UQAC

8INF957 – Programmation objet avancée

Université du Québec
Chicoutimi (UQAC)

Département d'informatique et de mathématique

TP2 — Question 1 : Simulation d'alimentation de pigeons

Date de distribution : 07 octobre 2025
Date de remise : 11 novembre 2025

Auteurs :

Manon Le Vallois
Roland Oucherif
Sophie Rumeau

Professeur : Hamid Mcheick

Session : Automne 2025

Table des matières

1 Introduction

Ce document est le rapport correspondant à la Question 1 du TP2 pour le cours 8INF957 (Programmation objet avancée). Il décrit la conception, l'implémentation et les décisions prises pour réaliser une simulation d'alimentation de pigeons dans un espace public.

2 Objectifs

Les objectifs spécifiques de la question 1 sont :

— Mettre en œuvre une simulation multi-threadée où chaque pigeon est contrôlé par un thread.
— Garantir la robustesse concurrente lors de l'ajout/suppression de nourriture et du dessin.
— Fournir une interface graphique simple pour interagir (ajout de nourriture par clic) et observer le comportement.
— Respecter les contraintes fonctionnelles énoncées dans l'énoncé (sélection de la nourriture la plus fraîche, comportement face à la nourriture non fraîche, dispersion aléatoire, arrêt propre des threads, etc.).

3 Conception générale

3.1 Diagramme de classes

Image `diagrammeClassMMD.png` ou `diagrammeClass.pdf` manquante – placez le fichier dans le dossier `rapport/ressources/` pour l'inclure.

FIGURE 1 – Diagramme de classes (UML).

3.2 Structure des packages

La structure retenue pour le projet est la suivante (extrait du dépôt) :

— 'fr.pigeon.affichage' : classes UI (fenêtre, panneau de dessin, gestion des entrées utilisateur)
— 'fr.pigeon.entity' : modèles de domaine (Pigeon, Meal, Coordinate, Entity)
— 'fr.pigeon.multithreading' : classes de simulation et état global (Simulation, GameState)
— 'fr.pigeon.utils' : constantes et utilitaires

3.3 Contrat des composants (récapitulatif)

Pour chaque composant on précise brièvement le contrat (entrées, sorties, invariants, exceptions) :

- **Pigeon** : contrôle d'un pigeon (démarrage/arrêt du thread), méthode 'run()' pour gérer le comportement ; invariant : position toujours dans les limites de la fenêtre.
- **Meal** : position et niveau de fraicheur ; méthode 'isFresh()' ; suppression atomique lorsqu'un pigeon mange.
- **GameState** : contient listes de pigeons et de meals, méthodes synchronisées pour ajout/suppression et lecture pour le rendu.

3.4 Analyse des choix architecturaux

Ce sous-chapitre explique comment les principes d'ingénierie logicielle ont guidé les choix de conception.

SOLID

- **Single Responsibility** : chaque classe a une responsabilité unique — 'Display' s'occupe uniquement de l'affichage et des entrées, 'Pigeon' contient la logique comportementale, 'GameState' gère l'état partagé (pigeons + meals), 'Simulation' orchestre le cycle de vie des threads.
- **Open/Closed** : les fonctionnalités (ex : rasterisation SVG) sont ajoutées sous forme de modules optionnels ('SvgRasterizer') sans modifier le coeur du moteur de simulation, ce qui permet d'étendre le rendu sans modifier 'Display' de façon invasive.
- **Liskov** et **Interface Segregation** sont respectés par la séparation claire entre entités (domain) et affichage (UI).
- **Dependency Inversion** : les composants haut niveau dépendent d'abstractions (par ex. interfaces ou contrats simples) et non d'implémentations concrètes lorsque cela avait du sens.

Cohésion et couplage Les packages et classes sont conçus pour maximiser la cohésion (fonctions proches ensemble) et réduire le couplage (interfaces simples entre packages). Par exemple, 'GameState' expose des méthodes pour ajouter/retirer des 'Meal' et fournit une vue en lecture seule pour 'Display' afin d'éviter des dépendances croisées.

Généricité, Héritage et Délégation Le projet favorise la délégation sur l'héritage lorsque possible : les entités partagent une classe de base 'Entity' pour la position et le dessin minimaux, mais les comportements spécifiques (p.ex. 'Pigeon') sont encapsulés dans des classes dédiées. L'utilisation de génériques est limitée (pas nécessaire pour la taille du projet) mais la conception garde la porte ouverte pour des collections typées et des services réutilisables.

4 Programmation concurrente — détails d'implémentation

La partie la plus critique de la notation est la programmation concurrente. Cette section décrit précisément l'implémentation, les choix et les compromis.

4.1 Modèle d'exécution et cycle de vie des threads

Chaque pigeon est implémenté comme une classe 'Pigeon' qui implémente 'Runnable'. Lors de la création de la simulation, chaque 'Pigeon' reçoit son propre 'Thread' (référence conservée dans l'objet). Le cycle de vie suit ce schéma :

1. Création des instances 'Pigeon' et enregistrement dans 'GameState'.
2. Démarrage des threads (appel à 'pigeon.startThread()' qui crée et démarre le 'Thread').
3. Boucle principale dans 'run()' : calcul du mouvement, recherche de la cible (nourriture la plus fraîche), déplacement d'un pas, pause via 'Thread.sleep(stepMs)'.
4. Arrêt propre : 'Simulation.stop()' appelle 'stopThread()' sur chaque pigeon, ce qui met le drapeau 'running=false' et effectue un 'interrupt()' pour sortir d'un 'sleep' éventuel. Ensuite la simulation peut 'join()' sur les threads si nécessaire.

Voir le schéma correspondant en annexe : Schéma — cycle de vie d'un thread.

4.2 Mécanismes de communication et synchronisation

Le coeur du problème est le partage des ressources (liste de 'Meal') entre plusieurs threads de pigeons et le thread d'affichage (Swing EDT). Les choix faits :

- **Collections et stratégie d'accès** :
  - Pour les pigeons et autres entités, on utilise une collection qui favorise les lectures fréquentes durant le rendu (ex. 'CopyOnWriteArrayList' dans certains composants) — cela simplifie la gestion concurrente lorsque les écritures sont rares.
  - Pour les 'Meal', les opérations d'ajout (clic utilisateur) et de suppression (mangé / pourri) sont plus fréquentes et sensibles. La stratégie retenue consiste à protéger les modifications critiques par une synchronisation explicite sur 'GameState' (bloc 'synchronized') lors de l'itération et de la suppression.
- **Suppression atomique d'un meal** : lorsqu'un pigeon détecte qu'il est à portée d'un 'Meal' frais, il entre dans une section critique où il re-vérifie que le meal est toujours présent (n'a pas été supprimé par un autre pigeon) puis le retire. Cette vérification + suppression prennent place dans la même section 'synchronized' pour garantir qu'un seul pigeon supprime le meal.
- **Protection pendant le rendu** : Swing dessine sur l'EDT ; pendant le dessin, 'Display' prend une copie ou verrouille l'accès aux collections nécessaires (selon l'implémentation) pour éviter les ConcurrentModificationException. Le compromis consiste soit à utiliser des copies en lecture (bon pour la latence), soit à garder un verrou bref pendant la création de la vue à dessiner.

4.3 Exceptions, interruptions et robustesse des threads

— **InterruptedException** est attendue lors de l'arrêt des threads (interruption d'un 'sleep'). Le code attrape cette exception, restaure l'état d'interruption via 'Thread.currentThread().interrupt()' et termine proprement la boucle 'run()'.
— **Exceptions non contrôlées** : toute exception Runtime inattendue dans la boucle d'un thread est capturée et loggée de façon à éviter la terminaison silencieuse d'un thread. Selon la gravité, la thread peut tenter un redémarrage local ou signaler l'erreur au gestionnaire central ('Simulation') pour prise de décision.
— **Politique d'arrêt** : 'Simulation.stop()' est idempotent — il peut être appelé plusieurs fois sans effet secondaire, car chaque 'Pigeon' vérifie l'état avant d'agir. L'appel à 'interrupt()' accélère la sortie des threads en sommeil.

4.4 Décisions de conception et compromis

— **Un thread par pigeon vs thread pool** : pour la lisibilité et l'exercice pédagogique, nous avons choisi un thread par pigeon. Pour des simulations plus grandes (centaines / milliers d'entités), une architecture fondée sur un pool d'exécuteurs (ExecutorService) et des tâches périodiques serait plus performante et consommant moins de ressources.
— **Collections concurrentes vs verrou explicite** : les collections concurrentes (p.ex. 'ConcurrentLinkedQueue') réduisent la nécessité de verrous explicites mais complexifient certaines opérations atomiques (recherche + suppression) ; nous avons donc choisi la simplicité et la clarté pédagogique : verrous courts autour des sections critiques.
— **Consistance visuelle vs performances** : pendant le dessin, on accepte un léger biais de fraîcheur (lecture non verrouillée) pour garder une UI fluide ; les opérations d'écriture critiques restent protégées.

5 Gestion d'exceptions

La gestion d'exceptions a été pensée pour maximiser la robustesse et limiter la perte de service :

— Les points d'entrée des threads (méthode 'run') entourent le code par un bloc 'try/catch' capturant 'InterruptedException' et 'RuntimeException' pour éviter qu'un thread ne meure silencieusement.
— Les opérations d'E/S (chargement d'images, lecture de ressources) renvoient des erreurs non bloquantes : si le sprite PNG manque, le code tente de rasteriser le SVG, sinon un dessin de fallback (triangle) est utilisé. Ceci évite qu'une ressource manquante fasse planter l'application entière.
— Les erreurs fatales (ex : échec permanent d'initialisation) sont signalées à l'utilisateur via un message sur la console et une boite de dialogue, puis l'application s'arrête proprement.

6 Tests et validation

6.1 Tests automatiques

Un petit harness de tests (‘fr.pigeon.tests.TestSuite‘) est fourni et peut être exécuté par :

```
java -cp out fr.pigeon.tests.TestSuite
```

Les tests couvrent :
— Sélection du meal le plus frais (comportement A).
— Suppression sûre concurrente de ‘Meal‘ (comportement C).
— Arrêt propre des threads lors de la fin de la simulation (comportement B).

6.2 Tests manuels

Les tests manuels consistent à :
— Lancer l'application (‘make run‘) et ajouter des meals par clic, vérifier que les pigeons se dirigent vers la nourriture la plus fraîche.
— Vérifier que deux pigeons arrivant simultanément sur un meal ne provoquent pas de suppression double (un seul pigeon marque la nourriture comme mangée).
— Tester la fonctionnalité Restart et s'assurer que la touche Espace fonctionne après redémarrage (corriger le focus si nécessaire).

6.3 Résultats

À la date de rédaction, le harness de tests signale : **Tests passés : 9, Échecs : 0**. Les tests sont simples mais ciblent les aspects concurrents critiques.

7 Instructions pour exécuter

1. (Optionnel) créer et activer un venv pour l'outil de conversion SVG :
   — macOS / Linux :
   ```
   python3 -m venv .venv
   source .venv/bin/activate
   pip install -r requirements.txt
   ```

   — Windows (PowerShell) :
   ```
   python -m venv .venv
   .\.venv\Scripts\Activate.ps1
   pip install -r requirements.txt
   ```

2. Compiler : ‘make compile‘ (le Makefile tente la conversion SVG→PNG si le PNG manque).
3. Lancer : ‘make run‘.

8 Mode d'emploi interactif — comment jouer

Cette section remplace la version générique précédente par les commandes et comportements réellement implémentés dans l'application "Pigeon" (extraits du code source). Elle contient aussi des emplacements pour captures d'écran.

8.1 Lancer l'application

— Depuis la racine du projet, compiler et lancer : `make run` (ou exécuter la classe `fr.pigeon.Main` depuis votre IDE).
— La fenêtre s'ouvre avec la zone de simulation (taille fixe 800×600) et la barre de contrôle contenant le bouton `Restart`.

8.2 Commandes disponibles (clavier et UI)

Le code actuel expose les interactions suivantes :
— **Clic souris (zone de simulation)** : ajoute un `Meal` (nourriture) à la position du clic. Le panneau capture l'événement `mouseClicked` et appelle `simulation.addMeal(...)`. Note : cliquer sur le panneau donne également le focus afin que les raccourcis clavier fonctionnent.
— **Espace (SPACE)** : fait apparaître un pigeon aléatoirement dans la fenêtre (méthode `Simulation.spawnRandomPigeon(...)`). Cette action crée un nouvel objet `Pigeon`, démarre son thread et l'ajoute au `GameState`.
— **Bouton Restart (UI)** : le bouton `Restart` arrête proprement la simulation courante (appel à `Simulation.stop()`), crée une nouvelle instance `Simulation`, la lie au `Display` et démarre la nouvelle boucle de simulation dans un thread séparé.

Remarque : il n'y a pas de touche "Pause" implémentée dans le code source ni d'option `S` ou `Q` par défaut — ne vous fiez pas aux mentions génériques précédentes.

8.3 Comportements des pigeons (ce qui est codé)

Voici le comportement attendu et observé en fonction du code :
— **Démarche générale** : chaque pigeon est un objet `Pigeon` exécuté sur son propre `Thread`. La boucle `run()` effectue un pas de simulation (déplacement vers la cible si elle existe) puis dort 16 ms (60 Hz).
— **Recherche de nourriture** : si au moins un `Meal` est présent, chaque pigeon est mis en mode "vol" (`startPigeon()`) et se voit assigner comme cible la nourriture la plus fraîche (algorithme : choisir le `Meal` ayant la valeur maximale de `freshnessTicks`).
— **Arrêt / immobilté** : si aucune nourriture n'est présente, les pigeons sont mis au repos (`stopPigeon()`) et ne se déplacent plus jusqu'à l'apparition d'un `Meal`.
— **Collision / consommation** : lorsque la distance entre un pigeon et un `Meal` est inférieure au rayon d'interaction, la méthode `eatMeal(pigeon, meal)` est appelée. La suppression du `Meal` est faite dans un bloc `synchronized` sur le `GameState` afin d'assurer l'atomicité : un seul pigeon réussit à supprimer et à consommer le repas (son compteur `mealsEaten` est incrémenté).
— **Dégradation de la nourriture** : à chaque étape de simulation, la fraîcheur des `Meal` est décrémentée (`decreaseFreshness()`). Lorsqu'un `Meal` n'est plus frais, il

est retiré de la liste active et déplacé dans `rottenMeals` (affiché en rouge dans l'UI).

— **Peur / dispersion** : le `GameState` maintient une probabilité dynamique `scareChance` ; de façon aléatoire, la méthode `dispersePigeons()` est invoquée. Elle appelle `pigeon.setAfraid(` choisit une position aléatoire pour chaque pigeon et force le pigeon à voler vers cette nouvelle cible. L'état de peur dure un temps défini `FEAR_DURATION` dans `Pigeon`. Après expiration, le pigeon redevient calme.

8.4 Indications visuelles dans l'interface

Dans le rendu fourni par `Display` et `GameState` :

— Les **pigeons** sont dessinés en bleu (ou par un sprite si l'image est disponible). Leur orientation est tournée vers la cible lorsqu'ils ont un target.

— Les **meals** frais sont dessinés en vert ; les **meals** périmés (rotten) sont dessinés en rouge (liste `rottenMeals`).

— La légende dans le coin supérieur gauche indique les couleurs et quelques raccourcis (affichée par `drawLegend`).

8.5 Scénarios d'observation (guidés)

Scénario 1 — État initial

1. Lancer l'application (`make run`).
2. Observation : la fenêtre s'ouvre et les pigeons (s'ils existent à l'initialisation) peuvent être immobiles si aucune nourriture n'est présente.
3. Capture d'écran : déposer `ressources/screenshot_initial.png` pour l'illustration.

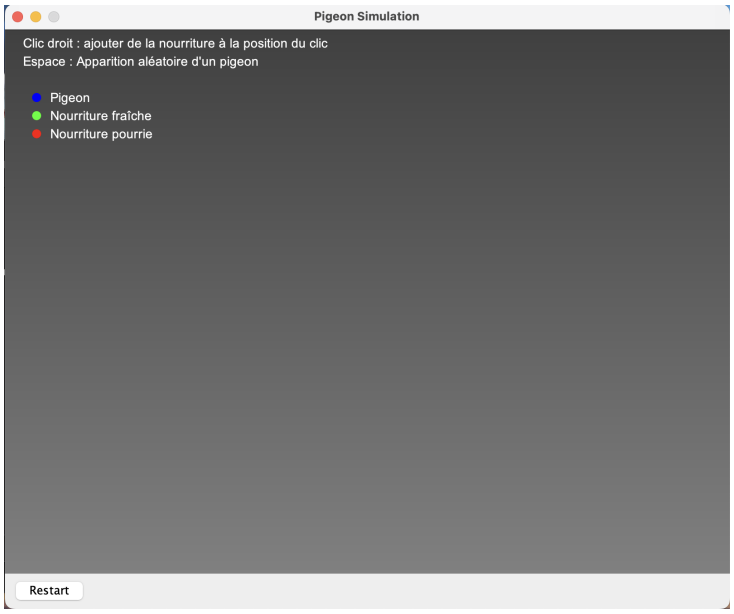

FIGURE 2 – État initial de la simulation (placeholder).

Scénario 2 — Ajouter une nourriture et observer la course

1. Cliquer (mouse click) dans la zone de simulation pour créer un `Meal`.
2. Observation : un cercle vert apparaît à l'emplacement ; dans les frames suivantes, les pigeons calculent la nourriture la plus fraîche et se dirigent vers elle.
3. Capture d'écran : déposer `ressources/screenshot_add_meal.png`.

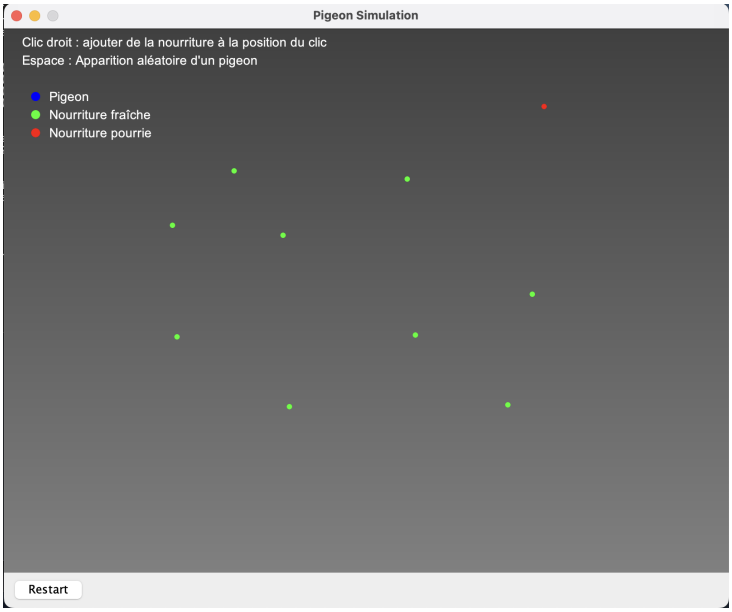

FIGURE 3 – Après ajout d'un meal (placeholder).

Scénario 3 — Course concurrente vers un même meal

1. Placer un `Meal` entre deux pigeons ou appuyer plusieurs fois pour créer des pigeons proches et un meal partagé.
2. Observation : plusieurs pigeons peuvent se diriger vers le même repas ; lorsqu'ils arrivent, la suppression se fait dans une section synchronisée. Un seul pigeon devrait réussir à supprimer/consommer le meal (les autres détectent l'absence et se redirigent).
3. Capture d'écran : déposer `ressources/screenshot_race.png`.

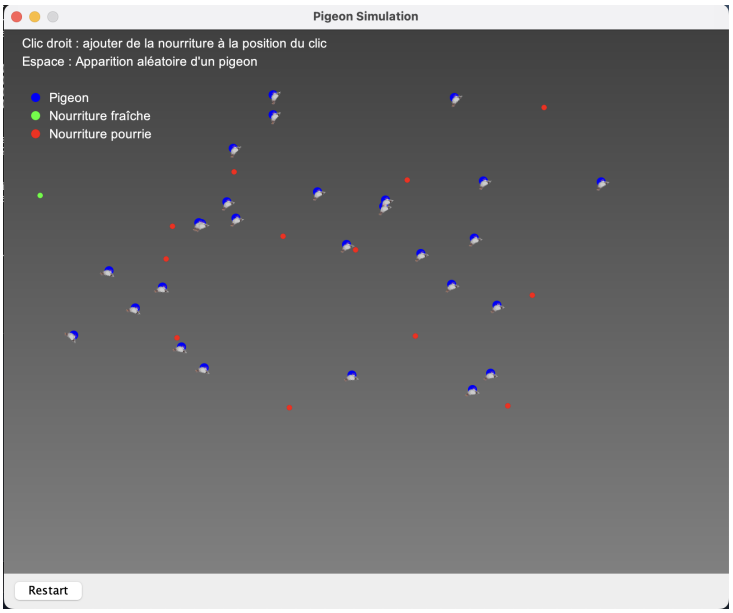

FIGURE 4 – Course de plusieurs pigeons vers un même meal (placeholder).

Scénario 4 — Décroissance des meals

1. Laisser un meal sur place et observer plusieurs étapes de la simulation.
2. Observation : la fraîcheur diminue via `decreaseFreshness()` ; quand un meal n'est plus frais il est retiré des `meals` actifs et placé dans `rottenMeals` (affiché en rouge).
3. Capture d'écran : déposer une image montrant un meal périmé si disponible.

Scénario 5 — Restart et génération de pigeons

1. Cliquer sur le bouton `Restart` en bas de la fenêtre : la simulation courante est arrêtée proprement et une nouvelle simulation démarre (les listes de pigeons/meals sont réinitialisées).
2. Après un restart, s'assurer que le panneau a le focus (cliquer dans la zone) puis appuyer sur `SPACE` pour générer un pigeon aléatoire.
3. Capture d'écran : avant/après restart (placer les fichiers correspondants dans `ressources/`).

8.6 Que surveiller (indicateurs de correct fonctionnement)

— **Suppression atomique** : un seul pigeon doit consommer chaque meal (vérifier `mealsEaten` et l'absence de duplicates).
— **Absence d'erreurs de concurrence** : pas de `ConcurrentModificationException` pendant le rendu ou les mises à jour (les collections sont manipulées de façon sûre).
— **Threads propres** : après un `Restart`, les threads de la vieille simulation sont interrompus et la nouvelle simulation démarre ses propres threads.
— **Indicateurs visuels** : meals frais = vert, meals périmés = rouge, pigeons = bleu ou sprite si présent.

A Extraits de code importants

A.1 Sélection de la nourriture la plus fraîche

Extrait (simplifié) :

```
Meal findFreshest(Collection<Meal> meals) {
  Meal best = null;
  for (Meal m : meals) {
    if (m.isFresh() && (best == null || m.getFreshness() > best.getFreshnes
      best = m;
    }
  }
  return best;
}
```

A.2 Suppression sûre d'un meal (schéma)

```
synchronized(gameState) {
  if (gameState.contains(meal) && meal.isFresh()) {
    gameState.remove(meal);
  }
}
```

A.3 Schéma — cycle de vie d'un thread

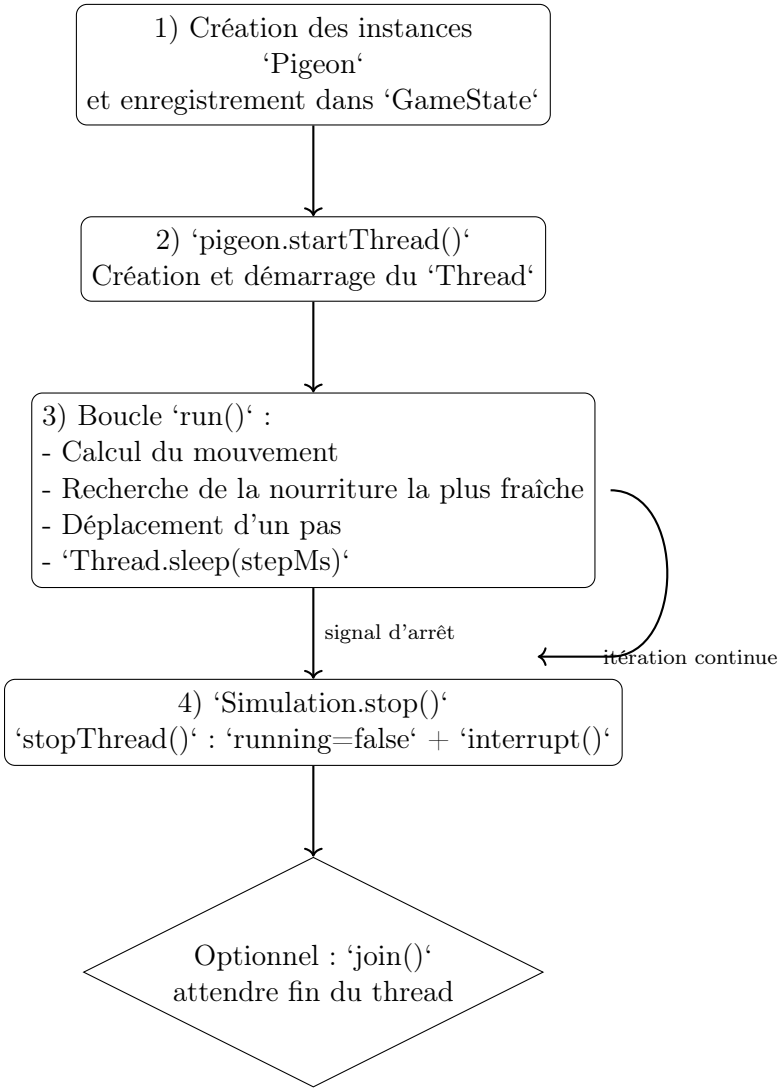

FIGURE 5 – Cycle de vie d'un thread gérant un pigeon — du démarrage à l'arrêt propre.

Fait par : Sophie, Manon et Rolland
Date : 7 novembre 2025