8INF957 - Programmation orienté objet avancée - Automne 2022

TP2

Rapport de la quetion1

EQUIPE

Nathan AMSELLEM: AMSN25080004
Valentin THEDON : THEV10079801

1. Introduction

Dans le cadre du cours 8INF957 de programmation orienté objet avancée. Nous avons réalisé la question 1 du TP2 visant à réaliser une application exploitant des *Threads*. La consigne est simple : nous devons créer une interface pouvant nourrir des pigeons. Cependant nous avions quelques contraintes que voici :

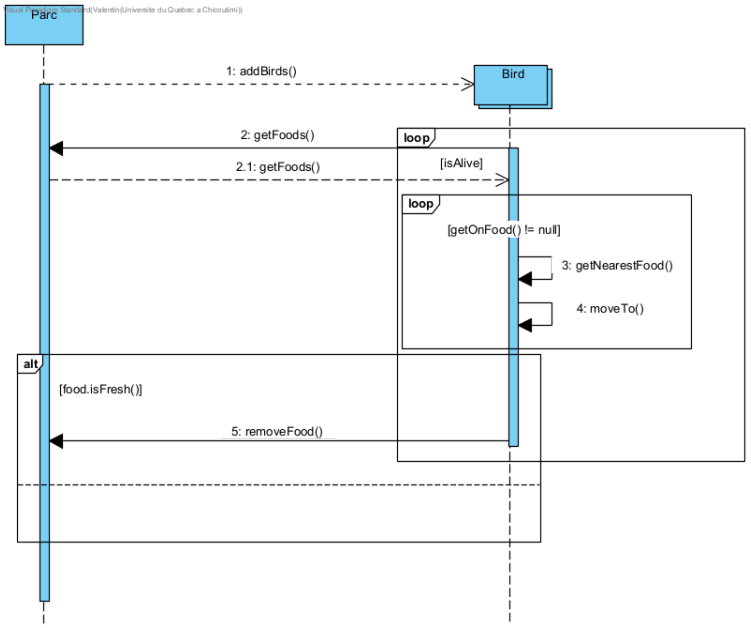
- Chaque pigeon est contrôlé par un thread;
- Si rien ne se passe, les pigeons s'endorment et ne bougent pas;
- En apercevant la nourriture, un pigeon se déplace vers la nourriture la plus fraîche;
- Une nourriture fraîche touchée est changée, donc elle doit disparaître immédiatement de la scène;
- Si un pigeon touche une nourriture pas fraîche, il l'ignore.
- Même en l'absence de nourriture, parfois les pigeons se font effrayer et ils se dispersent à des positions aléatoires.

Dans ce document, nous vous détaillerons dans un premier temps notre analyse du fonctionnement de l'application argumentée d'un diagramme de séquence, nous continuerons ensuite sur la conception de notre application, à l'aide d'un diagramme de classe. Enfin nous vous expliciterons quelques méthodes qui méritent une attention particulière.

2. Analyse de l'application

Pour ráliser cette application, nous avons tout d'abord identifier plusieurs intervenant dans notre application, qui interagiront tous ensemble :

- Le parc, dans lequel nos oiseaux et nos nourritures pourront évoluer;
- Les nourritures, qui permettront de nourrir nos oiseaux;
- L'oiseau, pour qui le comportement devra s'exécuter dans un *Thread*

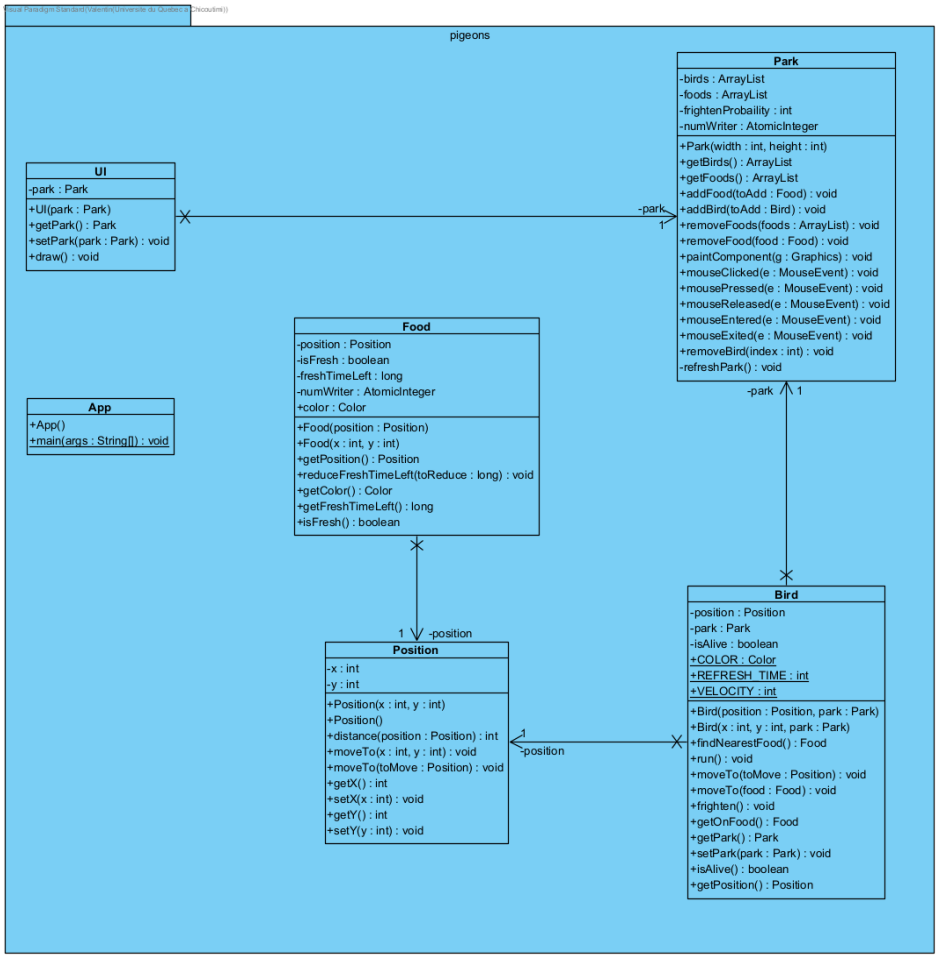


Dans ce diagramme de séquence, lorsque notre parc ajoute des oiseaux, il va démarrer le *Thread* relatif à cet oiseau. Ensuite, notre oiseau va quant à lui démarrer sa recherche de nourriture, qui l'exécute tant qu'il est en vie (dans notre application, un oiseau ne mourra jamais, mais pour plus de réutilisabilité sur notre code, il était intéressant d'implémenter cette option). Pour ce faire, il récupère l'ensemble des nourritures présentes sur le parc, va calculer la plus proche de sa position, et se déplace vers cette dernière, une fois sur cette nourriture, il la mange et donc la retire du parc.

En ce qui concerne, la nourriture, pour gérer son mécanisme de fraîcheur, le parc pourra vérifier quelles nourritures sont périmées à intervalle de temps régulier, à l'image d'un employé qui passerait faire le ménage de manière régulière.

3. Conception de l'application

Après avoir réalisé notre analyse, nous commençons à voir se dessiner nos différentes classes/objets qui seront dans notre application.



Nous observons au travers de ses classes, les 3 classes métiers de notre application : *Food*, *Bird* et *Park*. La classe *Bird* possède tout de même une particularité, elle implémente l'interface *Runnable*, cela a pour effet de nous obliger à rédiger une méthode run(), cette méthode sera par la suite exécutée dans un *Thread*.

Au vue de la simplicité l'application et de l'interface graphique, nous avons pris la décision d'implémenter directement dans nos classes métiers l'interface graphique. Il aurait été possible de mettre en place un modèle vue-contrôleur, mais cela n'est pas l'objet d'étude de cette question.

4. Focus sur des parties critiques

4.1. Le processus du pigeon

```
@Override
  public void run() {
    while (isAlive) {
      Food nearestFood = findNearestFood();
      if (nearestFood != null) {
        this.moveTo(nearestFood);
        Food onFood = this.getOnFood();
        if (onFood != null && onFood.isFresh()) {
          this.park.removeFood(onFood);
        }
      }
      try {
        Thread.sleep(REFRESH_TIME);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
```

Revenons sur la partie de code exécutée par un *Thread*, par un oiseau. Comme spécifié plus haut, tant que notre oiseau est en vie, nous travaillerons. Nous commençons donc par rechercher la nourriture la plus proche du parc. S'il existe une nourriture sur le parc, alors on se déplace vers cette nourriture. Enfin, si nous sommes sur une nourriture et que cette dernière est fraîche, alors nous la mangeons, et la supprimons du parc.
Nous attendons, un temps de rafraîchissement, défini de manière statique à la classe *Bird,* puis réitérons, le travail.

En effectuant de cette manière, cela nous permet d'ajouter de la nourriture à n'importe quel moment, plus proche que celle vers laquelle se dirige l'oiseau. L'oiseau pourra donc se diriger vers cette nourriture fraîchement distribué

4.2. Le système de verrou

```
public synchronized void removeFoods(ArrayList<Food> foods) {
    numWriter.incrementAndGet();
    this.foods.removeAll(foods);
    numWriter.decrementAndGet();
    notifyAll();
  }
```

```
public synchronized ArrayList<Food> getFoods() {
    try {
      while (numWriter.get() != 0) {
        wait();
      }
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    return foods;
  }
```

Pour éviter tout problème de famine, ou bien de perte de données, nous avons mis en place un système de verrou. En effet, le pigeon, souhaitant trouver la nourriture désire être sur qu'aucune autre nourriture a été ajouté au parc. Quand on désire retirer une nourriture, quand elle est mangé par exemple, on incrémente la variable num Writer, qui est de type *AtomicInteger* (les altérations sur ce genres de types sont faites de manières atomique, donc impossible de perdre la cohérence sur la donnée), on effectue ensuite notre suppression. Et enfin on décrémente notre variable numWriter et notifions l'ensemble des *Threads* en attente.

Pour le moment, on ne comprend pas trop l'intérêt de notre variable numWriter. En jetant un coup d'œil à la méthode getFoods(), nous comprenons directement, tant qu'une écriture est en cours (ou dans la liste d'attente), nous patientons, une fois que nous somme notifier, nous vérifions si personne demande à écrire, si c'est ce cas, nous renvoyons les nourritures du parc.

5. Conclusion

Au cours de cette question nous avons mis en œuvre des processus fonctionnant en parallèles, des *Threads*. Afin de mieux comprendre notre code, nous vous invitons à consulter la documentation Javadoc (présente dans le dossier doc) et à fouiller ce dernier.