(CS534)

# Compte Rendu Systeme Multi Agents

IR&C - ESISAR (2024-2025)


Elève Ingénieur: **Dorra BEN HAJ KALBOUSSI**
Classe: 5e Année IRC
E-mail: *dorra.ben-haj-kalboussi@etu.esisar.grenoble-inp.fr*

Elève Ingénieur: **Mohamed Yassine BAGHDADI**
Classe: 5e Année IRC
E-mail: *mohamed-yassine.baghdadi@etu.esisar.grenoble-inp.fr*

Tuteur: **Mr. Clement Raievsky**
E-mail:
*clement.raievsky@lcis.grenoble-inp.fr*


November 25, 2024

# Contents

# Introduction Générale

Les systèmes multi-agents (MAS, pour Multi-Agent Systems) sont une approche innovante et puissante dans le domaine de l'informatique, inspirée par des systèmes naturels comme les colonies de fourmis, les essaims d'abeilles ou encore les réseaux sociaux humains. Ces systèmes sont composés d'entités autonomes, appelées agents, qui interagissent entre eux et avec leur environnement pour accomplir des objectifs communs ou individuels. Dans un contexte où la résolution de problèmes complexes nécessite une collaboration décentralisée, les MAS se distinguent par leur capacité à reproduire des comportements émergents à partir d'interactions simples. Ils trouvent des applications variées dans des domaines tels que :

- La robotique collective.
- Les simulations de trafic urbain.
- Les jeux vidéo.
- Les systèmes de recommandation.

# 1 Mise en contexte: Systeme Multi Agents

## 1.1 Qu'est-ce qu'un système multi-agent ?

Un système multi-agent est une organisation logicielle composée d'agents autonomes qui interagissent pour atteindre des objectifs spécifiques. Ces agents peuvent être définis par:

1. **L'autonomie :** chaque agent peut prendre des décisions indépendamment des autres.
2. **L'interaction :** les agents collaborent ou entrent en compétition via des messages ou des actions communes.
3. **L'adaptabilité :** ils sont capables de modifier leur comportement en fonction des changements dans l'environnement.
4. **La coopération / la compétition :** certains agents travaillent ensemble, tandis que d'autres poursuivent des objectifs divergents.

Les MAS permettent de modéliser des systèmes distribués où il n'existe pas de contrôle centralisé, rendant les simulations réalistes pour les environnements complexes.

## 1.2 Pourquoi utiliser une plateforme comme MaDKit ?

MaDKit (Multi-Agent Development Kit) est une plateforme dédiée au développement de systèmes multi-agents. Cette bibliothèque Java offre un cadre structuré et des outils intégrés pour créer, simuler, et analyser les comportements d'agents dans des environnements diversifiés.
**Les avantages de MaDKit incluent :**

1. **Modularité** : les développeurs peuvent structurer leurs projets autour des concepts de rôles, groupes et agents, facilitant ainsi la gestion et la réutilisation du code.
2. **Facilité de communication :** des fonctionnalités natives permettent d'envoyer des messages entre agents et de définir des protocoles d'interaction.
3. **Évolutivité :** la plateforme peut gérer un grand nombre d'agents en parallèle, ce qui est essentiel pour simuler des systèmes à grande échelle.
4. **Simplicité d'intégration :** étant basé sur Java, MaDKit s'intègre facilement dans des projets existants et bénéficie d'un large écosystème d'outils et de bibliothèques.

## 1.3 Objectifs du LAB

Le laboratoire visait à initier les participants à la conception et à la simulation de systèmes multi-agents. Les objectifs principaux incluaient :

### 1.Comprendre les concepts fondamentaux des MAS :

- Définir les rôles, les groupes et les agents.
- Observer les interactions et leur impact sur l'émergence de comportements complexes.

### 2. Apprendre à utiliser MaDKit :

- Créer des agents autonomes et adaptatifs.
- Implémenter des systèmes où la communication est au cœur des interactions.

### 3. Simuler des environnements interactifs :

- Concevoir des agents capables de collaborer ou de rivaliser pour atteindre des objectifs.
- Étudier les comportements émergents dans des simulations complexes, comme les colonies d'insectes ou les systèmes économiques.

### 4.Développer des compétences pratiques :

- Appliquer des concepts théoriques dans un environnement de développement structuré.
- Analyser les résultats des simulations pour en tirer des conclusions sur les comportements des agents.

# 2 Explication de la première communication dans MaDKit

## 2.1 Création d'une société artificielle

Pour que les agents puissent interagir, ils doivent exister dans une société artificielle qui se structure en :

- **Communautés:** Un regroupement logique d'agents partageant un objectif commun (ici, "communication").
- **Groupes:** Des sous-ensembles au sein d'une communauté (ici, "ex01").
- **Rôles:** Les positions que les agents prennent pour se différencier et interagir (ici, "ping agent").

Dans la classe Society, ces constantes *(COMMUNITY, GROUP, ROLE)* définissent cette structure.

## 2.2 Activation de l'agent (activate)

La méthode activate configure l'agent dans la société :

- Création d'un groupe :*createGroup(COMMUNITY, GROUP);*
  Si le groupe n'existe pas, il est créé. Cela permet d'organiser les agents.
- Demande d'un rôle : *requestRole(COMMUNITY, GROUP, ROLE);* L'agent se voit attribuer un rôle dans le groupe. Cela lui permet d'interagir avec d'autres agents ayant le même rôle.

## 2.3 Recherche et communication dans live

L'agent commence à vivre et à interagir :

- **Recherche d'un autre agent:** *other = getAgentWithRole(COMMUNITY, GROUP, ROLE);* -Cette méthode permet à l'agent de trouver un autre agent qui a pris le même rôle dans la société.
  -La recherche est effectuée en boucle jusqu'à ce qu'un agent soit trouvé.
  -Une fois trouvé, son adresse est stockée dans AgentAddress other, qui représente une "carte d'identité" permettant de le contacter.
- **Envoi d'un message:** sendMessage(other, new Message());
  -Une fois l'adresse de l'autre agent obtenue, l'agent envoie un message.
  -Ici, le message est une instance de la classe Message. C'est une structure de base dans MaDKit qui peut être personnalisée pour transporter des données spécifiques.
- **Attente de réponse:** *waitNextMessage();*
  -Après avoir envoyé un message, l'agent attend de recevoir un message en retour.
  -Cette méthode suspend l'exécution jusqu'à ce qu'un message arrive.

### 2.4 Lancement de l'agent (main)

La méthode main permet de lancer deux instances de PingAgent :
*executeThisAgent(2, true);*

- Cela signifie que deux agents PingAgent seront créés et exécutés.
- Les agents communiqueront entre eux en échangeant des messages.

## 3 Échange de messages en ciblant les rôles avec PingAgent2

Dans cette section, nous présentons le fonctionnement de l'agent `PingAgent2`, qui illustre une méthode de communication au sein d'un système multi-agents en ciblant un rôle spécifique au lieu d'un agent particulier. Cette approche exploite la structure organisationnelle des agents, définie en termes de communautés, groupes et rôles.

### 3.1 Principe de fonctionnement

L'agent `PingAgent2` utilise la méthode `sendMessage()` pour envoyer des messages à des agents occupant un rôle spécifique (`ROLE`) dans un groupe (`GROUP`) d'une communauté (`COMMUNITY`). Cette communication est indépendante de l'identité réelle des agents récepteurs.

- **Recherche de destinataires :** L'agent tente d'envoyer un message de manière répétée jusqu'à ce qu'il trouve un autre agent possédant le rôle cible. Cette vérification repose sur la valeur de retour de la méthode `sendMessage()`, qui indique si l'envoi a réussi.
- **Traitement des messages :** Une fois un message envoyé avec succès, l'agent utilise `nextMessage()` pour traiter les messages reçus.

### 3.2 Lancement et simulation

Dans la méthode `main()`, plusieurs instances de `PingAgent2` et `PingAgent` sont créées à l'aide de la commande suivante :

```
new Madkit("--launchAgents", PingAgent2.class.getName() + ",true,3;",
           PingAgent.class.getName());
```

Cette configuration garantit qu'au moins un agent est toujours disponible pour recevoir des messages, ce qui permet à l'agent `PingAgent2` de sortir de sa boucle d'attente.

### 3.3 Avantages de la communication basée sur les rôles

Cibler les rôles plutôt que les identités spécifiques des agents offre plusieurs avantages :

- **Flexibilité :** Les agents peuvent rejoindre ou quitter les groupes sans perturber l'organisation globale.
- **Abstraction :** Il n'est pas nécessaire de connaître l'identité exacte des récepteurs pour envoyer un message, simplifiant ainsi la logique de communication.

### Conclusion

Le programme `PingAgent2` met en œuvre un concept fondamental des systèmes multi-agents : la communication par rôles. Cette approche est particulièrement adaptée aux environnements dynamiques où les agents interagissent en fonction de leurs responsabilités au sein du système.

# 4 Jeux de rôle dans une société multi-agents

Cette section met en œuvre une interaction dynamique entre trois types d'agents dans une société multi-agents, utilisant la plateforme MaDKit. L'objectif est de démontrer comment les rôles des agents peuvent évoluer en fonction des messages échangés et des règles définies.

## 4.1 Présentation des rôles des agents

Trois types d'agents interagissent dans le système :

- **Agents émetteurs (`EmetteurAgent`) :**
  - **Rôle :** Envoyer un nombre aléatoire de messages à des agents compteurs.
  - **Comportement :**
    1. Rejoignent la société avec le rôle `emetteur` dans le groupe `GroupTest` de la communauté `communication`.
    2. Envoient un nombre aléatoire de messages (entre 1 et 5) à des agents `CounterAgent`.
    3. Terminent leur cycle de vie via la méthode `killAgent`.
- **Agents compteurs (`CounterAgent`) :**
  - **Rôle :** Compter les messages reçus des agents émetteurs.
  - **Comportement :**
    1. Prennent le rôle `conteur` à l'activation.
    2. Incrémentent un compteur interne pour chaque message reçu.
    3. Lorsqu'ils atteignent 5 messages :
       * Notifient un agent `ControllerAgent` via un message.
       * Se transforment en `EmetteurAgent` via `launchAgent`.
       * Se terminent eux-mêmes avec `killAgent`.
- **Agents contrôleurs (`ControllerAgent`) :**
  - **Rôle :** Maintenir un nombre constant d'agents compteurs.
  - **Comportement :**
    1. Supervisent les messages des `CounterAgent`.
    2. Lorsqu'un `CounterAgent` informe de sa transformation en `EmetteurAgent`, le contrôleur crée un nouveau `CounterAgent` avec `launchAgent`.

### 4.2 Interactions dynamiques et exécution

1. **Initialisation :** Le système démarre avec un `ControllerAgent`, un `EmetteurAgent`, et un `CounterAgent`.
2. **Envoi de messages :** Les `EmetteurAgent` envoient des messages aux `CounterAgent`, qui incrémentent leur compteur interne.
3. **Transformation des rôles :** Lorsque le compteur d'un `CounterAgent` atteint 5, il :
   - Informe le `ControllerAgent`.
   - Crée un nouvel `EmetteurAgent`.
   - Termine son cycle de vie.
4. **Supervision par le contrôleur :** Le `ControllerAgent` crée un nouveau `CounterAgent` pour remplacer celui qui s'est transformé.

### Conclusion

Cette section illustre l'évolution dynamique des rôles et la gestion adaptative dans une société multi-agents. Il démontre des concepts clés tels que :

- La communication par messages à travers des objets `Message`.
- La supervision et la régulation par un agent contrôleur.
- L'évolution dynamique des rôles avec `launchAgent` et `killAgent`.

Ce système offre une base solide pour explorer la gestion des rôles dans des systèmes distribués complexes.

## 5 Simulation des abeilles

### 5.1 Résumé des classes principales de la solution

La simulation repose sur plusieurs classes importantes, définies comme suit :

- **AbstractBee** : Classe abstraite représentant une abeille générique, utilisée comme base pour les abeilles standards et les reines. Elle contient des comportements communs tels que les déplacements et les interactions avec l'environnement.
- **Bee** : Classe dérivée de `AbstractBee`, représentant une abeille standard capable de collecter des ressources et d'interagir avec l'environnement.
- **QueenBee** : Classe héritant de `AbstractBee`, représentant une reine avec un comportement spécifique, notamment la ponte d'œufs pour générer de nouvelles abeilles.
- **BeeEnvironment** : Modélise l'environnement dans lequel évoluent les abeilles. Elle gère les ressources disponibles et les interactions des agents.

- **BeeScheduler** : Responsable du cycle de vie des abeilles, incluant leur création et leur destruction selon les paramètres définis.
- **BeeViewer** : Gère le rendu graphique de la simulation, permettant de visualiser les abeilles et l'environnement.

## 5.2 Modifications proposées

### 5.2.1 Affichage des reines sous forme de cercles

Pour distinguer visuellement les reines des abeilles standard, une modification de la méthode `computeFromInfoProbe` dans la classe `BeeViewer` est proposée :

Listing 1: Modification de la méthode computeFromInfoProbe

```
private void computeFromInfoProbe(Graphics g) {
    g.drawString("You are watching " + beeProbe.size() + " MaDKit agents",
    Color lastColor = null;
    final boolean trailMode = (Boolean) trailModeAction.getValue(Action.SEL
    for (final AbstractBee arg0 : beeProbe.getCurrentAgentsList()) {
        final BeeInformation b = beeProbe.getPropertyValue(arg0);
        final Color c = b.getBeeColor();
        if (arg0 instanceof QueenBee) { // V rifie si l'abeille est une re
            final Point p = b.getCurrentPosition();
            g.setColor(Color.RED); // Couleur rouge pour les reines
            g.fillOval(p.x - 5, p.y - 5, 10, 10); // Dessine un cercle pour
        } else {
            if (c != lastColor) {
                lastColor = c;
                g.setColor(lastColor);
            }
            final Point p = b.getCurrentPosition();
            if (trailMode) {
                final Point p1 = b.getPreviousPosition();
                g.drawLine(p1.x, p1.y, p.x, p.y);
            } else {
                g.drawLine(p.x, p.y, p.x, p.y);
            }
        }
    }
}
```

### 5.2.2 Paramètres de génération et de destruction

Pour améliorer la lisibilité et la gestion des ressources, les ajustements suivants sont apportés :

- **Réduction du nombre d'abeilles générées** : Modifiez la méthode de génération dans `BeeScheduler` :

Listing 2: Réduction du nombre d'abeilles générées

```
int nbBees = random.nextInt(5) + 1; // Génère entre 1 et 5 abeilles
```

- **Augmentation de la durée de vie des reines** : Dans la classe `QueenBee`, augmentez les cycles de vie pour prolonger leur présence.

### 5.2.3 Amélioration de l'affichage graphique

Ajout d'un fond clair pour une meilleure lisibilité des captures d'écran :

Listing 3: Ajout d'un fond clair

```
display.setBackground(Color.GRAY);
display.setForeground(Color.WHITE);
```

## 5.3 Résultats obtenus

Après les modifications, les améliorations suivantes sont attendues :

- Les reines sont visibles sous forme de cercles rouges distincts.
- Moins d'abeilles sont générées, rendant la simulation plus lisible.
- Les reines vivent plus longtemps, facilitant l'observation de leur comportement.
- Le fond clair améliore la lisibilité pour les rapports et présentations.

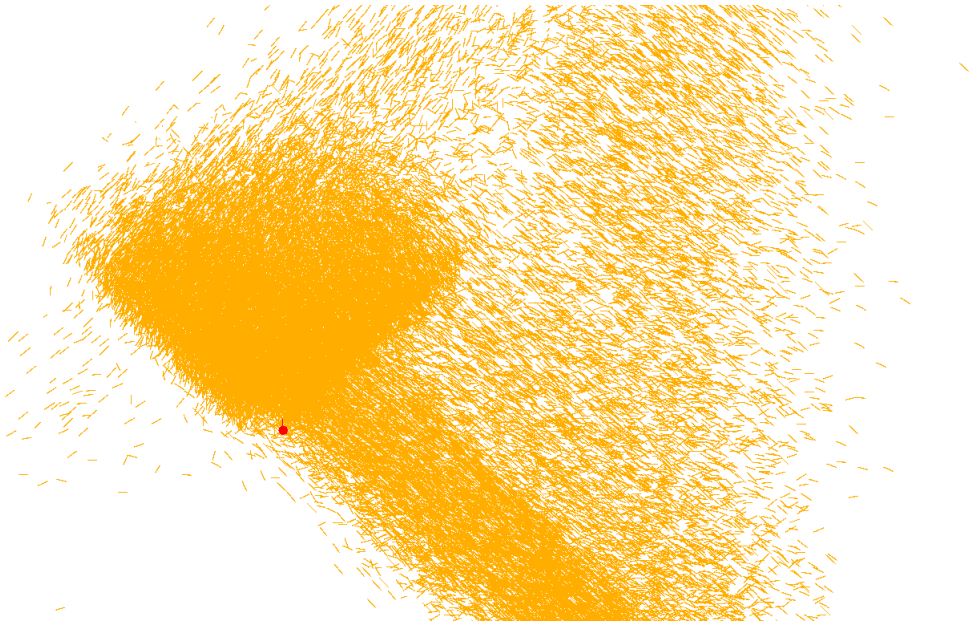

Figure 1: Visualisation des résultats obtenus

# 6 Implémentation des classes Bee et Hornet

Les classes Bee et Hornet étendent la classe abstraite AbstractBee et modélisent respectivement les comportements des abeilles et des frelons dans une simulation. Elles gèrent les interactions entre les abeilles, leur déplacement, leur coordination ainsi que leurs réponses face aux menaces, telles que les attaques des frelons. La classe Bee se concentre sur le comportement social et défensif des abeilles, tandis que la classe Hornet représente un prédateur qui menace les abeilles en simulant des attaques.

## 6.1 Obstacles rencontrés

- Gérer les collisions possibles dans les messages envoyés entre plusieurs agents a ajouté de la complexité.
- Vérifier en continu le nombre d'abeilles dans le rayon du frelon tout en minimisant l'impact sur les performances a été un défi.
- Gérer les interactions entre le comportement défensif des abeilles et les conditions d'attaque du frelon a nécessité une structure de logique conditionnelle plus complexe.
- Concevoir un système efficace de communication inter-agents tout en évitant des conflits ou des redondances a pris du temps.

Pour surmonter ces difficultés, les classes Bee et Hornet ont été conçues avec une logique bien structurée, permettant de simuler les comportements attendus tout en intégrant des mécanismes de communication et de gestion des actions différées.

## 6.2 Classe Bee

La classe Bee est responsable de la gestion du comportement des abeilles dans la simulation. Elle assure la coordination avec les autres abeilles et la gestion des menaces telles que les attaques de frelons.

### 6.2.1 Attributs Clés

Les principaux attributs de la classe Bee sont :

- `BeeInformation leaderInfo` : Contient les informations relatives au leader (généralement l'abeille reine).
- `AgentAddress leader` : Adresse de l'agent leader actuel.
- `boolean stop` : Indique si l'abeille doit s'arrêter à la suite d'une attaque.

### 6.2.2 Méthodes Principales

Les principales méthodes de la classe Bee sont :

- `activate()` : Initialise l'abeille avec les rôles `bee` et `follower` dans le groupe de simulation `SIMUGROUP`.

- `buzz()` : Traite les messages reçus. Si le message est `"STOP"`, l'abeille s'arrête et coordonne une défense avec les autres abeilles. Si le message est `"KILL"`, l'abeille est éliminée. Dans les autres cas, elle met à jour son leader.
- `computeNewVelocities()` : Calcule les nouvelles vitesses de l'abeille, soit en suivant le leader, soit de manière aléatoire.
- `updateLeader(ObjectMessage<BeeInformation> m)` : Permet de mettre à jour les informations relatives au leader, avec un changement de leader pouvant être aléatoire.
- `followNewLeader(ObjectMessage<BeeInformation> leaderMessage)` : Permet à l'abeille de suivre un nouveau leader et d'appliquer les nouvelles informations de ce leader.

## 6.3 Classe `Hornet`

La classe `Hornet` modélise un prédateur qui attaque les abeilles. Elle gère des mécanismes d'attaque, de défense et de déplacement en fonction des abeilles détectées.

### 6.3.1 Attributs Clés

Les principaux attributs de la classe `Hornet` sont :

- `target` : Cible actuelle du frelon.
- `killCounter` : Compteur du nombre d'abeilles tuées par le frelon.
- `beeSurroundTimer` : Chronomètre pour détecter un encerclement par les abeilles.
- `targetSurroundTimer` : Chronomètre pour mesurer la durée de l'attaque sur une cible.
- `minDist` : Distance minimale pour détecter une abeille comme étant proche.

### 6.3.2 Méthodes principales

Les principales méthodes de la classe `Hornet` sont :

- `activate()` : Initialise le frelon dans le groupe de simulation en attribuant les rôles `HORNET` et `bee`.
- `findnearest()` : Calcule la distance entre le frelon et les autres abeilles afin de déterminer la plus proche.
- `detecterAbeillesProches()` : Identifie les abeilles situées à proximité dans un rayon défini par `minDist`.
- `gererMessagesEntrants()` : Gère les messages entrants. Si le message `"SURROUNDED"` est reçu, le frelon se suicide.
- `verifierEssaiAbeilles()` : Vérifie si le frelon est encerclé par un grand nombre d'abeilles et se tue dans ce cas.

- `gererAttaque()` : Effectue une attaque contre les abeilles proches en envoyant les messages `"STOP"` et `"KILL"`.
- `computeNewVelocities()` : Ajuste la vitesse du frelon en fonction de la position de sa cible.
- `getMaxVelocity()` : Retourne la vitesse maximale du frelon.

### 6.3.3 Résumé des interactions

La méthode `buzz()` est au cœur des actions du frelon. Elle gère la réception des messages entrants, la sélection de la cible, la détection des abeilles proches, l'évaluation de l'encerclement, et enfin l'attaque des abeilles lorsque les conditions sont réunies.

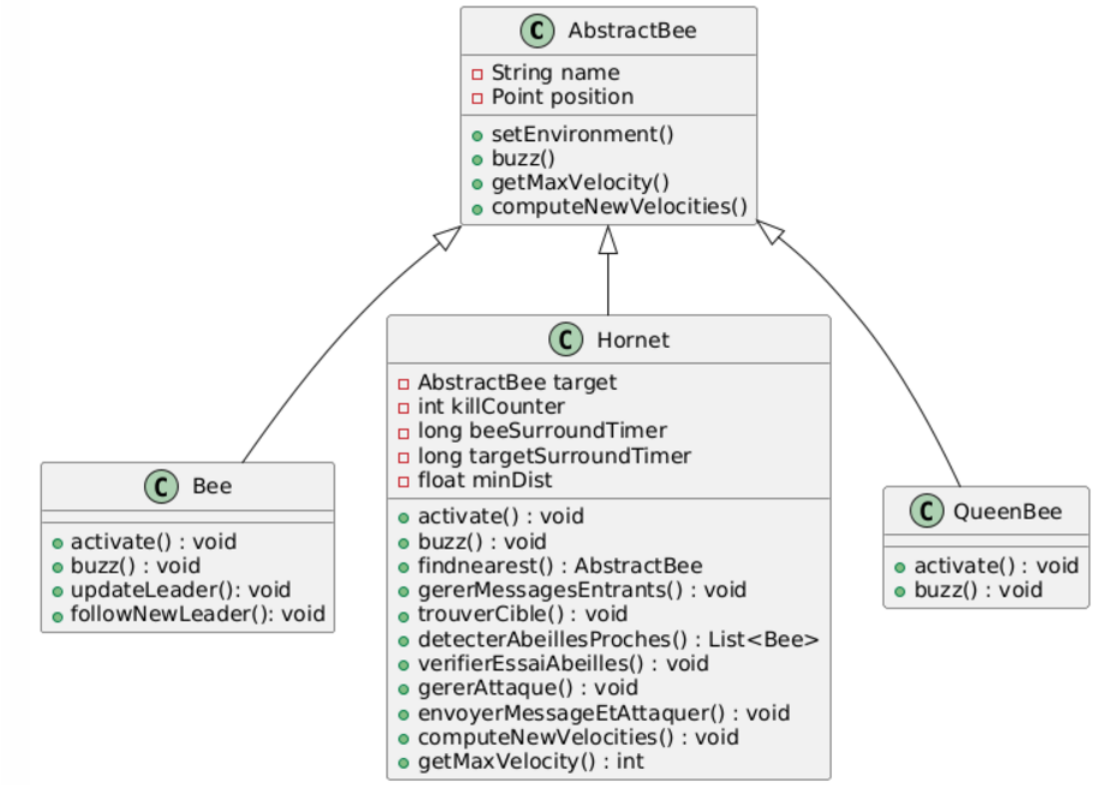


Figure 2: Diagramme de classes

## Conclusion

Les classes `Bee` et `Hornet` permettent de simuler des interactions complexes entre les abeilles et les frelons. La classe `Bee` est conçue pour gérer les comportements sociaux des abeilles, telles que la défense collective et le suivi du leader. La classe `Hornet` modélise un prédateur réactif qui attaque les abeilles tout en étant vulnérable à une défense collective. Ensemble, ces deux classes illustrent l'importance des interactions locales et des règles simples mais efficaces pour modéliser des comportements complexes dans une simulation.

# 7 Éléments d'apprentissage abordés

- **Compréhension de l'héritage** : Nous avons approfondi notre maîtrise de l'héritage en créant des classes avec des relations d'héritage, comme entre `Hornet`, `Bee` et `AbstractBee`.
- **Conception de systèmes multi-agents** : Nous avons exploré les systèmes multi-agents en modélisant des interactions entre agents (`Hornet`, `Bee`), comme l'attaque et la détection d'abeilles proches.
- **Gestion des comportements et interactions** : Nous avons géré des comportements dynamiques, tels que les déplacements et les actions des agents.