PROJET - JAVA

Modelisation d'un écosystème

Élèves :

Yacoub MESBAHI

Aïssa PANSAN

Enseignant :

John CHAUSSARD

Pierre FOUILLOUX

16 avril 2023

Sommaire

1 Introduction générale

Nous vous présentons notre rapport sur le projet de modélisation d'un écosystème en Java. Ce projet s'inscrit dans le cadre de notre formation d'ingénieur en tant qu'étudiants en première année d'ingénierie en informatique, mais aussi de notre cours de programmation orienté objet.

Nous sommes fiers de partager avec vous les résultats de notre travail et espérons qu'il démontrera notre compréhension des concepts que l'on a étudiés durant ce semestre.

Le but de ce projet est de modéliser et de simuler de manière réaliste l'évolution d'un écosystème. L'environnement est découpé en plusieurs zones qui peuvent être de plusieurs types (Foret, Désert, Plaine) et composé d'un biome très disparate (Mammifères, Insectes, Oiseaux, Plantes et Arbres). L'objectif de notre programme est de réussir à coder de manière cohérente le biome. Il est aussi nécessaire de programmer les différentes actions inhérentes à chaque animal ou végétaux, du plus simple comme l'action de boire ou plus complexe comme le fait de manger un autre animal. Enfin, il est tout autant nécessaire de trouver la meilleure représentation graphique du déroulement de la modélisation, par le biais de la réalisation d'un moteur graphique.

Enfin, ce rapport présente les détails de notre travail, expliquant les décisions prises et les difficultés rencontrées durant son élaboration. Il a pour objectif de synthétiser notre démarche de résolution en détaillant succinctement les étapes clé du projet.

2 Réalisation du moteur physique

2.1 Diagramme de classes

Notre programme se divise en deux parties, un moteur physique qui représente l'ensemble de la modélisation et un moteur graphique qui va se charger d'afficher le produit du moteur physique. Dans cette partie, noue détaillerons succinctement les différentes classes composant le moteur graphique.

2.2 Ecosysteme et bases

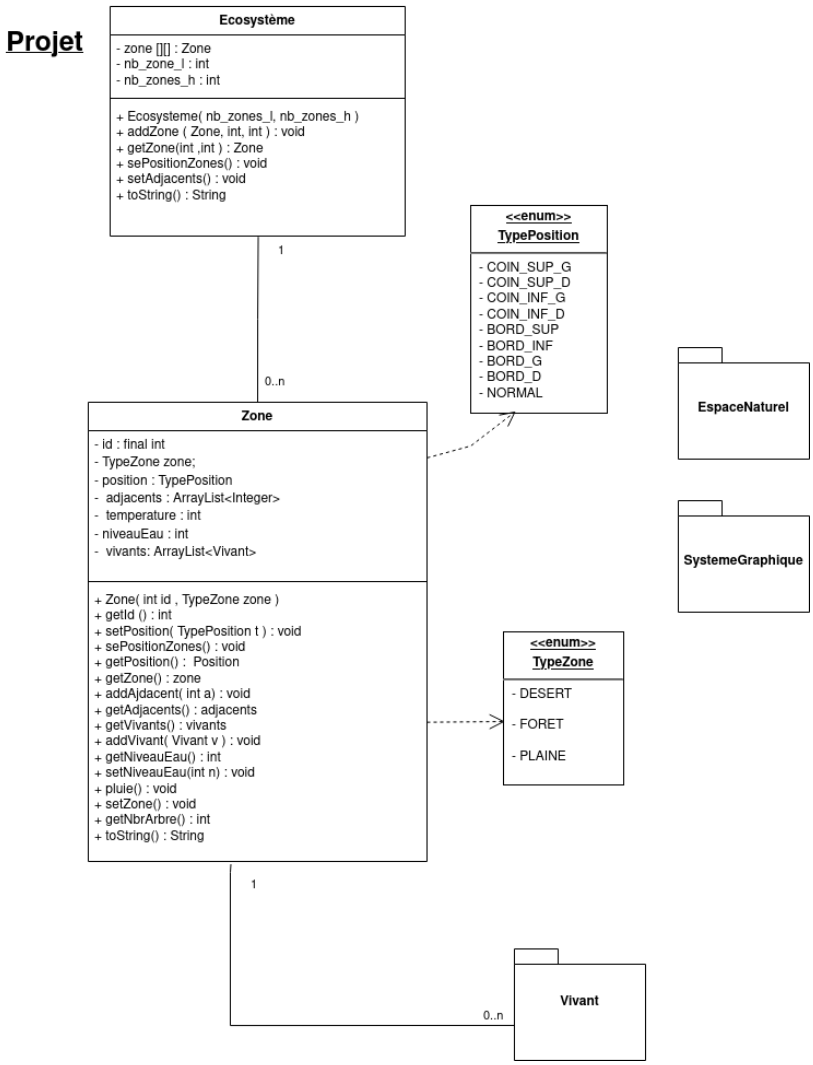

FIGURE 1 – Diagramme de classes - Écosystème et zones

Nous avons implémenté une classe écosystème qui sera associé à plusieurs zones, ces zones vont représenter l'espace de vie des arbres et animaux. Une zone est par ailleurs constituée d'un id, qui sera indispensable pour le positionnement. Une zone peut aussi être de 3 types, chaque type a des caractéristiques intrinsèques en termes de température et de d'eau. Ainsi, pour le début de notre modélisation, nous avons défini que :

— Forêt : Température : De 12 à 25°C / Eau : 100%-75%

— Plaine : Température : De 18 à 35°C / Eau : 75%-35%

— Forêt : Température : De 40 à 55°C / Eau : 5%-35%

La zone implémente une ArrayList de *Vivants* cette classe sera explicitée ultérieurement dans ce rapport

2.2.1 Analyse détaille du système de déplacement

Pour garantir une certaine une simplicité d'utilisation à travers les différentes classes, nous avons mis au point un système de coordonnée par un repère personnalisée.

FIGURE 2 – Représentation du système de déplacement

Nous découpons les grilles de notre écosystème suivant 9 zones (Correspondant aux valeurs prise dans «enum» : TypePosition. La méthode *SetZone()* permet ainsi d'associer à chaque case son type de zone correspondant.

```
public void setPositionZones() {
    for (int i = 0; i < nb_zones_l; i++) {
        for (int j = 0; j < nb_zones_h; j++) {
            if (i == 0 && j == 0)
                zones[i][j].setPosition(TypePosition.COIN_SUP_G);
            else if (i == nb_zones_l-1 && j == 0)
                zones[i][j].setPosition(TypePosition.COIN_SUP_D);
```

```
            else if (i == 0 && j == nb_zones_h-1)
                zones[i][j].setPosition(TypePosition.COIN_INF_G);
            else if (i == nb_zones_l-1 && j == nb_zones_h-1)
                zones[i][j].setPosition(TypePosition.COIN_INF_D);
            else if (i == 0) zones[i][j].setPosition(TypePosition.BORD_G);
            else if (i == nb_zones_l-1)
                zones[i][j].setPosition(TypePosition.BORD_D);
            else if (j == 0) zones[i][j].setPosition(TypePosition.BORD_SUP);
            else if (j == nb_zones_h-1)
                zones[i][j].setPosition(TypePosition.BORD_INF);
            else zones[i][j].setPosition(TypePosition.NORMAL);
        }
    }
}
```

Chaque zone permet de se déplacer suivant différentes zones selon un déplacement torique. Chaque zone va ainsi savoir dans quelle zone elle peut se déplacer grâce à la méthode *setAdjacent()*.

```
public void setAdjacents() throws ZoneNotFoundException {
    for (int i = 0; i < nb_zones_l; i++) {
        for (int j = 0; j < nb_zones_h; j++) {

            int l = nb_zones_l;
            int h = nb_zones_h;

            Zone z = zones[i][j];
            int z_id = z.getId();

            // Ajouter les adjacents
            switch (z.getPosition()) {
                case COIN_SUP_G -> {
                    z.addAjacent(getZoneById(z_id+1));
                    z.addAjacent(getZoneById(z_id+h));
                    z.addAjacent(getZoneById(z_id+h+1));
                    z.addAjacent(getZoneById((z_id+h)-1));
                    z.addAjacent(getZoneById((z_id+2*h)-1));
                    z.addAjacent(getZoneById(z_id+h*(l-1)));
                    z.addAjacent(getZoneById((z_id+h*(l-1))+1));
                    z.addAjacent(getZoneById((z_id+h*l)-1));
                }
                case COIN_SUP_D -> {
                    z.addAjacent(getZoneById(z_id+1));
                    z.addAjacent(getZoneById(z_id-h));
                    z.addAjacent(getZoneById((z_id-h)+1));
                    z.addAjacent(getZoneById((z_id+h)-1));
                    z.addAjacent(getZoneById(z_id-1));
                    z.addAjacent(getZoneById(z_id-h*(l-1)));
```

```
                z.addAjacent(getZoneById((z_id-h*(l-1))+1));
                z.addAjacent(getZoneById((z_id-h*(l-2))-1));
            }
            case COIN_INF_G -> {
                z.addAjacent(getZoneById(z_id-1));
                z.addAjacent(getZoneById(z_id+h));
                z.addAjacent(getZoneById((z_id+h)-1));
                z.addAjacent(getZoneById((z_id-h)+1));
                z.addAjacent(getZoneById(z_id+1));
                z.addAjacent(getZoneById(z_id+h*(l-1)));
                z.addAjacent(getZoneById((z_id+h*(l-1))-1));
                z.addAjacent(getZoneById((z_id+h*(l-2))+1));
            }
            case COIN_INF_D -> {
                z.addAjacent(getZoneById(z_id-1));
                z.addAjacent(getZoneById(z_id-h));
                z.addAjacent(getZoneById((z_id-h)-1));
                z.addAjacent(getZoneById((z_id-h)+1));
                z.addAjacent(getZoneById((z_id-2*h)+1));
                z.addAjacent(getZoneById(z_id-h*(l-1)));
                z.addAjacent(getZoneById((z_id-h*(l-1))-1));
                z.addAjacent(getZoneById((z_id-h*l)+1));
            }
            case BORD_SUP -> {
                z.addAjacent(getZoneById(z_id+1));
                z.addAjacent(getZoneById(z_id+h));
                z.addAjacent(getZoneById(z_id-h));
                z.addAjacent(getZoneById(z_id+h+1));
                z.addAjacent(getZoneById((z_id-h)+1));
                z.addAjacent(getZoneById(z_id-1));
                z.addAjacent(getZoneById(z_id+h-1));
                z.addAjacent(getZoneById((z_id+2*h)-1));
            }
            case BORD_INF -> {
                z.addAjacent(getZoneById(z_id-1));
                z.addAjacent(getZoneById(z_id+h));
                z.addAjacent(getZoneById(z_id-h));
                z.addAjacent(getZoneById((z_id+h)-1));
                z.addAjacent(getZoneById((z_id-h)-1));
                z.addAjacent(getZoneById(z_id+1));
                z.addAjacent(getZoneById((z_id-h)+1));
                z.addAjacent(getZoneById((z_id-2*h)+1));
            }
            case BORD_G -> {
                z.addAjacent(getZoneById(z_id-1));
                z.addAjacent(getZoneById(z_id+1));
                z.addAjacent(getZoneById(z_id+h));
                z.addAjacent(getZoneById(z_id+h+1));
                z.addAjacent(getZoneById((z_id+h)-1));
                z.addAjacent(getZoneById(z_id+h*(l-1)));
```

```
                z.addAjacent(getZoneById((z_id+h*(l-1))+1));
                z.addAjacent(getZoneById((z_id+h*(l-1))-1));
            }
            case BORD_D -> {
                z.addAjacent(getZoneById(z_id-1));
                z.addAjacent(getZoneById(z_id+1));
                z.addAjacent(getZoneById(z_id-h));
                z.addAjacent(getZoneById((z_id-h)+1));
                z.addAjacent(getZoneById((z_id-h)-1));
                z.addAjacent(getZoneById(z_id-h*(l-1)));
                z.addAjacent(getZoneById((z_id-h*(l-1))-1));
                z.addAjacent(getZoneById((z_id-h*(l-1))+1));
            }
            case NORMAL -> {
                z.addAjacent(getZoneById(z_id-1));
                z.addAjacent(getZoneById(z_id+1));
                z.addAjacent(getZoneById(z_id-h));
                z.addAjacent(getZoneById(z_id+h));
                z.addAjacent(getZoneById((z_id-h)-1));
                z.addAjacent(getZoneById((z_id+h)-1));
                z.addAjacent(getZoneById(z_id+h-1));
                z.addAjacent(getZoneById(z_id+h+1));
            }
            }

            // Enlever les doublons
            Set<Zone> adjacentsSansDoublons = new HashSet<>(z.getAdjacents());
            z.getAdjacents().clear();
            z.getAdjacents().addAll(adjacentsSansDoublons);

        }
    }
}
```

Les zones voisines seront ainsi stockées dans un tableau appelé *adjacent*

2.3 Classe Vivant

La classe Vivant regroupe des attributs communs à tous les êtres vivants de notre simulation telle que le nom et son niveau d'eau, compris entre 0 et 100. Il implémente aussi une *TypeTaille* que nous détaillerons plus loin. La classe implémente enfin des méthodes telles que la mort d'un animal.

La classe Animal, permet d'introduire un champ de nourriture, spécifique à tous les animaux, et d'implémenter des méthodes pour se nourrir, pour boire et pour modéliser la

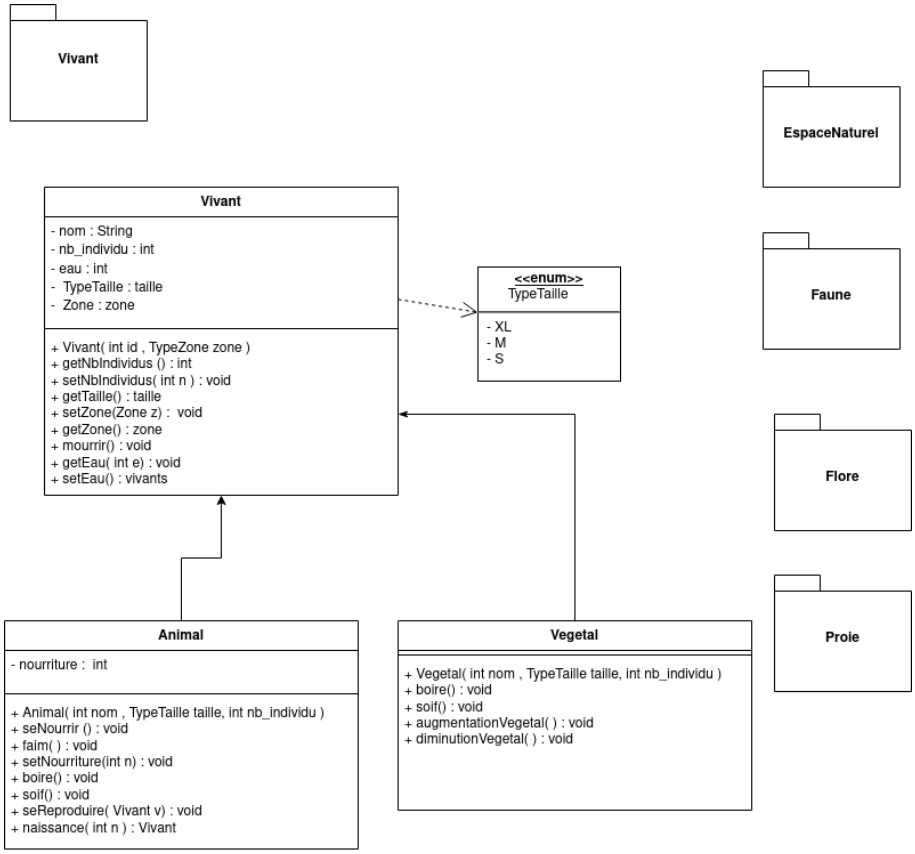

FIGURE 3 – Diagramme de classe - Package Vivant

soif et la fin. Enfin, une méthode *seReproduire*, qui teste la faisabilité d'une reproduction et une méthode *naissance* pour faire apparaitre de nouveaux individus.

2.3.1 TypeTaille

Le TypeTaille permet de rendre compte des différences de taille entre les animaux, notamment dans l'action de se nourrir. En effet, la jauge de nourriture sera remplie différemment en fonction de la taille de l'animal. C'est l'objet de la méthode *seNourrir* qui permet de gérer les différents cas, sachant que tous les animaux sont catégorisés selon 3 taille : XL (Pour les animaux à grand gabarit comme un loup), M (Pour un gabarit plus petit, comme un lapin) et S (Pour des animaux de petites taille de type insectes).

```
// Différents cas en fonction de la taille
switch (getTaille()) {
case S -> {
switch (v.getTaille()) {
   case S -> {
      setNourriture(nourriture + 100);
```

```
                }
                case M -> {
                    break;
                }
                case XL -> {
                    break;
                }
            }
        }
        case M -> {
            switch (v.getTaille()) {
                case S -> {
                    setNourriture(nourriture + 50);
                }
                case M -> {
                    setNourriture(nourriture + 100);
                }
                case XL -> {
                    break;
                }
            }
        }
        case XL -> {
            switch (v.getTaille()) {
                case S -> {
                    setNourriture(nourriture + 25);
                }
                case M -> {
                    setNourriture(nourriture + 50);
                }
                case XL -> {
                    setNourriture(nourriture + 100);
                }
            }
        }
    }

    // Tuer le vivant mangé
    v.mourrir();
```

2.4 Package Espace Naturel

Ce package comprend 3 interfaces représentant les 3 type d'une zone. Lorsqu'elles sont implémentées, elles permettent de modéliser certaines contraintes que peuvent subir les vivants dépendamment d'où ils vivent. Si certains animaux se retrouvent dans une zone

hors de leurs zones dites naturelle, ils subiront des contraintes d'environnement plus élevé (Comme une soif accélère).

2.5 Package Proie

Pour modéliser le système de prédation de manière cohérente, il nous a fallu implémenter plusieurs interfaces. Une interface proie et plusieurs interfaces *estProie* qui sont implémentés pour chaque animal. Cette interface, lorsqu'elle est implémentée par un animal précis, permet d'étiqueter cet animal comme étant une proie de l'animal spécifié. Cet étiquetage servira par la suite à déterminer, pour chaque prédateur voulant manger un animal, quel animal il pourra manger parmi la liste des animaux dans la liste des *Vivants*. Le même procédé est utilisé pour les herbivores. Cependant, pour simplifier dans un premier temps, on considère que les herbivores peuvent manger n'importe quelles plantes.

2.6 Faune

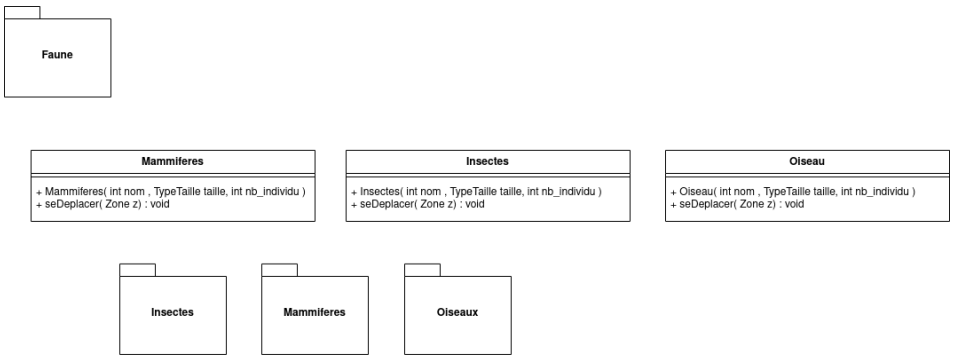

FIGURE 4 – Diagramme de classe - Package Faune

Le package Faune contient les 3 Types d'animaux présents dans la simulation, à savoir des Mammifères, des insectes et des oiseaux. Le but de cette distinction est de séparer les différents cas de déplacement inhérent à chaque animal. Les Mammifères, dans leur ensemble, peuvent uniquement se déplacer en marchant. Les animaux auront ainsi une probabilité de se déplacer une seule fois par tour dans les cases autorisé. En revanche, les oiseaux, peuvent se déplacer 2 fois en un tour, la fonctionnalité n'a cependant pas encore été implémenté de manière fonctionnelle. Pour simplifier, il a été décidé que les

insectes peuvent se déplacer comme les Mammifères ou les Oiseaux, dépendamment du type d'insectes.

2.7 Flore

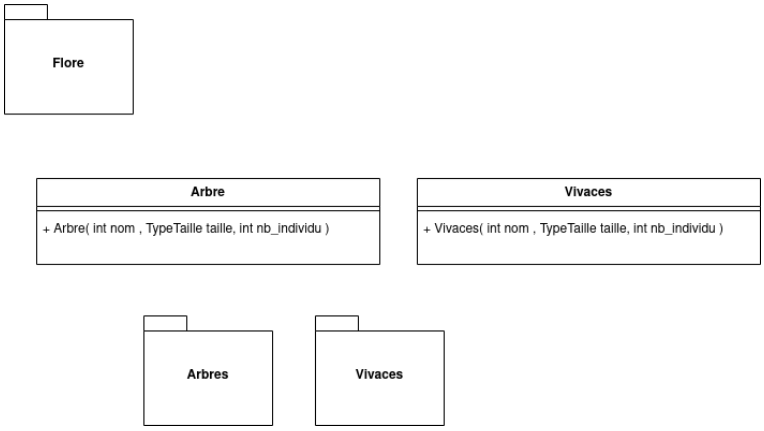

FIGURE 5 – Diagramme de classe - Package Flore

Le package Flore contient des arbres, une des variable clé pour changer le type de zone, et des vivaces, qui sont la principale source de nourriture pour les animaux de type herbivores.

2.8 Loi de la Nature et initalisation de l'environnement

La classe "Loi de la Nature" permet de gérer l'appel de toutes les fonctions lié à l'activité des animaux. Les fonctions d'initialisation de l'environnement telle que *initVivants*, pour l'initialisation et l'attribution d'un *Vivant* dans une zone, va appeler des méthodes de probabilité dans la classe *LoiNature* pour attribuer ou non un *Vivant*. C'est en modifiant ces paramètres que l'on régler au mieux la simulation.

3 Conclusion générale

En conclusion, notre projet de simulation d'un écosystème en utilisant Java a été une expérience enrichissante et passionnante. Nous avons réussi à créer un environnement virtuel réaliste où les différentes espèces interagissent de manière dynamique et évolutive.

Au cours de ce projet, nous avons pu mettre en pratique nos connaissances en programmation orientée objet, en utilisant des concepts tels que l'héritage, l'encapsulation et le polymorphisme pour modéliser les entités de l'écosystème. Nous avons également utilisé des structures de données appropriées pour représenter les populations d'espèces et leurs interactions.

L'utilisation de Java nous a permis de bénéficier de sa robustesse, de sa portabilité et de sa flexibilité, ce qui a grandement facilité le développement du projet. Nous avons également utilisé des bibliothèques et des frameworks supplémentaires pour créer une interface utilisateur conviviale et intuitive, permettant aux utilisateurs de visualiser et d'interagir avec l'écosystème simulé.

Grâce à notre simulation, nous avons pu observer les dynamiques complexes qui se produisent au sein d'un écosystème. Les différentes espèces ont interagi les unes avec les autres, formant des réseaux trophiques et des relations de prédation. Nous avons également pu étudier l'impact des changements environnementaux sur la population des espèces, ce qui nous a permis de mieux comprendre l'équilibre délicat qui existe dans un écosystème naturel.

Ce projet nous a également permis de développer nos compétences en matière de résolution de problèmes, de collaboration et de gestion de projet. Nous avons travaillé en équipe pour concevoir, implémenter et tester la simulation, en nous adaptant aux défis rencontrés et en itérant sur nos solutions.

Cependant, il a été regrettable à notre égard d'implémenter les méthodes sans forcément les tester auparavant

En conclusion, ce projet nous a offert une occasion unique d'explorer et de comprendre les complexités d'un écosystème, tout en affinant nos compétences en programmation.