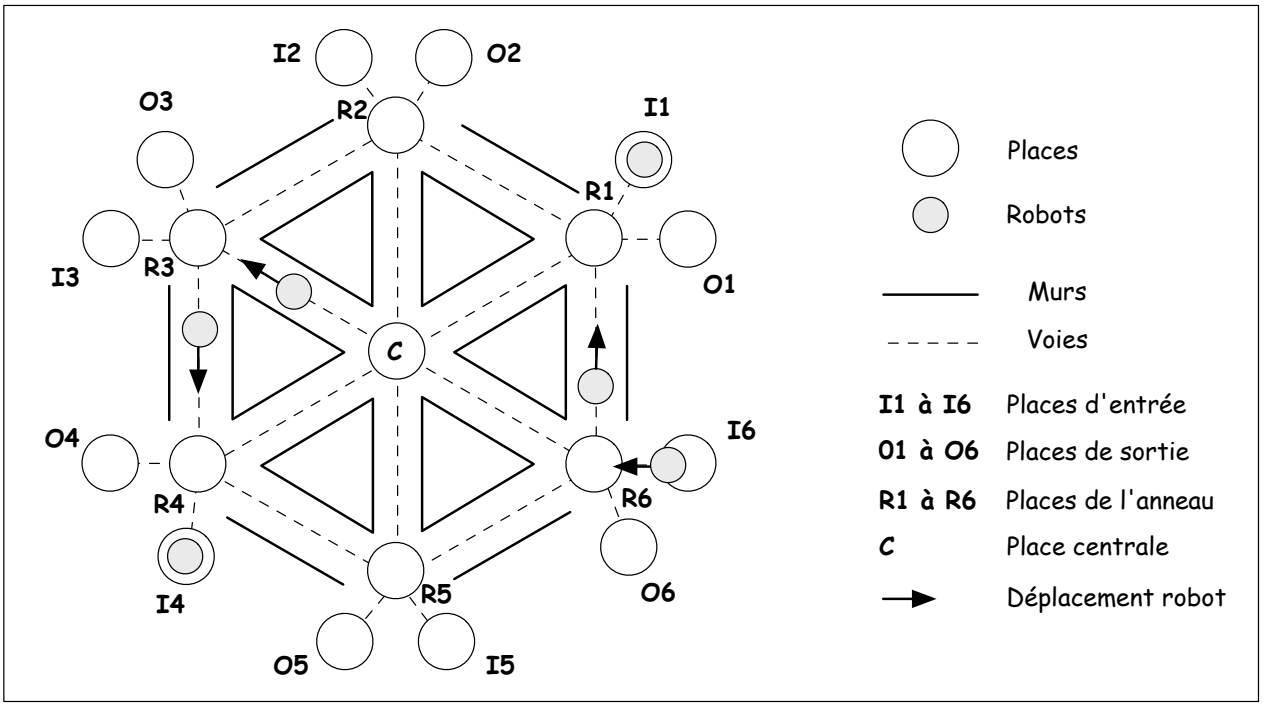
Module Ada – sujet de projet

Simulation de l'ordonnancement des missions d'une flotte de robots autonomes

© B. THIRION, ENSISA
Septembre 2013

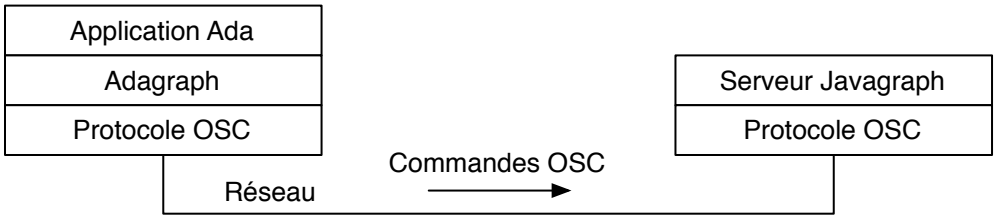
Le projet consiste à simuler l'ordonnancement des missions et le routage d'une flotte de robots autonomes partageant une infrastructure (site) de circulation commune. Le schéma ci-dessous décrit l'infrastructure.



Les robots s'engagent dans l'infrastructure partagée à partir des places d'entrées **I1** à **I6**, et quittent l'infrastructure par les places de sortie **O1** à **O6**. La circulation le long des voies est possible dans les deux sens (sauf pour les entrées et les sorties) mais un seul robot peut occuper une place ou une voie de circulation à un instant donné. Pour qu'un robot puisse se déplacer vers une place il doit avoir acquis un **privilège d'utilisation** de cette place. Pour aller d'une place d'entrée **Ii** à une place de sortie **Oj** plusieurs routes sont possibles. On se limitera cependant au cas simple consistant à toujours choisir la route la plus courte ne passant par le centre que pour les missions diagonales. Ainsi la route allant de I1 à O5 est I1-R1-R6-R5-O5 et la route allant de I1 à O4 est I1-R1-C-R4-O4 .

L'infrastructure partagée pose un problème d'**accès concurrent** et plusieurs algorithmes sont envisageables. Une solution simple, mais pas nécessairement optimale en terme de trafic, consiste pour chaque robot à acquérir à l'avance toutes les ressources de la route qu'il souhaite emprunter et à libérer ces ressources au fur et à mesure du parcours. L'allocation des ressources doit se faire selon une certaine règle pour **éviter les inter-blocages**; ce qui serait le cas si 2 robots acquièrent chacun une partie des ressources et essaient d'acquérir celles que l'autre a déjà acquises.

Afin de permettre une visualisation graphique simplifiée et portable, un paquetage graphique (Adagraph) est fourni. Ce paquetage exploite un **principe client-serveur**, le serveur graphique étant réalisé en Java.



Un exécutable de démonstration est fourni sur **Moodle** de façon à illustrer le fonctionnement attendu. Téléchargez la démonstration, lancez d'abord le serveur graphique (Javagraph.jar) puis l'application Ada (robots). **Veillez toujours à bien arrêter votre application Ada**.

La suite du projet est décrite étape par étape afin d'en faciliter la réalisation.

A) Prise en main de l'environnement

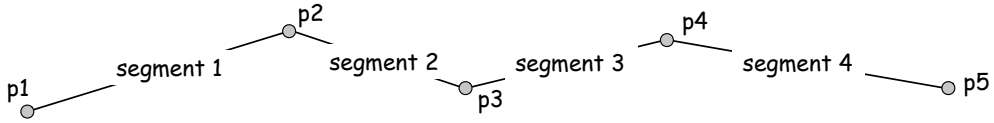
Le projet est réalisé à l'aide de **GNAT** et de l'environnement de développement **GPS**. Téléchargez (à partir de Moodle) le dossier **projects** (archives projet_win32.zip ou projet_os_x.zip). Il contient la démo (robots), les bibliothèques **adagraph** et **adaosc**, le serveur **javagraph** et un dossier **basic** avec un projet GPS permettant de débuter. Consultez le contenu du fichier de description de ce projet pour voir comment les bibliothèques sont importées. Ce projet montre quelques possibilités d'Adagraph; familiarisez vous avec ses primitives.

B) Chemins, Robots, Trajectoires, Site

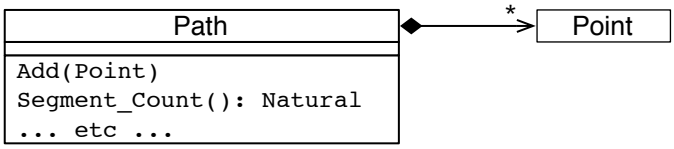
Les **robots** suivent des **trajectoires** le long de **chemins** d'un **site**. Cette phrase décrit les 4 abstractions de base qui seront traitées dans cette partie.

1) Chemins

Un **chemin** est approximé par une suite segments de lignes droites comprises entre 2 points.



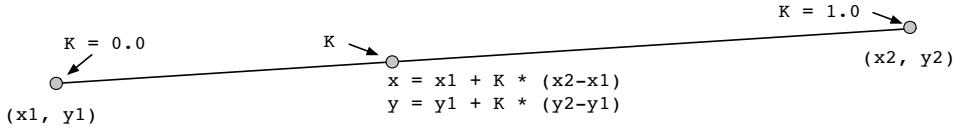
Un chemin est donc composé d'une collection de points délimitant les segments. Comme il s'agit d'une collection de taille variable, les **tableaux non contraints** et les **articles mutables** sont bien indiqués pour coder cette classe sans recourir à l'allocation dynamique. Le paquetage **Path** ci-dessous donne l'essentiel de la spécification. Complétez et codez la **spécification** et le **corps** du paquetage **Path**. Écrivez une unité **Test_Path** et testez.




```
package Path is
   type Object is private;
   Null_Path: constant Object;
   type Point is record
      X, Y : Float := 0.0;
   end record;
   type Points is array (Natural range <>) of Point;
   function Value (From: Points) return Object;
   function "&" (Left: in Object; Right: in Object) return Object;
   function "&" (Left: in Object; right: in Point ) return Object;
   function "&" (Left: in Point;  right: in Object) return Object;
   procedure Add  (Path: in out Object; P: in Point);
   function  Segment_Count (Path: in Object) return Natural;
   function  Segment_Length(Path: in Object; Segment: in Positive) return Float;
   procedure Draw (Path: in Object; Color: in Color_Type := Light_Green);
private
   subtype Count is Natural range 0..50;
   type Object (Size: Count := 0) is record
      Values: Points (1..Size);
   end record;
   Null_Path : constant Object := ......
end Path;
```

2) Accès paramétrique à un point du chemin

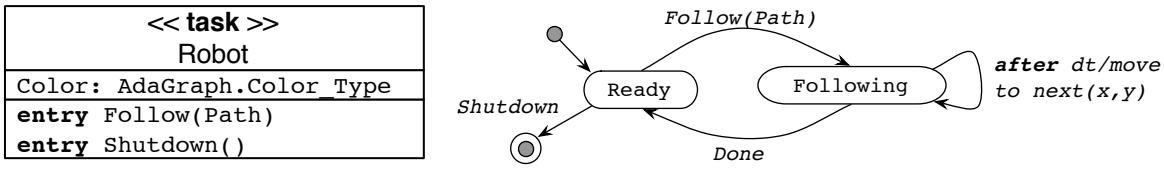
On souhaite accéder à n'importe quel point d'un segment. Pour cela on utilise l'équation paramétrique suivante :



Tout point d'un chemin peut alors être indexé en utilisant un numéro de segment et une valeur de K comprise entre 0.0 et 1.0. Ajoutez 2 méthodes *X* et *Y* à **Path** donnant l'abscisse et l'ordonnée d'un point accédé de cette manière (segment plus valeur de K). Testez en parcourant un chemin point par point à intervalle régulier (dt = 50 msec, dK = 0.1). Vous pouvez par exemple dessiner un petit «robot» (cercle ou disque) qui se déplace le long du chemin.

3) Tâche Robot

La boucle de parcours d'un chemin obtenue précédemment, ainsi que le dessin du «robot», peuvent-être transformés en une **tâche Robot** offrant 2 points d'entrée:



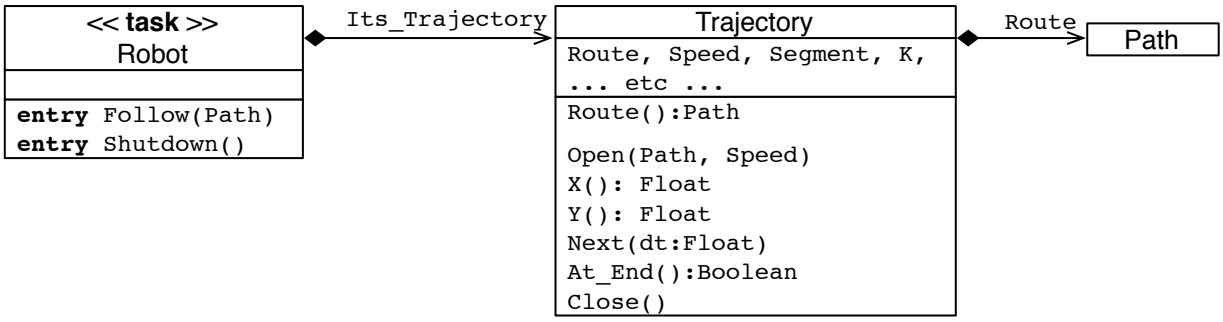
L'entrée *Follow* sert à attribuer un chemin au robot, l'entrée *Shutdown* sert à arrêter définitivement le robot. *Follow* et *Shutdown* ne sont acceptés que si le robot est dans l'état *Ready*. Une tâche robot possède un **discriminant** pour lui attribuer une couleur. Codez un type tâche selon :

```
package Robot is
   task type Object (Color: ...) is ...
   ... etc ...
end Robot;
```

Codez le corps de tâche dans le corps du paquetage Robot et refaites les tests de suivi de chemin.

4) Classe Trajectoire et unité enfant privée

En 3 vous constatez que si K évolue par pas dK constants, la vitesse du robot change selon la longueur des segments. Elle est lente sur les segments courts et rapide sur les segment longs. Pour obtenir une vitesse d'évolution constante quelle que soit la longueur du segment, la valeur dK doit évoluer selon la relation $dK = (V / L) * dt$, avec V la vitesse du robot, L la longueur du segment courant, et dt le pas d'échantillonnage (0.05 sec.). La **classe Trajectory** a pour rôle d'abstraire ce comportement sous la forme d'un **itérateur** qui sert d'interface. Trajectory découple les classes Robot et Path de la manière suivante :



Comme la classe **Trajectory** est spécifique au fonctionnement du Robot elle sera définie dans une **unité enfant privée** du paquetage Robot selon:

```
private package Robot.Trajectory is

   type Object is tagged private;

   ... etc ...

end;
```

La classe **Trajectory** sera dérivée dans une partie ultérieure afin de construire des **trajectoires sûres** qui auront pour rôle d'allouer et de libérer les ressources nécessaires. Pour l'instant la méthode *Close* est sans effet. Testez et constatez la vitesse régulière du robot (prenez une vitesse de l'ordre de 75.0). Vous pouvez faire un test avec plusieurs tâches robots évoluant simultanément, cependant le résultat n'est pas garanti car les différents

robots utilisent de manière concurrente les primitives de dessin du paquetage AdaGraph c'est donc une ressource qu'il faut protéger.

5) Site et graphisme concurrent

Afin de décrire le site, de le dessiner, et de dessiner ou effacer les robots selon leur position on construit un **singleton Site**. Pour simplifier l'animation graphique des mouvements des robots et éviter des problèmes de rafraîchissement on ne dessinera ni les voies de circulation ni les places (sauf éventuellement en phase de test). Seul les couloirs de circulation sont dessinés, les mouvements des robots ne perturbent donc pas l'affichage. Lorsqu'un robot commence une nouvelle trajectoire, le chemin qu'il va suivre est dessiné. Lorsqu'il progresse le long de sa trajectoire, l'effacement du robot (dessin du robot en noir sur fond noir) effacera progressivement le chemin (voir démo). Plusieurs robots accéderont au Site de manière concurrente, il y a donc lieu d'être prudent quant à la protection des données. S'il est possible d'abstraire l'ensemble du site à l'aide d'un objet protégé ce n'est cependant pas indispensable. En effet, si les méthodes de **Site** ne sont que des accesseurs (**fonctions sans effets de bords**) et ne modifient pas l'état du site, il n'y a pas de risque particulier vu que les sous-programmes Ada sont ré-entrants. Par contre, les primitives de dessin de l'unité **AdaGraph** sont utilisées de manière concurrente par les différents robots. Le plus simple pour obtenir une exclusion mutuelle est d'utiliser un **objet protégé** (singleton). Cet objet protégé peut être logé dans le paquetage Site de la manière suivante :

```
with AdaGraph; use AdaGraph;
package Site is

   type    Place_Names is (I1, I2, ...)
   subtype Ring_Places is ...

   ...

   function Way_Out(To: Output_Places) return Ring_Places;

   ...

   protected Safely is
      procedure Draw_Site  ...
      procedure Draw_Path  ...
      procedure Draw_Robot ...
      procedure Hide_Robot ...
      ... etc ...;
   end;

end Site;
```

Ces opérations protégées feront typiquement appel aux opérations fournies par Adagraph. Modifiez vos appels aux primitives de dessin afin d'utiliser les opérations protégées. Puisque le paquetage Site exporte des services de dessin, il est logique de créer la fenêtre de dessin, et le dessin du fond du site (partie fixe du dessin) dans ce paquetage. Vous pouvez utiliser la **partie élaboration du corps** du paquetage **Site**, à cet effet.

Site propose des fonctions pour «naviguer» de place en place (*Way_In*, *Next*, *Previous*, *Way_out*, *Opposite*, etc) et pour construire des chemins (*Route*) allant d'une place à une autre. Ces fonctions capturent la topologie de l'infrastructure. Codez la spécification et le corps de Site. Testez en générant des chemins correspondant au Site. Afin de faciliter la suite du TP modifiez l'entrée *Follow* du **Robot** en *Go(From: Site.Input_Places, To: Site.Output_Places)*, faites de même avec la méthode *Open* de la classe **Trajectory**, à savoir *Open (From, To, Speed)* et construisez la route dans le corps de cette méthode.

6) Appel d'entrée temporisé

Pour utiliser plusieurs robots simultanément, ajoutez un tableau de robots (6 par ex.) dans votre programme de test, et recrutez les robots pour leur attribuer des missions (de Ii à Oj) générées de manière aléatoire. Pour cela utilisez un générateur de nombre aléatoire.

Lorsque vous attribuez une mission (*Go(From, To)*) à un robot vous ne savez pas quand cette mission s'achève et vous ne savez donc pas si un robot est libre ou non pour traiter une nouvelle mission. Interroger les robots ne sert pas à grand chose puisque cela n'évite pas une attente active. Pour l'instant une idée simple consiste à utiliser un **appel d'entrée temporisé**, c'est à dire qu'il suffit d'essayer d'attribuer une mission à un robot et d'essayer avec le robot suivant si la mission n'est pas acceptée au bout d'un certain temps d'attente maximum (0.5 sec par ex.). Utilisez l'**instruction select adéquate** dans votre programme de test pour mettre en oeuvre cette première stratégie d'allocation des missions.

Dans tous les tests on supposera que l'acheminement d'un robot vers une place d'entrée se fait de manière instantanée. C'est à dire qu'après recrutement, un robot apparaît sur une place d'entrée, effectue sa mission puis disparaît lorsqu'il atteint la place de sortie de sa mission.

C) Exclusion mutuelle pour l'accès à l'infrastructure

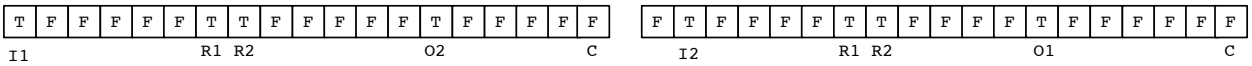
A ce stade les abstractions principales sont en place. Vous pouvez recruter des robots et leur attribuer des missions au sein d'une infrastructure, mais les trajectoires des robots ne sont pas sûres. Cette partie décrit la mise en place des trajectoires sûres qui requiert un mécanisme d'allocation et de libération de ressources.

1) Pool de ressources

Un robot qui doit effectuer une mission acquiert l'ensemble des ressources associées à sa mission (y compris les places d'entrée et de sortie) avant de pouvoir se déplacer. Il libère les ressources au fur et à mesure qu'il quitte les places qu'il a réservées. Il est donc clair qu'un robot doit **acquérir plusieurs ressources** simultanément, ce qui peut poser un problème si l'allocation n'est pas faite correctement. Par exemple si un robot 1 doit aller de I1 à O2 il doit acquérir {I1, R1, R2, O2} et si un robot 2 doit aller de I2 à O1 il doit acquérir {I2, R2, R1, O1}. Si l'allocation se fait dans l'ordre des listes précédentes il peut y avoir un blocage si le robot 1 a acquis {I1, R1} et le robot 2 a acquis {I2, R2}, car le robot 1 cherchera à acquérir R2 que le robot 2 a déjà réservé, il en va de même pour le robot 2 qui cherchera à acquérir R1 qui est déjà réservé par le robot 1, ce qui conduit à une **étreinte fatale**. Une solution simple pour éviter ce genre d'inter-blocage consiste à définir une relation d'ordre sur les ressources et à toujours allouer les ressources en respectant cette relation d'ordre. Par exemple si on a la relation:

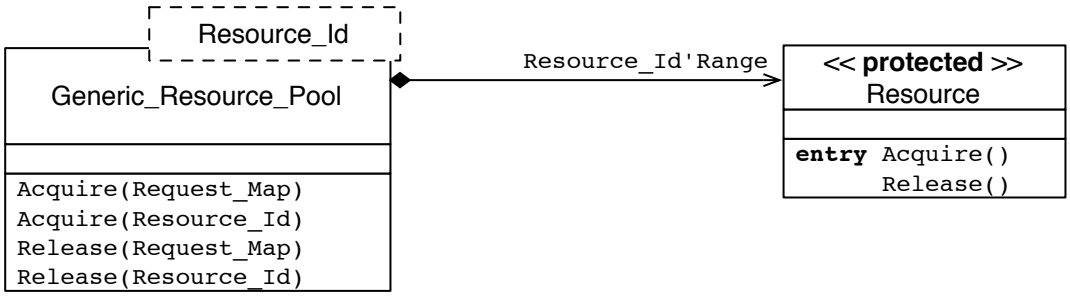
I1<I2<I3<I4<I5<I6<R1<R2<R3<R4<R5<R6<O1<O2<O3<O4<O5<O6< C

Le robot 1 réservera les ressources dans l'ordre I1, R1, R2, O2 et le robot 2 réservera les ressources dans l'ordre I2, R1, R2, O1, il ne peut donc plus y avoir d'étreinte fatale entre les 2 robots. Afin de mettre en oeuvre cette stratégie d'allocation on utilisera des **requêtes sous forme de bitmap** (des vecteurs de booléens). Par exemple pour les 2 requêtes précédentes on aurait :



Chaque **ressource unitaire** est modélisée à l'aide d'un **objet protégé** offrant 2 primitives *Acquire* et *Release*. L'ensemble des ressources est mis dans un **Pool de ressources** qui offre des primitives d'allocation et de libération de ressources unitaires ou par paquet en respectant la relation d'ordre.

Plutôt que de réaliser une abstraction totalement dédiée au problème on construira une abstraction générique paramétrée par l'identité des ressources. Le diagramme suivant récapitule la situation.



Un objet **Generic_Resource_Pool** est essentiellement un vecteur de ressources et ne nécessite pas de protection particulière. Le type protégé **Resource** peut être privé à l'unité Generic_Resource_Pool.

Codez et testez la spécification et le corps du paquetage générique **Generic_Resource_Pool**.

2) Itérateur de nom de places

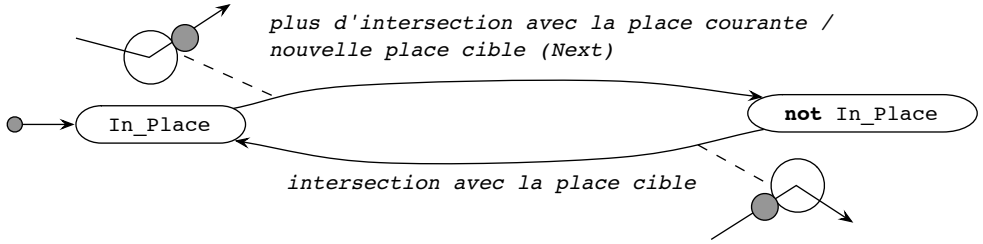
Les ressources nécessaires pour un trajet particulier sont identifiées par une liste de noms de places. Cette liste est de taille variable. Pour coder une telle liste de taille variable plusieurs solutions sont possibles. A titre pédagogique, on propose d'utiliser une **collection paresseuse** définie par un itérateur. Cet itérateur trouve naturellement sa place dans une **unité enfant du paquetage Site**, soit par exemple dans un paquetage **Site.Places_Path**. Codez et testez ce paquetage.

Site.Places_Path
Open(From, To: Place_Names): Places_Path At_End(): Boolean Value (): Place_Names Next() ...

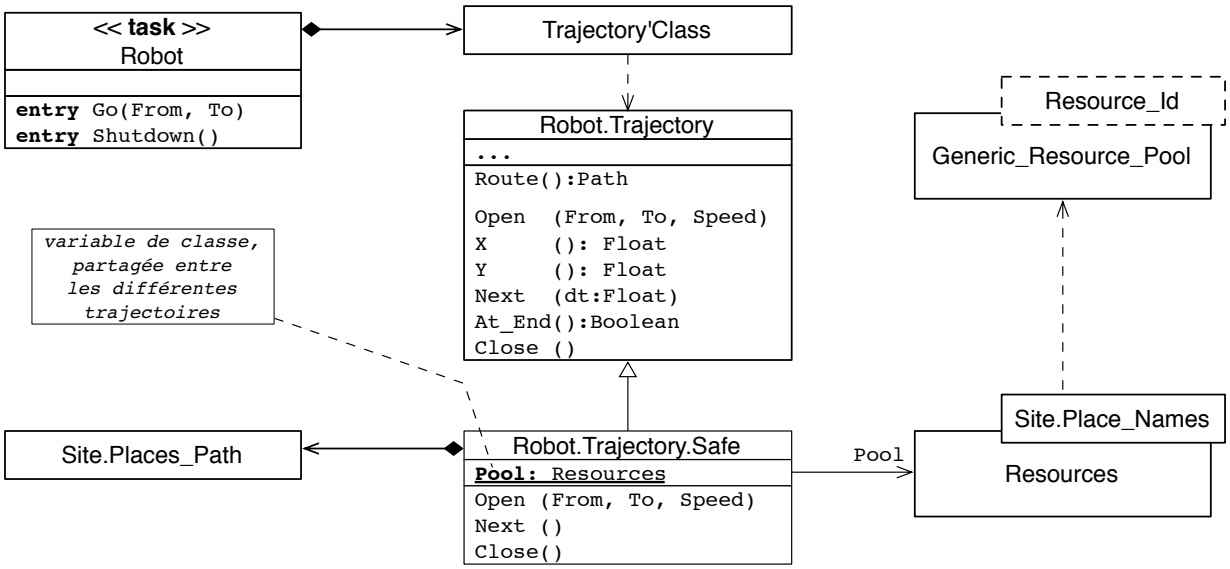
Afin de pouvoir exploiter cet itérateur il faut également savoir quand un robot se trouve sur une place. Pour cela rajouter une méthode *Robot_Intersects (Place, XRobot, YRobot)* au paquetage **Site**.

3) Sous-classe Trajectoire Sûre

En se déplaçant le long d'un chemin le robot visite les différentes places. Au départ il se trouve sur une place et à l'arrivée également. Le robot effectue des transitions (qu'il faut détecter en fonction de la distance entre le robot et le centre de la place) entre 2 états *In_Place* et *not In_Place*, selon le schéma :



La succession des places à traverser est obtenue à l'aide d'un itérateur de noms de places. Une classe **Trajectoire Sûre**, sous classe de trajectoire, contient un tel itérateur ce qui lui permet de s'occuper de l'allocation et de la libération progressive des ressources. Le diagramme suivant résume la situation.

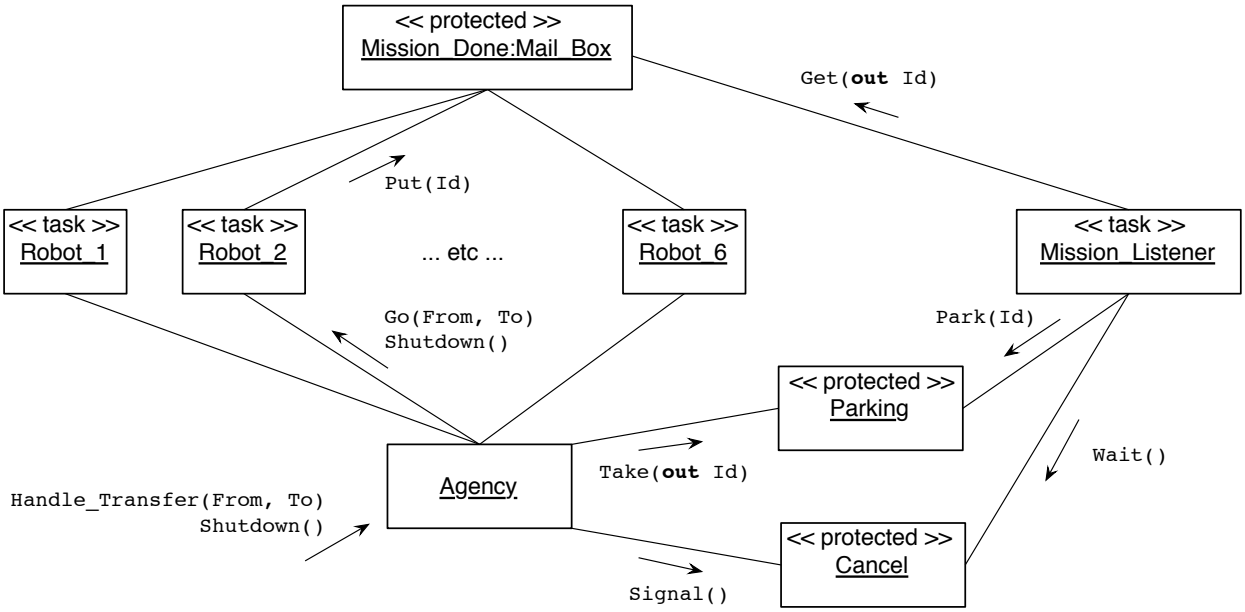


La Trajectoire Sûre redéfinit les opérations *Open*, *Next* et *Close* afin de gérer les ressources. *Next* libère les ressources au fur et à mesure, *Close* libère la dernière ressource lorsque le robot disparaît de sa place de destination. Pour l'allocation des ressources 2 principes sont possibles. Le plus simple consiste à demander toutes les ressources associées à une trajectoire dans la fonction *Open*. Cette stratégie a pour inconvénient de ne pas permettre la visualisation du robot sur une place d'entrée tant que toutes les ressources nécessaires ne sont pas disponibles. L'autre stratégie, légèrement plus difficile, consiste à ne réserver que la place d'entrée dans la fonction *Open*, ce qui permet de visualiser le robot en attente. Les autres ressources sont demandées (en bloc) lors de la première invocation de *Next* (qui signifie que le robot veut bouger).

Vous pouvez coder la trajectoire dans un paquetage enfant **Robot.Trajectory.Safe**. A ce stade les robots devraient circuler de manière fiable et sans blocage.

D) Agency, Mail_Box, Parking, Mission_Listener

Il reste à améliorer l'allocation des missions aux différents robots et l'arrêt complet du site(*Shutdown*). Pour cela une **agence** (singleton) gère une collection de robots. Elle offre 2 services *Handle_Transfer(From, To)* et *Shutdown*. Le fonctionnement général est illustré par le diagramme suivant.



Lorsqu'un transfert est demandé, l'agence consulte (*Take*) un **objet Parking** pour obtenir l'identité d'un robot libre. **Si aucun robot n'est libre l'appel est bloquant**. En fonction du robot "extrait" du parking, l'agence attribue la mission et peut ensuite traiter d'autres demandes de transfert. Les robots ont une identité (numéro) et lorsqu'un robot termine une mission il signale la fin de sa mission en déposant son identité dans une **boîte aux lettres** (**Mission_Done**). Une tâche **Mission_Listener** (membre de l'agence) extrait les messages signalant les fins de mission et met à jour l'état du parking (*Park*). La tâche **Mission_Listener** n'a aucune entrée mais est en attente, soit d'un message de la boîte aux lettres (*Get*), soit d'un **signal Cancel** (*Wait*). Elle effectue cette double attente à l'aide d'une instruction de **transfert asynchrone de contrôle** (**select** cancel.wait ... **then abort** listener.get ...). **Cancel** est un objet de synchronisation de type signal et offre 2 primitives *Wait* et *Signal*. *Signal* est invoqué par l'agence en cas de *Shutdown*.

1) l'Agence

Faites un paquetage **Agency** contenant un tableau de Robots.

2) la Boîte aux lettres

Faites une boîte aux lettres générique et instanciez la dans la spécification du paquetage Robot avec des identités de Robot. Modifier la tâche **Robot** en ajoutant 2 discriminants. Un **discriminant pour l'identité** du robot et un **discriminant accès** pour la référence vers la boîte aux lettres. Compléter le corps de la tâche **Robot** pour envoyer les messages de fin de mission.

3) le Parking

Réalisez un paquetage **Parking** qui contient un **type objet protégé** offrant 2 services. Une **entrée** *Take* qui renvoie l'identité d'un robot libre (choisi de manière équitable) et une **procédure** *Park* qui "rend" le robot au parking. Le Parking est un objet de synchronisation car **l'entrée** *Take* **est bloquante** tant qu'un robot n'est pas libre. L'appelant sera réveillé dès qu'un robot se libère ce qui évite une attente active. Le parking ne modélise que la **présence ou non des robots dans le parking** mais pas contient pas les robots. Cet objet peut également visualiser les robots sur le parking, pour cela il faut ajouter les opérations de dessin adéquates dans **Site**.

4) le Mission_Listener et l'objet protégé Cancel

Dans le corps d'Agency ajouter une **tâche Mission_Listener** et un **objet protégé Signal**. Terminez l'application en mettant en oeuvre le schéma de collaboration décrit dans le diagramme précédent. Pour cela on supposera que les robots passent instantanément de leur position dans le parking à la place d'entrée correspondant à leur mission (voir démo).

Rapport de TP

Rendez un rapport (succinct) décrivant l'*architecture générale* du logiciel, les *diagrammes de classes*, d'éventuels *diagramme d'interaction* illustrant le fonctionnement et un diagramme d'objet complet montrant l'ensemble des objets en place et leurs relations **.**