université PARIS-SACLAY

Travaux Dirigés du projet FoxWar

Cours de programmation modulaire

—Licence MI/IM/MNSI - Info 3—

Types abstraits pour le projet Lapins/Renards

Nous allons étudier les types abstraits découlant d'une analyse du projet de simulation du système Renards/Lapins. Les questions qui suivent ont pour but de vous aider dans la réalisation du projet. Nous allons considérer les **types abstraits** suivants :

- `Coord` pour contenir les coordonnées d'un point de la grille ;
- `Ensemble` pour contenir un ensemble ;
- `Animal` pour coder un animal ;
- `Population` pour coder les populations d'animaux ;
- `Grille` pour contenir l'état du monde à un moment donné.
- `Jeu` pour contenir l'état du jeu à un moment donné.

Les coordonnées et ensemble de coordonnées Dans tout ce sujet d'aide, on suppose que l'on dispose des types abstraits `Coord` et `Ensemble` que vous implémenterez lors du dernier TP. Les méthodes utiles du type abstrait `Coord` seront :

- le constructeur `Coord(int x, int y);` qui construit la coordonnée (x, y) ;
- les getters `Coord::getX()` et `Coord::getY()` ;
- les opérateurs d'égalité et d'inégalité pour le type `Coord` ;
- la fonction `EnsCoord voisins(Coord c);` qui renvoie l'ensemble des coordonnées des cases voisines de c.

1 Architecture générale

Le jeu se passe sur une grille, on va donc naturellement stocker le monde avec un tableau à deux dimensions (sachant qu'il existe des structures optimisées comme les quad-tree pour rassembler les cases vides). Cependant, si la grille comporte un nombre important de cases ou si la structure animal devient grosse, choisir de stocker une structure d'animal dans chaque case de la grille va consommer beaucoup de mémoire. On va donc choisir une autre solution :

- Les animaux seront stockés dans une structure à part nommée `Population`. Pour retrouver un animal dans la population, chaque animal aura un identifiant unique qui sera un entier positif (typiquement une position dans un tableau). Chaque animal connaît son identifiant et sa position (`Coord`) dans la grille.
- Avant de créer un nouvel animal, on demandera à la `Population` de nous fournir un nouvel identifiant. Elle pourra en profiter pour réserver de la mémoire pour stocker le nouvel animal.
- Un animal conservera son identifiant tout au long de sa vie. À un moment donné deux animaux différents auront des identifiant différents. Après la mort d'un animal, son identifiant ainsi que l'emplacement mémoire qu'il occupait peuvent être réutilisés.

— Chaque case de la grille contiendra l'identifiant de l'animal s'il y en a un ou bien un nombre négatif (par exemple -1) s'il n'y a pas d'animal.

2 Animaux et cases de la grille

Commençons par le type pour les animaux. Il faut d'abord représenter l'espèce de l'animal :

1. Déclarer un type `Espèce` pour représenter l'espèce d'un animal qui peut être soit lapin, soit renard.

En utilisant ce type, on construit un type `Animal`. Chaque animal sera repéré dans la population par un entier appelé *identifiant*. Deux animaux différents ne peuvent avoir le même identifiant.

— Le contructeur `Animal` crée un animal à partir de son identifiant (un entier), son espèce et ses coordonnées.
Note : Selon comment on stocke la population d'animaux (en particulier si l'on utilise un tableau), on peut avoir besoin de créer aussi un constructeur par défaut (sans paramètres) pour les animaux.

— Accès :

`Animal::getId` renvoie l'identifiant de l'animal.

`Animal::getCoord` renvoie les coordonnées de l'animal.

`Animal::setCoord` change les coordonnées de l'animal.

`Animal::getEspece` renvoie l'espèce de l'animal.

`Animal::toString` convertit l'animal en chaîne de caractères pour l'affichage.

— Prédicats :

`Animal::meurt` est-ce que l'animal meurt de faim.

`seReproduit` est-ce que l'animal se reproduit, connaissant le nombre de voisins vides.

— Modification :

`Animal::mange` l'animal se nourrit.

`Animal::jeune` l'animal ne mange pas.

1. Donnez les spécifications des contructeurs, fonctions et méthodes ci-dessus en précisant bien lesquelles sont constantes.

3 La population

— Le contructeur `Population` crée une population vide ;
— Accès :

`Population::get` renvoie l'animal ayant un identifiant donné.

`Population::getIds` renvoie l'ensemble de tous les identifiants d'une population.

— Modification :

`Population::reserve` réserve un identifiant non déjà existant pour un nouvel animal. Une place mémoire est réservée pour stocker cet animal.

`Population::set` range un animal dans la population. L'identifiant de l'animal doit avoir été obtenu avec la méthode `Population::reserve`.

`Population::supprime` supprime un animal de la population connaissant son identifiant.

2. Donnez les spécifications des contructeurs, fonctions et méthodes ci-dessus en précisant bien lesquelles sont constantes.
3. Proposer une structure de données pour le type concret `Population`.

4 La grille

Le type abstrait `Grille` représente une grille. Voici un extrait de ses spécifications :

— Le contructeur `Grille` crée une grille vide ;
— Accès :

`Grille::caseVide` renvoie `true` ou `false` selon que la case est vide ou non.

`Grille::getCase` renvoie l'identifiant de l'animal qui est dans la grille à des coordonnées données.

— Modification :

`Grille::videCase` vide une case connaissant ses coordonnées.

`Grille::setCase` range un animal (par son identifiant) dans une case de coordonnées données.

4. Donnez les spécifications des contructeurs, fonctions et méthodes ci-dessus en précisant bien lesquelles sont constantes.

5 Le jeu

Le jeu est donc une structure qui contient une grille et une population.

5. Écrire la méthode `Game::ajouteAnimal` qui étant donné un animal et des coordonnées crée un nouvel animal et le place dans la grille.
6. Réalisez le constructeur de jeu de façon à ce que dans chaque case, il y ait 7% de chance d'avoir un renard, 20% de chance d'avoir un lapin et donc 73% de chance que la case soit vide.

7. Réalisez une méthode `Jeu::verifieGrille` qui vérifie que chaque animal est bien à sa place dans la grille.
8. Spécifiez et réalisez une méthode `voisinsVides` qui retourne l'ensemble des cases voisines vides d'une case.
9. Comment généraliser cette méthode pour qu'elle retourne les voisins qui contiennent les animaux d'une espèce donnée.
10. Étant donnés une jeu contenant déjà des animaux, et un animal, écrire une méthode qui déplace l'animal sur une case voisine libre et le range dans la grille.

6 Intégrité de la modélisation

L'état de la simulation est donc stocké dans deux structures de données : la grille et la population. Ces structures de données sont *redondantes*, c'est-à-dire que *la même information est stockée plusieurs fois*. C'est une manière de programmer dangereuse, car en cas d'erreur de programmation, les deux informations peuvent devenir incohérentes. Voici les redondances :

— **Identifiant d'un animal :** Chaque animal enregistre son identifiant. On pourrait avoir stocké par erreur un animal à un emplacement qui ne correspond pas à son identifiant.

De plus, chaque animal enregistre ses coordonnées dans la grille qui elle-même sait quel animal est dans une case donnée.

— **Cohérence animal-grille :** On peut donc avoir un animal qui «pense» être dans une case qui est vide.
— **Cohérence grille-animal :** Ou bien une case qui contient un animal qui n'existe pas dans la population.

13. Écrire une méthode qui teste que tout est bien cohérent. En cas de problème, faire un affichage qui décrit le problème, puis sortir du programme avec une exception `runtime_error` Pour ceci, il faut avoir inclus `exception`.
14. Cette procédure pourra être appelée après chaque mouvement d'animal pour vérifier la cohérence de la simulation.

Note : tous ces problèmes viennent de l'utilisation des indices et des coordonnées. En général, on évite ces redondances en utilisant des méthodes de programmation plus avancées comme les références ou les pointeurs.