

- Programmation Orientée Objet -

Simulation de systèmes complexes

Fourmis et ressources

TD2

3^e année ESIEA - INF3034

A. Gademer & H. Wassner

2011 -2012

Avant propos

Nous allons créer notre première application Java, intégrant tous les mécanismes OO et déployant la puissance des interfaces graphiques !

1 Illustration des systèmes complexes

Un système complexe est composé d'une multitude d'entités qui interagissent (souvent selon des règles simples) et dont le comportement général produit des effets imprévisibles par l'analyse des règles d'interactions : on parle de **comportement émergent**.

L'étude des systèmes complexes est une discipline en plein essor avec la découverte d'une grande variété de systèmes : les réseaux de neurones dans le cerveau, les comportements de ruches des insectes sociaux, le réseau Internet, etc.

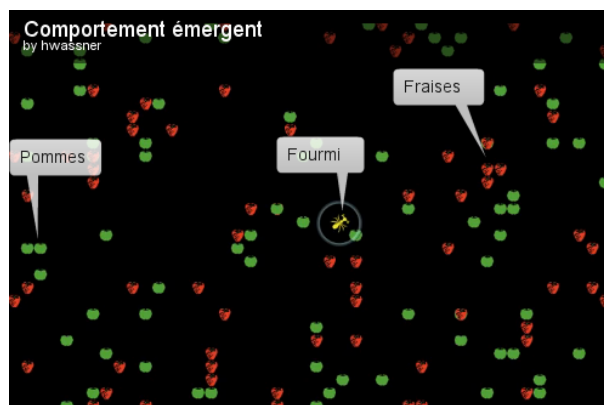
Une des illustrations les plus courantes de ces comportements émergents est celle du tri des ressources par des robots-fourmis (bien moins équipés que leurs condisciples naturels) qui suivent trois règles simples :

- Un déplacement aléatoire (sans percuter les objets)
- Si elle est face à une ressource et qu'elle a les "mains" libres, elle l'emporte.
- Si elle est face à une ressource de type identique à celle qu'elle porte, elle la dépose.

Les fourmis n'ont aucune information sur leur environnement global, ni aucune mémoire mais uniquement une information sur ce qu'elles ont en face d'elles sur le moment.

Ces trois règles engendrent un comportement émergent (une répartition spatiale des ressources) qui peut être observé sur la vidéo suivante :

<http://professeurs.esiea.fr/wassner/?2011/10/14/346-comportement-mergent>



2 Formalisation des concepts en classes

Avant de se lancer bille en tête dans le code, voyons comment passer de notre objectif : **simuler l'interaction entre des fourmis et des ressources** à un ensemble de classes "Objet".

2.1 Définition des classes et de leurs relations

Nous avons donc une simulation (**Simulation**), des fourmis (**Ant**) et des ressources (**Ressources**). Il n'y a qu'un seul type de fourmis (pour le moment) mais on trouve différents types de ressources (**RessourceType**).

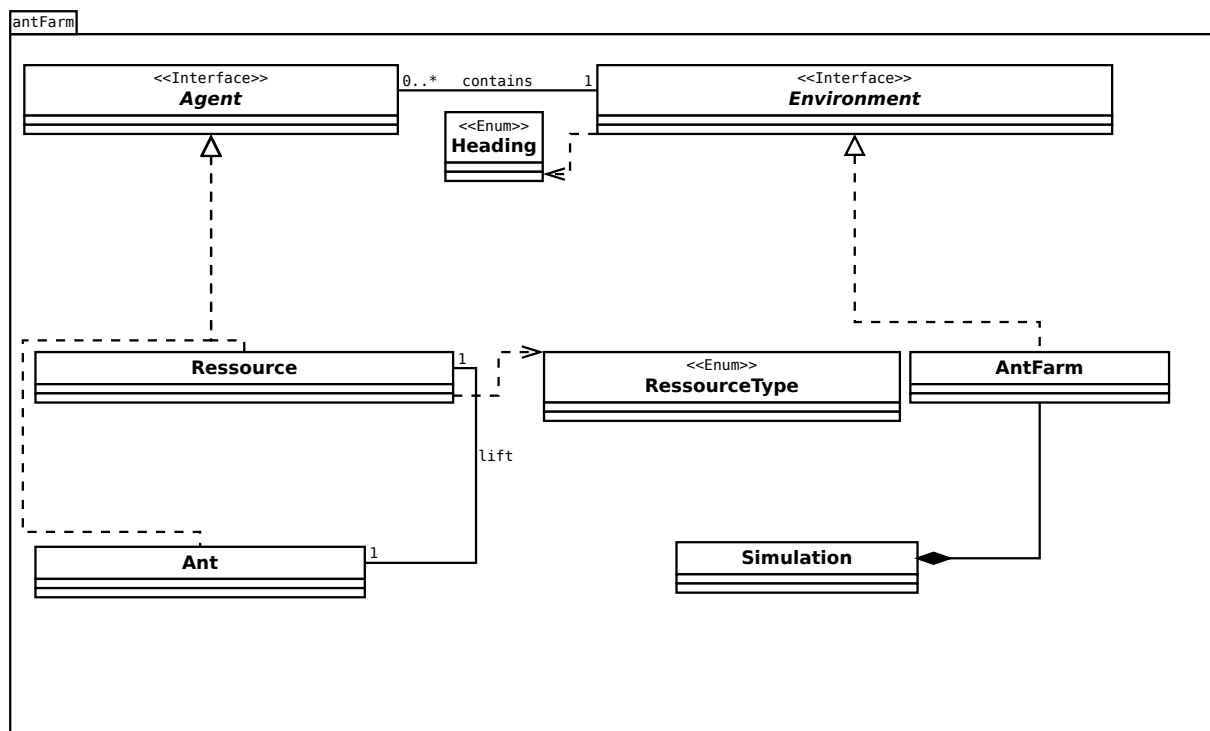
Les fourmis et les ressources sont situées dans un environnement commun que l'on peut appeler terrarium ou "ferme à fourmis" (**AntFarm**).

Pour simplifier notre problème, nous pouvons le considérer comme un environnement discret (un ensemble de cases repérées par des coordonnées x et y entières). De la même manière nous considérerons que les fourmis ne peuvent se tourner que dans les quatre directions (**Heading**) cardinales : Nord, Est, Sud, Ouest.

Afin de ne pas ressentir les effets de bords liés à la petite taille du terrain nous considérerons aussi que le terrarium est **torique** (ce qui sort d'un côté réapparaît de l'autre côté).

Si l'on **généralise** un peu notre propos, nous dirons que la simulation fait intervenir des agents (**Agent**) dans un environnement (**Environnement**) donné. Cette généralisation simplifiera la manipulation sous une même appellation des fourmis et des ressources et réduira l'interdépendance des classes.

Cela nous donne le diagramme UML de classe suivant, qui montre les relations entre les classes.



Ant et **Ressource** implémentent **Agent** et possèdent une relation d'association (quand la fourmi porte la ressource).

Une même relation d'association lie **Agent** et **Environnement**.

L'interface **Environment** est implémentée par **AntFarm** qui est un élément constitutif (composant) de la **Simulation**.

Enfin, **Environnement** dépend de **Heading** pour fournir l'information de direction et **Ressource** dépend de **RessourceType** pour décrire les différents types possibles de ressources.

2.2 Description des interfaces

Agent

On attend des **Agent** (fourmis ou ressources) qu'ils puissent faire quelque-chose `doSomething(Environment env)` à condition de connaître leur environnement (passé en paramètre). Ils doivent aussi donner des informations sur leur état : sont-ils transportables (`isLiftable()`), sont-ils en cours de transport (`isLifted()`), quel est leur nom (`getName()`) et leur position actuelle (`getPosition()`). Ils peuvent aussi recevoir des messages qui définissent leur nouveau porteur (`setLifter(Agent lifter)`), une nouvelle position (`setPosition(Position position)`) ou les informent qu'ils ont été lâchés (`dropped()`).

Enfin, afin de pouvoir les afficher dans une interface graphique, ils doivent pouvoir se dessiner à condition de recevoir un environnement graphique (`drawIt(Graphics g)`).

Environment

Pour sa part l'environnement doit faire l'interface avec les agents d'un côté et avec la simulation de l'autre côté.

Du côté agent, **Environment** doit pouvoir leur fournir les coordonnées de la case en face d'une position donnée si l'on regarde dans une direction donnée (`getPositionInFrontOf(Position position, Heading heading)`), ainsi qu'un générateur de nombres aléatoires commun pour tous les agents (`getRandomGenerator()`).

Du point de vue de la simulation, l'environnement doit pouvoir initialiser son état (`void reset()`) puis évoluer (`void evolve()`) pas à pas au fur et à mesure de la simulation.

En tant que partie intégrante de l'interface graphique, **Environment** doit pouvoir se dessiner (`void paint(Graphics g)`) et fournir l'échelle des cases du terrain (`int getScale()`).

2.3 Interface graphique

L'ensemble des classes, présenté précédemment (**Agent**, **Environment** et leurs implémentations), décrit le fonctionnement interne de notre simulation. Mais nous avons pour l'instant laissé de côté la manière dont nous désirons afficher le résultat d'une étape donnée de cette simulation et les possibilités de contrôle offertes à l'utilisateur. Ces deux informations (la vue et le contrôleur) ont volontairement été séparées du modèle de données¹ et concerne l'interface graphique du programme à proprement parler.

Notre interface graphique est minimaliste ; chaque étape de la simulation est affichée l'une après l'autre après un délai variable qui détermine la vitesse de la simulation.

L'utilisateur peut :

- démarrer/arrêter la simulation,
- si la simulation est à l'arrêt, la faire avancer d'une étape,
- réinitialiser la simulation aléatoirement,
- choisir la vitesse d'exécution.

Ceci est très simple à implémenter pour peu que nous nous appuyons sur les bibliothèques `javax.swing` et `java.awt`. Ces bibliothèques permettent de construire rapidement des fenêtres (`JFrame`) contenant des composants graphiques comme, entre autres, des textes (`JLabel`), des boutons (`JButton`) ou des sliders (`JSlider`) ainsi que leurs contrôleurs associés (`ActionListener`).

Reste l'affichage de la simulation elle-même. Nous avons dit que les **Environment** devaient savoir s'afficher grâce à leur méthode `void paint(Graphics g)`, mais d'où vient ce contexte graphique (`Graphics`) ?

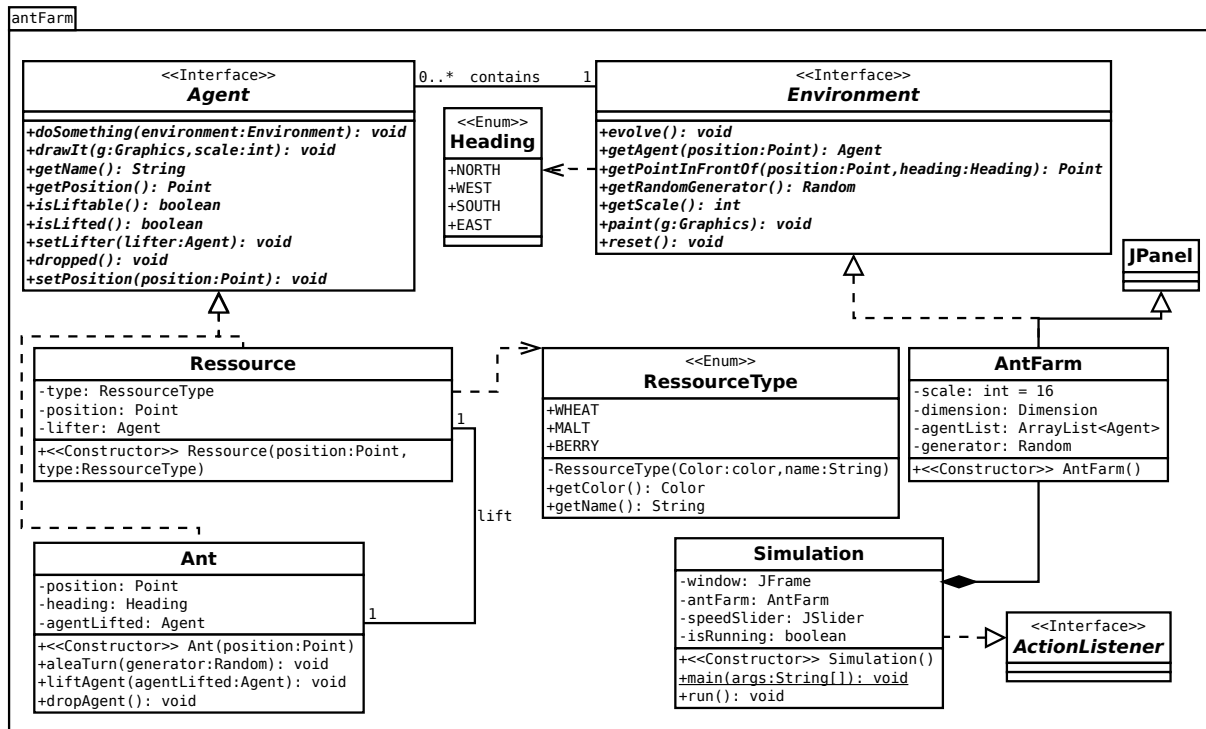
En fait nous allons détourner l'un des composants graphiques les plus simples : un `JPanel` pour lui demander d'afficher notre **Environment** et pour ce faire nous allons faire hériter **AntFarm** de `JPanel`.

Ce faisant notre classe **AntFarm devient un composant Swing à part entière** et peut être ajoutée à la fenêtre afin d'être affichée avec le reste des composants.

1. Vous verrez en détail les "design pattern", comme le Modèle-Vue-Contrôleur dans le module Génie Logiciel en S2.

2.4 UML complet

Si l'on rajoute les autres classes, que nous décrirons plus en détail au fur et à mesure, nous obtenons le diagramme de classe UML suivant :



3 Création du projet

3.1 Le plus petit programme Java du monde !

- Lancez NetBeans.
- Allez dans le menu Fichier > Nouveau projet.
- Sélectionnez Java, puis Application Java.
- Donnez un nom à votre projet : **AntFarm**.
- Notez l'emplacement de votre projet, il vous servira à le récupérer plus tard :-)
- La case **Créer une classe principale** doit être cochée.
- Dans le champ à gauche de la case, écrivez : **Simulation** (ce sera le nom de notre classe principale).
- Cliquez sur **Terminer**.

NetBeans devrait vous ouvrir un fichier contenant :

```

/*
 * To change this template, choose Tools / Templates
 * and open the template in the editor.
 */

/**
 *
 * @author
 */
public class Simulation {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }
}

```

```
}
}
}
```

4 Étape 1 : Création de la fenêtre de travail

Pour le moment, le diagramme UML 1 (p. 5) est minimal.

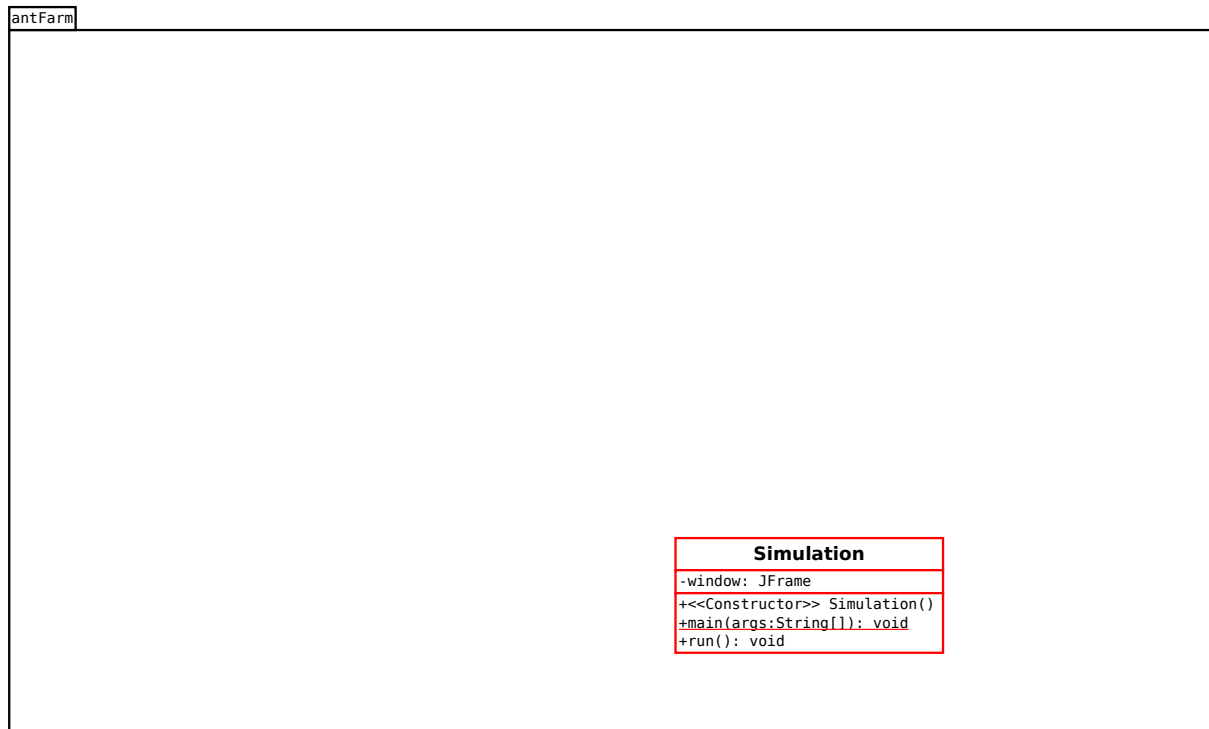


Figure 1 – Diagramme UML - Étape 1

- L'attribut `window` correspond à un objet `JFrame` qui va constituer notre fenêtre de travail.
- La méthode principale `main` se contente de créer un nouvel objet `Simulation` (1 seule ligne :-p)
- Le constructeur `Simulation`, sans paramètres, initialise l'objet `window` avec les fonctionnalités minimales :

```
//Allocating a new JFrame to be the main window
JFrame window = new JFrame("MyWindow");
//Default configuration of the window
window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
window.setMinimumSize(new Dimension(100,100));
//Resizing the windows considering the new elements
window.pack();
//Showing the windows
window.setVisible(true);
```

EXERCICE 1



Modifiez le code généré par Netbeans pour correspondre au diagramme UML. On désire appeler la fenêtre "AntFarm" et lui donner une taille minimale de 320x240. Compilez (🔧) et exécutez votre code (▶).

Vous devriez obtenir ceci :



5 Étape 2 : Mise en place des interfaces

Notre programme fait interagir deux types d'objets abstraits : les **Agent** (objets qui se déplacent sur un plan et interagissent entre eux) et l'**Environment** qui sert de terrain de jeux aux agents.

Le nouveau diagramme UML est illustré par la figure 2 (p. 6).

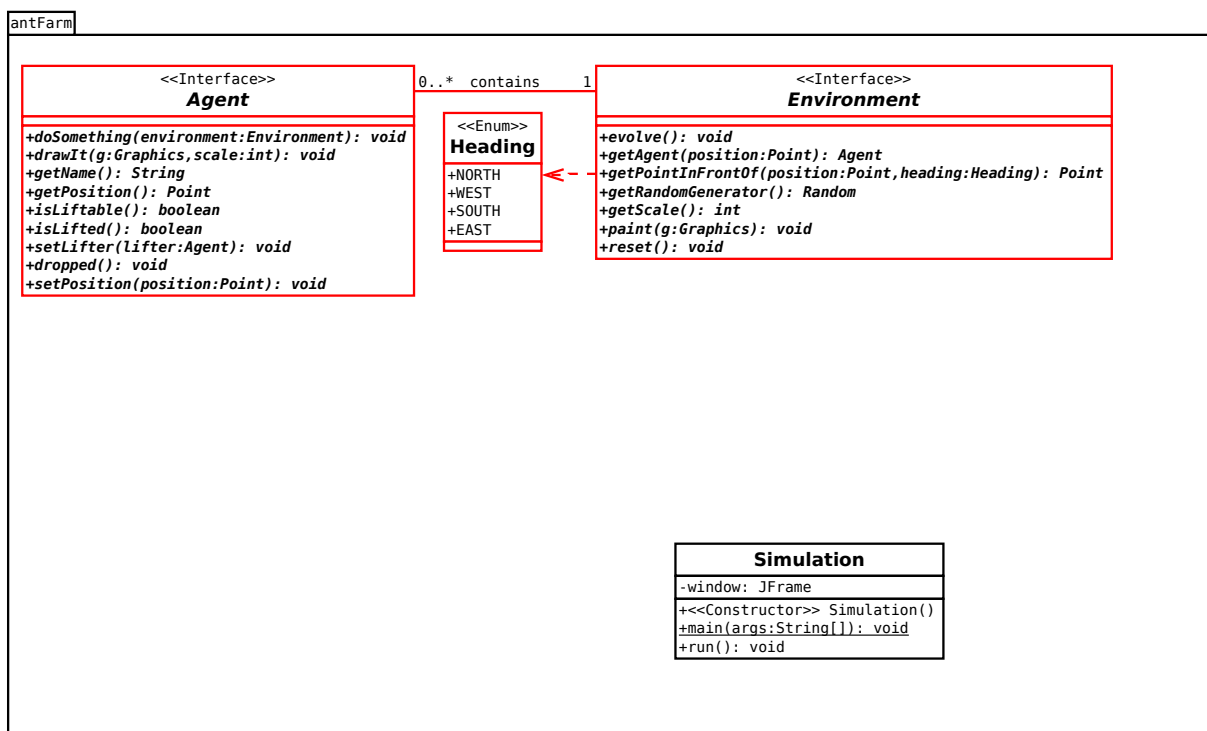





Figure 2 – Diagramme UML - Étape 2

EXERCICE 2

 Rajoutez les deux nouvelles interfaces à votre projet (Clic droit sur le package dans l'arborescence de gauche > Nouveau > Interface Java) et ajoutez les méthodes correspondantes. Compilez () et exécutez votre code (.



Énumération : Les énumérations sont des classes particulières dont l'ensemble des instances est fixé à l'avance. On les utilise généralement pour des attributs qui ne prennent qu'un nombre limité de valeurs.

Code minimaliste d'une énumération :

```
package antfarm;

public enum Heading {
    NORTH, WEST, SOUTH, EAST
};
```

Utilisation de l'énumération :

```
Heading heading; // Reference on an enum
heading = Heading.NORTH; // Setting the reference on the NORTH instance.
heading = new Heading(); // KO Enum may not have a public constructor
```

Bonus : nous pouvons utiliser des énumérations dans un **switch** :

```
switch(heading) {
    case NORTH:
        //Do something
        break;
    case SOUTH:
        //Do something
        break;
    case EAST:
        //Do something
        break;
    case WEST:
        //Do something
        break;
}
```

EXERCICE 3



Rajoutez une nouvelle énumération **Heading** à votre projet (Clic droit sur le package dans l'arborescence de gauche > Nouveau > Autre > Énumération Java) et ajoutez les instances correspondantes. Compilez (🔧) et exécutez votre code (▶).

6 Étape 3 : Aire de jeux

Il est temps de passer aux choses sérieuses ! La classe **AntFarm** est centrale car c'est elle, en héritant de **JPanel** qui va nous permettre de "dessiner" ce que nous voulons au sein de l'IHM.

La classe **AntFarm** implémente aussi notre interface **Environment** puisqu'on désire pouvoir lui ajouter des **Agent** et les faire évoluer.

Le nouveau diagramme UML est illustré par la figure 3 (p. 8).

6.1 Attributs d'instance

L'objet **AntFarm** possède quatre attributs :

- **scale** qui correspond à la largeur en pixel d'une "case".
- **dimension** qui correspond au nombre de "cases" en largeur et en hauteur. On utilise pour cet attribut un objet **Dimension**.

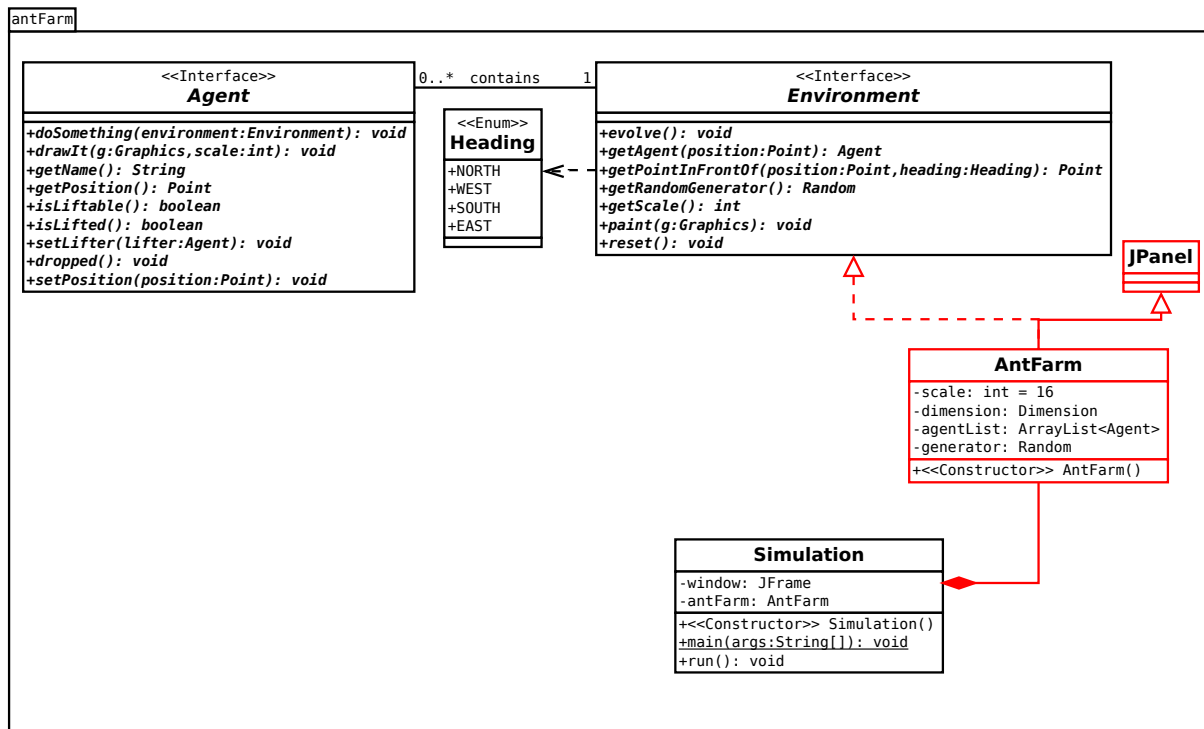


Figure 3 – Diagramme UML - Étape 3

- **agentList** qui correspond à la liste des agents (fourmis ou ressources) qui se trouvent sur le plateau. Pour conférer une grande souplesse en terme d'ajout/suppression de nouveaux agents, on utilise un objet de type `ArrayList<E>` (tableau de taille dynamique). Cette classe est dite "générique" car elle peut contenir des objets dont le type est défini à la compilation. C'est pourquoi nous remplaçons le `<E>` générique par `<Agent>`.
- **generator** est un générateur de nombres aléatoires (objet `Random`) qui nous permettra de tirer au hasard les positions des agents.



Liste dynamique vs. tableau 2D

Étant donné que notre environnement est constitué d'un ensemble bidimensionnel de cases, nous aurions pu pencher pour l'utilisation d'un tableau 2D plutôt que d'une liste dynamique. Le tableau a l'avantage de permettre de retrouver instantanément un **Agent** en fonction de sa position (puisque'il est dans la case correspondante) mais à l'inverse les cases vides sont coûteuses (en mémoire et lors des parcours). Enfin cette méthode présente une redondance d'information par rapport à l'information de position stockée dans l'objet **Agent** et nécessite donc un changement de case de l'**Agent** à chaque fois qu'il se déplace! **Retenez** : si c'est le contenant qui déplace les objets (pièces d'un échiquier par exemple), le tableau 2D peut être utilisé pour stocker la position. En revanche, si ce sont les contenus qui se déplacent, la liste dynamique est probablement plus efficace.

6.2 Constructeur `AntFarm()`

Dans ce constructeur, on cherche à initialiser les attributs des instances.

Commencez avec une **dimension** de 32 cases par 24, un **generator** sans argument et une **agentList** vide (`new ArrayList<Agent>()`).

En tant que composant Swing, il nous faut définir **notre taille en pixel** : c'est le **nombre de cases multiplié par la taille d'une case**. Pour envoyer le message à notre classe mère on utilise la méthode `setPreferredSize()`.

Enfin, le constructeur doit faire appel à la méthode `reset()` qui s'occupera de mettre en place les agents.

6.3 Accesseurs Random `getRandomGenerator()` et `int getScale()`

Les attributs étant privés, on définit deux accesseurs pour permettre à des objets de récupérer le générateur de nombres aléatoires et l'échelle.

6.4 Méthode `void reset()`

Pour le moment, la méthode `reset` ne fait pas grand chose (nous n'avons pas encore précisé quels seront nos agents).

Contentez-vous de vider la liste des agents avec la méthode `clear()` puis mettez à jour l'interface graphique avec la méthode `updateUI()` héritée de `JPanel`.

6.5 Méthode `void evolve()`

Cette méthode simule l'évolution de l'environnement. Si nous avons des propriétés internes à l'environnement (génération ou destruction spontanée des ressources par exemple), ce serait à cet endroit qu'il faudrait le faire.

Dans notre cas, cette méthode se contente de relayer le message `doSomething(this)` à chaque agent, pour lui demander de réaliser son action. Remarquez que `doSomething` attend un `Environment` et ça tombe bien car notre objet, en est un ! D'où le `this`.

Une fois chaque agent contacté, la méthode met à jour l'interface graphique avec la méthode `updateUI()` héritée de `JPanel`.

6.6 Méthode `Agent getAgent(Point position)`

Cette méthode permet de réclamer un agent de la liste en fonction de sa position.

Pour ce faire, elle parcourt la liste d'agents et compare (`equals`) leur position avec celle recherchée.

Attention, on ne prend en compte que les agents au sol (dont la méthode `isLifted()` renvoie faux).

Si aucun agent n'est présent à la position donnée, la méthode renvoie `null`.

6.7 Méthode `Point getPointInFrontOf(Point position, Heading heading)`

Étant données une position et une direction, cette méthode renvoie les coordonnées de la case "devant" la case donnée.

	0	1	2	3	4
0					
1		N (x,y-1)			
2	W (x-1,y)	(x,y)	E (x+1,y)		
3		S (x,y+1)			
4					

Voisins

	0	1	2	3	4
0	E (x+1,y)			W (x-1,y)	(x,y)
1					S (x,y+1)
2					
3					
4					N (x,y-1)

Hypothèse du
monde torique

Par exemple si un agent se trouve sur la case (1, 2) et qu'il est orienté vers l'est, la méthode renverra les coordonnées (2, 2).



Astuce

Rappelez-vous l'usage du `switch` avec une énumération (cf. p. 7)



Attention

Nous nous placerons dans l'hypothèse d'un monde torique infini. Un agent sortant du tableau à gauche réapparaît à droite. Le plus simple est de commencer par calculer les coordonnées d'arrivées, puis à ajuster celles-là si elles deviennent négatives ou supérieures ou égales à la taille du `AntFarm`.

Dans l'exemple ci-dessus si un agent se trouve sur la case (4, 0) et qu'il est orienté vers le nord, la méthode doit retourner les coordonnées (4, 4) puisque

```
y = 0 - 1 // Go North => y - 1
y = -1 // Negative coordinate !
y = -1 + 5 // Add the height of the AntFarm
y = 4 // Valid coordinate
```

Dans l'exemple ci-dessus si un agent se trouve sur la case (4, 0) et qu'il est orienté vers l'est, la méthode doit retourner les coordonnées (0, 0) puisque

```
x = 4 + 1 // Go East => x + 1
x = 5 // Coordinate out of bounds !
x = 5 - 5 // Subtract the width of the AntFarm
x = 0 // Valid coordinate
```



La classe `Graphics`

En Java, le dessin de composant passe par l'utilisation de la classe `Graphics` dont une instance est récupérée en paramètre.

Quelques méthodes de base de `Graphics` :

```
g.clearRect(int x, int y, int width, int height)
g.setColor(Color c)
g.drawString(String str, int x, int y)
g.drawLine(int x1, int y1, int x2, int y2)
g.drawOval(int x, int y, int width, int height)
g.fillOval(int x, int y, int width, int height)
g.drawRect(int x, int y, int width, int height)
g.fillRect(int x, int y, int width, int height)
```

6.8 Méthode `void paint(Graphics g)`

Cette fonction redéfinit l'affichage du `JPanel` afin de dessiner notre terrain.

Commencez par effacer l'ensemble du terrain.

Ensuite, et bien que nous n'ayons encore aucune idée de l'apparence de nos agents, nous savons grâce à l'interface `Agent` que ceux-ci possèdent une méthode `drawIt(Graphics g, int scale)`.

Nous pouvons donc envoyer le message `drawIt` à tous nos (futurs) agents qui sont dans la liste en leur passant en argument l'objet `g` reçu en paramètre et l'attribut `scale`, qui est une propriété interne de notre classe.

EXERCICE 4



Rajoutez la nouvelle classe `AntFarm` (Clic droit sur le package dans l'arborescence de gauche > Nouveau > Classe Java) et ajoutez les méthodes correspondantes. Compilez (🔧) et exécutez votre code (▶).



Remarque

Bien que notre code n'ait pas beaucoup de charme pour le moment, il est essentiel de compiler et d'exécuter au fur et à mesure pour éviter d'accumuler une quantité astronomique d'erreurs.

Normalement votre nouvelle fenêtre est un peu plus grande que la précédente (puisqu'elle tient compte des nouvelles dimensions de notre terrain.)

7 Étape 4 : Ajoutons les ressources

Peuplons un peu notre désert gris. La classe `Ressource` implémentera l'interface `Agent`.

Pour nous, une ressource :

- possède une position (x,y) sur le terrain (comme tous les agents),
- possède un type (blé, malt ou baie) et une couleur associée à ce type (jaune, orange, rouge).
- possède éventuellement une référence sur son porteur (ou `null` si elle n'est pas portée).

Afin de gérer facilement les différents types de ressources nous utiliserons une nouvelle **énumération**, mais cette fois nous utiliserons la possibilité offerte par Java de lui donner des attributs `name` et `color` et les accesseurs associés.

Le nouveau diagramme UML est illustré par la figure 4 (p. 11).

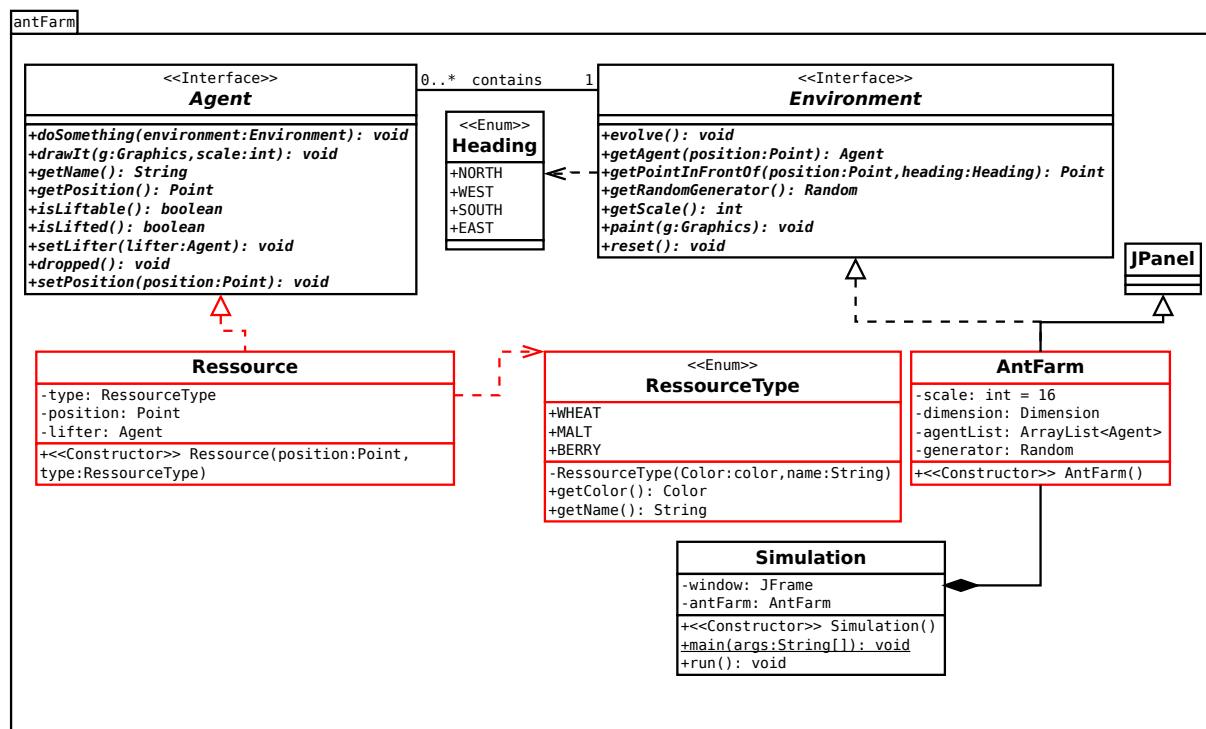


Figure 4 – Diagramme UML - Étape 4

7.1 Énumération RessourceType

Comme dans le cas de `Heading`, nous allons déclarer une énumération avec le mot-clef `enum`.

Mais contrairement à l'exemple minimaliste précédent, nous allons voir comment doter ces constantes d'attributs particuliers (une couleur et un nom).

l'`enum` doit toujours commencer par la liste des noms (en majuscules) des instances séparées par des virgules.

Mais cette fois, ces noms vont être suivis des valeurs des attributs d'instance entre parenthèses.

```
public enum RessourceType {
    WHEAT(Color.yellow, "wheat"), MALT(Color.orange, "malt"), BERRY(Color.red, "berry");
    /* ... */
}
```

Pour rendre ce code valide, il nous faut encore :

- déclarer les attributs privés qui vont contenir ces valeurs.
- déclarer un constructeur **privé** `RessourceType` qui prend en paramètre un objet `Color` et un objet `String`.
- Déclarer deux accesseurs pour pouvoir accéder aux attributs privés.

EXERCICE 5



Créez la nouvelle énumération et ajoutez les méthodes correspondantes.
Compilez (🔧) et exécutez votre code (▶).

7.2 Attributs d'instance, constructeur, accesseurs

Déclarez les trois attributs d'instance de `Ressource`, créez le constructeur (`lifter` est mis à `null` à la création), et rajoutez les accesseurs `getName`, `getPosition`, `setPosition`.

7.3 Porteur ?

- elle est portée si son porteur est non `null`,
- une ressource est toujours portable si elle n'est pas portée.

En plus des fonctions définies par l'interface `Agent`, une ressource peut recevoir un message

```
void setLifter(Agent lifter)
```

lui indiquant son nouveau porteur ou

```
void dropped()
```

qui indique qu'elle a été posée par terre (plus de porteur).

Cette méthode doit simplement mettre à jour l'attribut associé.

7.4 Quelque chose à faire ?

Dans notre modèle, les ressources sont inertes. Dans d'autres modèles agents on pourrait imaginer que les plantes croissent, que l'eau s'évapore, que la nourriture périmé, etc.

7.5 Dessine-moi une ressource

Les ressources seront représentées par des cercles remplis de couleurs (la couleur associée au type), centrés sur la case et de diamètre $\frac{scale-2}{2}$.

7.6 Rajoutons des Ressources dans notre AntFarm

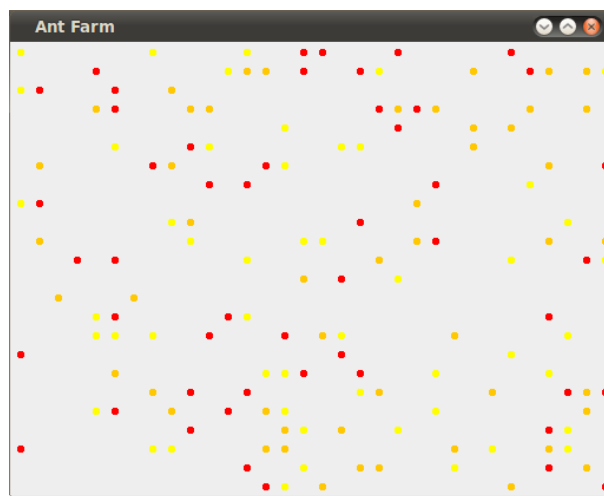
Afin d'égayer un peu notre terrain de jeu, modifiez la méthode `reset` de la classe `AntFarm` pour ajouter 50 grains de blé, 50 grains de malt et 50 baies à des positions aléatoires (attention, évitez les superpositions : vous ne pouvez poser de nouvel agent que si la position est libre de tout agent).

EXERCICE 6



Créez la nouvelle classe et ajoutez les méthodes correspondantes. Modifiez les classes concernées. Compilez (🔧) et exécutez votre code (▶).

Vous devriez avoir quelque chose ressemblant à ça :



8 Fourmis formidables

Il est temps de mettre un peu de mouvement dans cet univers statique. La classe `Ant` implémentera l'interface `Agent`.

Pour nous, une fourmi :

- possède une position (x,y) sur le terrain (comme tous les agents),
- une orientation (contrairement aux ressources, une fourmi possède un devant et un derrière),
- possède un référent sur l'éventuelle ressource transportée.

en plus des actions définies par `Agent`, les fourmis peuvent :

- choisir une nouvelle direction aléatoirement

```
void aleaTurn(Random generator)
```

- mettre une nouvelle ressource sur leur dos

```
void liftAgent(Agent agentLifted)
```

- déposer la ressource portée sur le sol

```
void dropAgent()
```

Le nouveau diagramme UML est illustré par la figure 5 (p. 14).

8.1 Attributs d'instance, constructeur, accesseurs

Déclarez les trois attributs d'instance de `Ant`, déclarez le constructeur (la fourmi est tournée vers le nord et `agentLifted` est mis à `null` à la création), et rajoutez les accesseurs `getName` (toutes les fourmis s'appellent "Ant"), `getPosition`, `setPosition`.

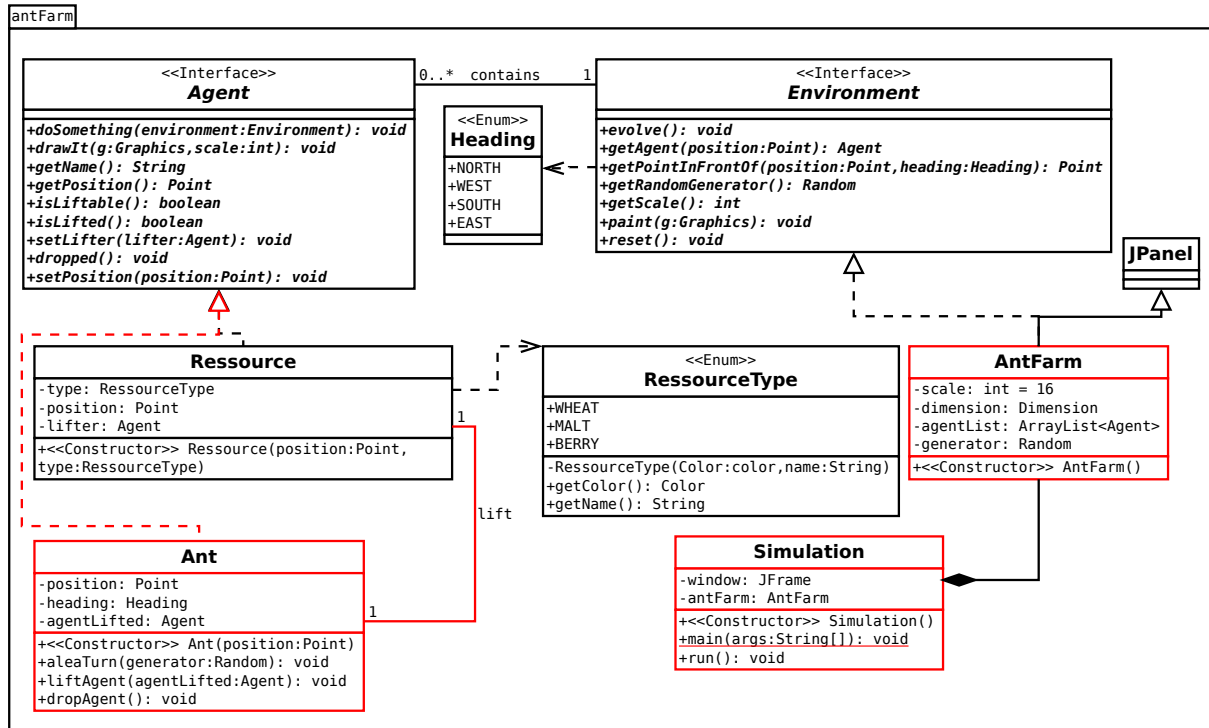


Figure 5 – Diagramme UML - Étape 5



Attention

Une fourmi ne voyage pas seule. Lorsqu'elle transporte un **Agent** sur son dos, elle doit changer la position de l'objet transporté en même temps que sa propre position.

8.2 Transporteur de fonds

On considérera que :

- les fourmis ne sont pas transportables et donc jamais transportées.
- les méthodes `setLifter` et `dropped` ne font donc absolument rien.

La méthode `liftAgent` envoie un message `setLifter` à l'agent transporté, puis met à jour l'attribut.

La méthode `dropAgent` envoie un message `dropped` à l'agent transporté, puis met à jour l'attribut.



Remarque

Comme on le voit sur le diagramme UML, le **lien d'association** entre **Ant** et **Ressource** est **bidirectionnel**, cela explique que nous prévenions l'agent transporté qu'il vient d'être soulevé ou déposé.

8.3 Quelque chose à faire ?

Dans notre modèle, les fourmis sont très actives.

Lorsqu'on lui demande d'agir, une fourmi :

- choisit une direction aléatoire,
- demande à l'environnement la position qui se trouve en face d'elle,
- demande à l'environnement l'éventuel **Agent** qui se trouve sur cette case,

- SI cette case est vide, elle change sa position pour aller sur cette case,
- SI la case contient un agent soulevable :
 - SI elle a les mandibules libres, elle soulève l'**Agent** puis prend sa place,
 - SI elle a déjà un **Agent** sur le dos ET que l'**Agent** en face d'elle est de même type (même nom),
ALORS elle dépose l'**Agent** au sol sur sa case,
- Dans les autres cas, la fourmi ne fait rien d'autre.

8.4 Dessine-moi une ressource

Dans un premier temps, les fourmis seront représentées par des cercles remplis de noir, centrés sur la case et de diamètre ($scale - 2$).

8.5 Rajoutons des Ressources dans notre AntFarm

Modifiez la méthode `reset` de la classe `AntFarm` pour ajouter 10 fourmis à des positions aléatoires (attention, éviter les superpositions : vous ne pouvez poser de nouvel agent que si la position est libre de tout agent).



Astuce

Pour des raisons d'affichage, ajoutez les fourmis **AVANT** les ressources dans la liste d'agents. De cette manière les ressources apparaitront **AU-DESSUS** des fourmis car les fourmis seront dessinées en premier.

8.6 Action !

Afin de mettre un peu de dynamique dans tout ça, il nous faut **une boucle de simulation**, c'est-à-dire une boucle qui, à intervalle régulier, va demander à notre `AntFarm` de faire bouger tous les **Agents**.

Pour cela nous allons rajouter une nouvelle méthode :

```
public void run()
```

Cette méthode définit la boucle de notre simulation :

- envoyez un message à `AntFarm` demandant de faire quelque chose.
- patientez 100 millièmes de secondes.
- recommencez à l'infini (tant que le programme n'est pas interrompu).

La méthode `run()` est appelée à la fin du constructeur de `Simulation`, juste avant de rendre visible la fenêtre.

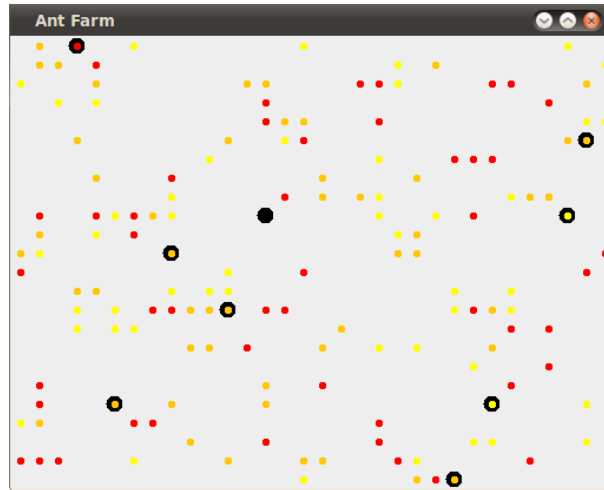
EXERCICE 7



Créez la nouvelle classe et ajoutez les méthodes correspondantes. Modifiez les classes concernées. Compilez (🔧) et exécutez votre code (▶).

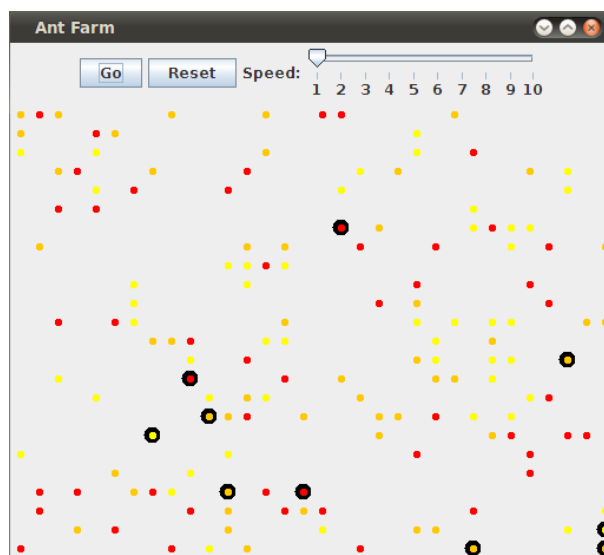
Vous devriez avoir quelque chose ressemblant à ça :

Nos fourmis s'agitent dans tous les sens !



9 Étape 6 : Interface Homme Machine

Nous allons modifier notre code pour que notre interface ressemble à ça :



Le nouveau diagramme UML est illustré par la figure 6 (p. 17).

9.1 Ajouter des composants

Pour ça, nous allons modifier le constructeur de `Simulation` pour ajouter dans l'emplacement `BorderLayout.NORTH` un `JPanel` (un vrai) qui contiendra nos deux `JButton`, un `JLabel` (`Speed:`) et un `JSlider`.

EXERCICE 8



Modifiez la classe concernée.
Compilez (🔧) et exécutez votre code (▶).

9.2 Lier les composants au reste du programme (1/2)

Notre première action va rattacher le `JSlider` à la vitesse d'exécution de notre boucle de simulation.

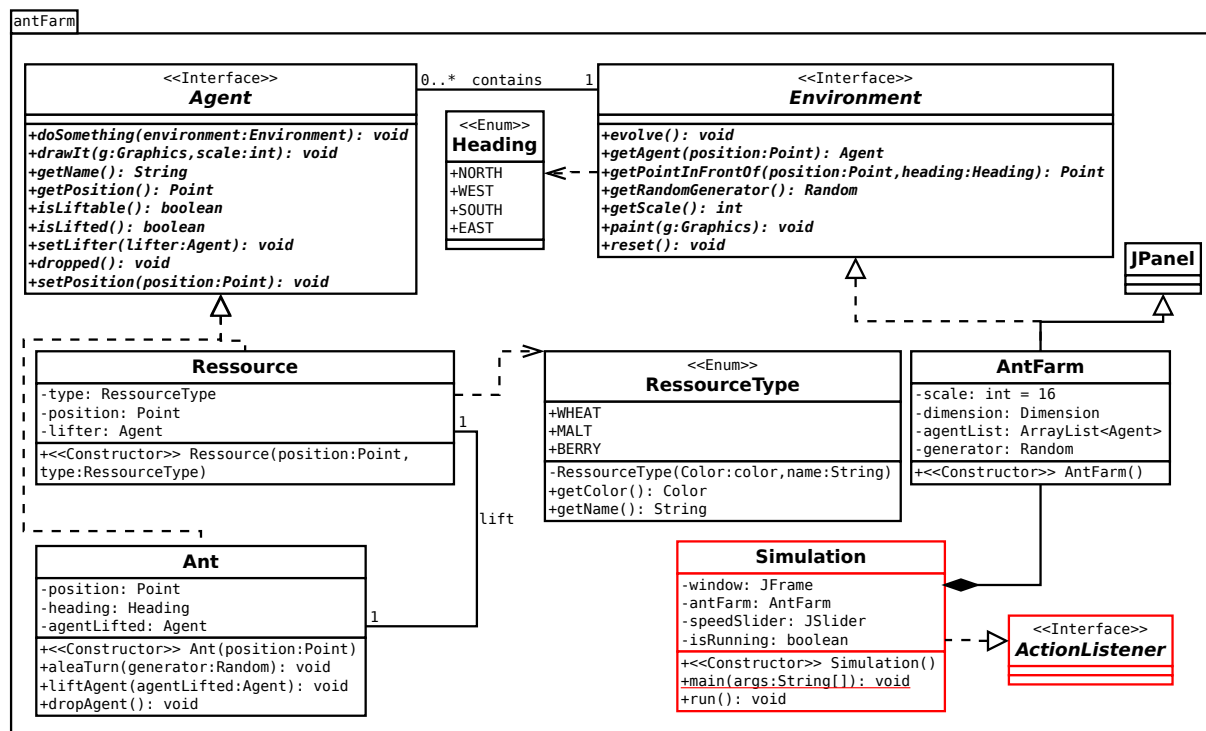


Figure 6 – Diagramme UML - Étape 6

Déclarez un attribut `speedSlider`, référence sur notre `JSlider` et qui permettra à toutes les méthodes d'instances d'avoir accès aux informations du composant.

Puis modifiez la méthode `run()` pour que le temps d'attente soit inversement proportionnel à la valeur de `speedSlider`.

$$T_{attente} = 1 + 20 \times (10 - speed)$$

Soit une valeur entre 191 ($speed = 1$) et 1 ($speed = 10$).



Astuce

Il est conseillé de laisser un minimum de 1 millièème de seconde de respiration pour éviter de saturer le processeur de la machine.

On utilisera la méthode

```
speedSlider.getValue() //Get the value of the slider set by the user
```

EXERCICE 9



Modifiez les classes concernées.
Compilez () et exécutez votre code ().

Vous devriez pouvoir contrôler la vitesse de déplacement de vos fourmis. Lorsque le curseur est sur 1, les fourmis vont lentement et vous pouvez vérifier leur comportement. Lorsque le curseur est sur 10, le temps file et vous pouvez voir les phénomènes émergents apparaître.

9.3 Lier les composants au reste du programme (2/2)

Notre deuxième possibilité d'action se fait au travers des `JButton`.

Les `JButton` sont l'exemple typique de la programmation événementielle : on ne sait pas quand l'utilisateur va juger bon de cliquer et ils n'ont pas vocation à être directement **associés** à notre programme.

La liaison se fait par l'intermédiaire d'un gestionnaire d'événements qui implémentera l'interface `ActionListener`.

Le plus simple pour nous est de demander à `Simulation` d'implémenter l'interface `ActionListener`.

Celle-ci définit une unique méthode :

```
public void actionPerformed(ActionEvent e)
```

Cette méthode est appelée par tous les composants qui sont liés (`JButton` mais aussi `JMenuItem` ou `JCheckBox`...) à elle lorsque l'utilisateur interagit avec eux.

On lie un composant à son "listener" avec la méthode `addActionListener` :

```
JButton goButton = new JButton("Go");  
goButton.setActionCommand("go"); // Unique marker to identify the source  
goButton.addActionListener(this); // If we are in the Simulation constructor,  
// "this" is the ActionListener
```



Qui a déclenché l'événement ?

Comme plusieurs composants peuvent être liés au même gestionnaire d'événements, on peut avoir besoin d'un indice pour distinguer la provenance du message : c'est le rôle de l'attribut `actionCommand` de type `String` et qui correspond à un identifiant différent pour chaque émetteur.



Comment accéder aux ressources ?

Lorsque l'on implémente une interface (`Runnable`, `ActionListener`), on a souvent besoin d'accéder aux ressources de notre programme (`JButton`, `AntFarm`, etc.). Le moyen le plus propre d'y accéder et de déclarer des attributs d'instance qui sont des référents sur ces objets et qui sont initialisés dans le constructeur. Ils sont alors accessibles implicitement depuis toutes les méthodes !

EXERCICE 10



Rajoutez la méthode `actionPerformed` puis modifiez le code de la fonction pour que celle-ci envoie le message `reset` à notre instance d'`AntFarm` lorsque le bouton `RESET` est cliqué.

9.4 Mettre en "pause"

On aimerait pouvoir arrêter (et reprendre) le déroulement de l'action à volonté, de manière à pouvoir observer l'état du terrain en toute tranquillité.

Pour cela, il va nous falloir rajouter un attribut `boolean isRunning`. Notre boucle de simulation devient :

- si `isRunning` vaut `true`
 - envoyer un message à `AntFarm` demandant de faire quelque chose.
 - patienter $T_{attente}$ millièmes de secondes.
 - recommencer à l'infini (tant que le programme n'est pas interrompu).
- `isRunning` vaut `false` au départ et c'est l'appui sur le bouton `GO` qui va modifier sa valeur.



Confort de l'utilisateur

Lorsqu'un même bouton peut avoir plusieurs fonctions selon le contexte (PAUSE ou GO, ici) il est d'usage de modifier le texte (ou l'icône) du bouton pour indiquer dans quel état il est. On aimerait donc afficher "GO" si la simulation ne tourne pas et "PAUSE" si elle tourne.

(Indices : utilisez `goButton.setText()` et faites attention à la portée de `goButton`.)

EXERCICE 11



Modifiez la méthode `actionPerformed` pour que celle-ci modifie l'état de `isRunning` et `goButton` lorsque le bouton GO est cliqué.

10 Améliorations

10.1 Pas à pas

EXERCICE 12



Rajoutez un bouton STEP qui n'est activé (`setEnabled()`) que si la simulation ne tourne pas. Lorsque l'utilisateur clique sur celui-ci `antFarm` reçoit le message `doSomething()` puis se met à jour.

10.2 Utiliser des images

La classe `ImageIcon` permet de charger des images en mémoire à partir de fichier PNG ou JPEG.

```
ImageIcon imgNorth = new ImageIcon("./img/ant_NORTH.png");
if (imgNorth.getImageLoadStatus() != MediaTracker.COMPLETE) {
    System.out.println(imgs[i] + " not found");
}
```

On utilise ces images dans la méthode `paint(Graphics g)` avec la méthode `paintIcon` :

```
imgNorth.paintIcon(null, g, x, y);
```



`ant_NORTH.png` `ant_SOUTH.png` `ant_EAST.png` `ant_WEST.png`

(Les images sont disponibles sur le blog.)

EXERCICE 13



Modifiez la classe `Ant` pour remplacer les cercles noirs par des images de fourmis orientées dans la bonne direction. Remarque : en déclarant ces nouveaux attributs `static` vous éviterez de charger les images en mémoire de nombreuses fois !

On obtient :

