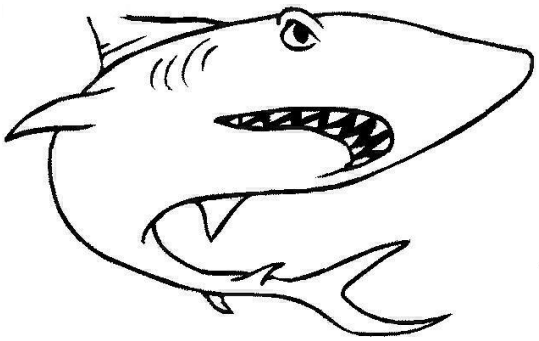
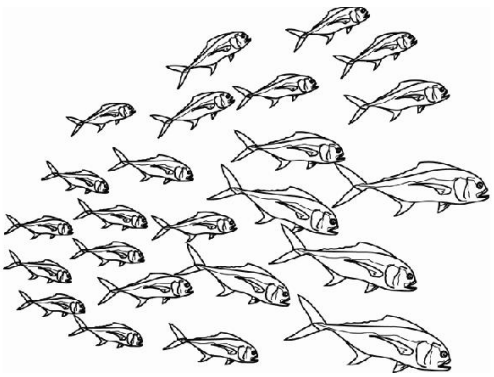
Rapport Final de Projet

Système Proie-Prédateur





Réalisé par

Guillaume Matuszek et Damien Sendner

Sous la direction de

Marc Johannides

Année universitaire 2009-2010

Remerciements

Nous remercions M.Johannides pour son écoute attentive et son aide tout au long de ce projet, ainsi que pour ces conseils avisés.

Un grand merci à M.Cogis pour ses cours de programmation qui nous ont permis de mener à bien ce projet.

Nous tenons aussi à remercier nos amis pour leurs regards critiques qui nous ont permis d'améliorer notre projet.

Table des matières

Glossaire

Système Proies-Prédateurs : C'est un écosystème où coexiste deux espèces dont l'une dévore l'autre.

JFreeChart : C'est une interface de programmation Java permettant de créer des graphiques et des diagrammes.

ArrayList : En Java, c'est un mélange entre les listes et les tableaux, sa taille est modifiable mais on peut aussi accéder directement à une valeur grâce à un indice.

Implémentation : Réalisation, mise en oeuvre.

Matrice : Tableau à deux dimensions, comme une grille.

Table des figures

Introduction

Lotka-Volterra a créé un modèle proie-prédateur. Ce modèle montre l'évolution des proies et des prédateurs dans un écosystème. Cette évolution est théorique ainsi on peut se demander si ce modèle peut s'appliquer en pratique. Pour répondre à cette problématique, nous avons élaboré une application java qui permettra d'observer l'évolution des proies et des prédateurs dans un environnement. Ainsi notre but est que nous puissions observer à la fin du projet une évolution similaire à celle de Lotka-Volterra en réglant d'une certaine façon les paramètres de notre application.

Nous allons tout d'abord établir notre objectif, notamment en définissant ce que sont les équations de Lotka-Volterra, puis nous décrirons le cahier des charges. Nous verrons ensuite, l'analyse du moteur et de l'interface graphique, nous poursuivrons par l'aspect plus technique de la programmation de ce projet. Et, finalement nous discuterons sur les améliorations que l'on pourrait lui apporter et sur les possibles applications d'un tel logiciel.

Chapitre 1

Définition et objectifs

1.1 Les équations de Lotka-Volterra

D'après l'encyclopédie libre Wikipedia [2] :

> En mathématiques, les équations de Lotka-Volterra, que l'on désigne aussi sous le terme de "modèle proie-prédateur", sont un couple d'équations différentielles non-linéaires du premier ordre, et sont couramment utilisées pour décrire la dynamique de systèmes biologiques dans lesquels un prédateur et sa proie interagissent. Elles ont été proposées indépendamment par Alfred James Lotka en 1925 et Vito Volterra en 1926.
>
> Elles s'écrivent fréquemment :
> $\frac{\mathrm{d}x(t)}{\mathrm{d}t} = x(t)(\alpha - \beta y(t))$
> $\frac{\mathrm{d}y(t)}{\mathrm{d}t} = -y(t)(\gamma - \delta x(t))$
> où
> - $x(t)$ est l'effectif des proies ;
> - $y(t)$ est l'effectif des prédateurs ;
> - t est le temps ;
> - $\frac{\mathrm{d}x(t)}{\mathrm{d}t}$ et $\frac{\mathrm{d}y(t)}{\mathrm{d}t}$ représentent les taux de croissance des populations au cours du temps ;
>
> Les paramètres suivants caractérisent les interactions entre les deux espèces
> - α, taux de reproduction des proies en l'absence de prédateurs
> - β, taux de mortalité des proies due aux prédateurs
> - γ, taux de reproduction des prédateurs en fonction des proies mangées
> - δ, taux de mortalité des prédateurs en l'absence de proies.
>
> L'équation de la proie devient :
> $\frac{\mathrm{d}x}{\mathrm{d}t} = \alpha x(t) - \beta x(t)y(t)$
>
> Les proies sont supposées avoir une source illimitée de nourriture et se reproduire exponentiellement si elles ne sont soumises à aucune prédation ; cette croissance exponentielle est représentée dans l'équation ci-dessus par le terme $\alpha x(t)$. Le taux de prédation sur les proies est supposé proportionnel à la fréquence de rencontre entre les prédateurs et les proies ; il est représenté ci-dessus par $\beta x(t)y(t)$.

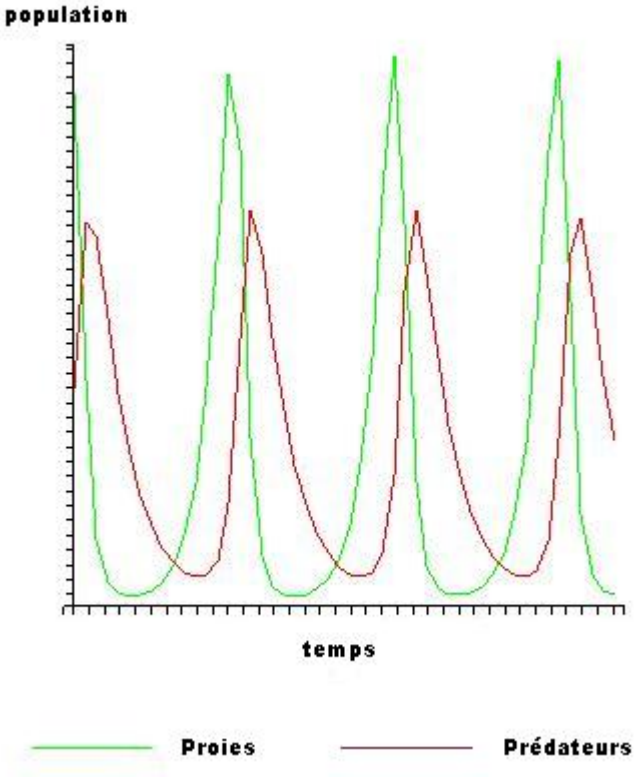


FIG. 1.1 – Évolution des populations au cours du temps

Si l'un des termes $x(t)$ ou $y(t)$ est nul, alors il ne peut y avoir aucune prédation.

Avec ces deux termes, l'équation peut alors être interprétée comme : la variation du nombre de proies est donnée par sa propre croissance moins le taux de prédation qui leur est appliqué.

L'équation du prédateur devient :
$\frac{\mathrm{d}y(t)}{\mathrm{d}t} = \delta x(t)y(t) - \gamma y(t)$

Dans cette équation, $\gamma x(t)y(t)$ représente la croissance de la population prédatrice. De plus, $\delta y(t)$ représente la mort naturelle des prédateurs ; c'est une décroissance exponentielle. L'équation représente donc la variation de la population de prédateurs en tant que croissance de cette population, diminuée du nombre de morts naturelles.

Les équations admettent des solutions périodiques qui n'ont pas d'expressions simple à l'aide des fonctions trigonométriques habituelles. Néanmoins, une solution approximative linéarisée offre un mouvement harmonique simple, avec la population des prédateurs en retard de 90 ° (un quart de période) sur celle des proies.

Dans le modèle utilisé (voir Fig. 1.1), les prédateurs prospèrent lorsque les proies sont nombreuses, mais finissent par épuiser leurs ressources et déclinent. Lorsque la population de prédateur a suffisamment diminuée, les proies profitant du répit se reproduisent et

leur population augmente de nouveau. Cette dynamique se poursuit en un cycle de croissance et déclin.

Pour résumé quand le nombre de proies est grand, le nombre de prédateurs augmente. Cependant à un certain seuil le nombre de proies diminue car il y a trop de prédateurs. Ceci entraîne la décroissance du nombre de prédateurs à cause de l'insuffisance du nombre de proies. Donc les proies se multiplient plus car il y a moins de prédateurs, etc... Ce cycle se reproduit donc à l'infinie.

1.2 Objectifs

Notre objectif est de mesurer l'évolution de deux populations en interactions dans un système proie-prédateur.

Les paramètres de cette observation pourront être modifiés par l'utilisateur. Les paramètres modifiables seront précisés dans le cahier des charges. Cela permettra d'observer les changements du système en fonction des paramètres.

L'objectif final est d'obtenir un graphique représentant l'évolution de la population de proies et de prédateurs en fonction du temps, dans le but de comparer ce graphique à la représentation théorique de la formule de Lotka-Volterra.

Chapitre 2

Cahier des charges

2.1 Simulation du système proie-prédateur

2.1.1 Environnement

L'environnement sera représenté par une matrice dans laquelle les organismes évolueront.

Cette matrice est gérée en tore, ce qui signifie que lorsque un organisme atteint les bornes de la matrice il réapparaît de l'autre côté. De cette façon nous pouvons simuler un monde sphérique.

Il pourra y avoir plusieurs organismes sur une même case.

Le temps est géré par un compteur dont la valeur augmente à chaque période. Pendant celle-ci tous les êtres vivants évoluent.

2.1.2 Prédateurs

Déplacement : Les prédateurs se déplacent selon une vitesse qui peut être changée par l'utilisateur. Pour se déplacer un prédateur choisit aléatoirement une direction parmi les huit cases qui lui sont adjacentes.

Prédation : Les prédateurs ont une zone d'influence que l'utilisateur peut modifier. C'est la zone dans laquelle ils sont capables de manger les proies.

Le prédateur mange à chaque période une seule proie dans sa zone d'influence.

Quand il mange une proie cela augmente ses chances de se reproduire.

Reproduction : Le moment auquel se reproduit le prédateur est défini grâce à ce que l'on nommera une "horloge". Cette horloge est calculée en fonction d'un paramètre lambda qui est modifié à chaque fois que le prédateur mange. Cette horloge est calculée à la naissance du prédateur et à chaque fois qu'il mange.

Mort : Les prédateurs reçoivent une date de mort dès leur naissance, celle-ci est calculée grâce à une horloge identique à l'horloge de reproduction qui prend un paramètre lambda que l'utilisateur peut modifier.

2.1.3 Proies

Déplacement : Les proies se déplacent selon une vitesse qui peut être changée par l'utilisateur. Pour se déplacer une proie choisit aléatoirement une direction parmi les huit cases qui lui sont adjacentes.

Nourriture : On considère qu'une proie mange automatiquement. Donc nous ne prendrons pas en compte la présence de nourriture.

Reproduction : Les proies ont une zone de ponte que l'utilisateur peut modifier. On choisit une case aléatoirement dans cette zone pour faire naître la progéniture de la proie. Le moment auquel doit se reproduire la proie est décidé grâce à une horloge. Celle-ci possède un paramètre lambda que l'utilisateur peut modifier pour augmenter ou diminuer le taux de reproduction des proies.

Mort : Une proie meurt uniquement par prédation.

2.2 Utilisation et visualisation

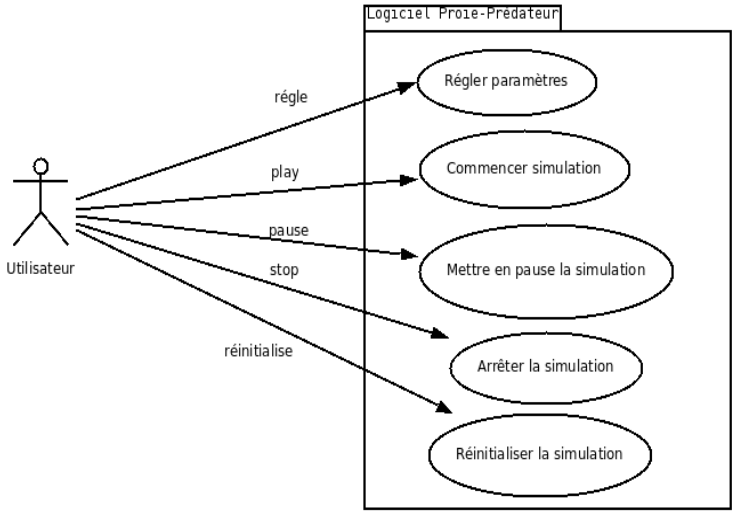


FIG. 2.1 – Le diagramme de cas d'utilisation

2.2.1 Déroulement

Ce déroulement est traduit à travers le diagramme de cas d'utilisation (voir Fig.2.1). L'utilisateur saisit les paramètres qu'il veut puis appui sur "Valider".

Il peut ensuite appuyer sur le bouton "lecture" pour lancer la simulation.

Pour changer les paramètres en cours de simulation il doit obligatoirement appuyer sur "pause". Pour effacer cette simulation et en recommencer une nouvelle il doit appuyer sur le bouton "Réinitialiser".

2.2.2 Paramètres

L'utilisateur peut modifier les paramètres suivants :
- la taille du monde
- le nombre initial de proies et le nombre initial de prédateurs
- la vitesse de déplacement des proies et des prédateurs
- la grandeur de la zone de ponte des proies
- la grandeur de la zone d'influence des prédateurs
- le taux de reproduction des proies
- l'augmentation du taux de reproduction des prédateurs pour chaque proie mangée

2.2.3 Graphe

On utilisera un graphique fait grâce à la librairie JFreeChart pour observer l'évolution des deux populations en temps réel. Ce graphe représente le nombre d'individus dans chaque population en fonction du temps.

Chapitre 3

Outils utilisés

3.1 LaTeX

Source : Konrad Florczak, Formation LaTeX[4]

Définition

Ce rapport est rédigé en LaTeX. C'est un ensemble de macros qui permet à un auteur de mettre en page son travail, avec la meilleure qualité typographique en utilisant un format professionnel pré-défini. La philosophie LaTeXconsiste à créer une mise en page en forçant l'auteur à décrire la structure logique de son document et en choisissant lui-même la mise en page la plus appropriée. Nous avons donc choisi d'utiliser ce logiciel de composition de documents afin d'avoir un rapport de la meilleure qualité possible. Pour la soutenance, nous utiliserons Beamer, une classe LaTeXadaptée à la création de présentations [3].

Avantages et inconvénients

Avantages
- Mise en page professionnelle
- Pour mettre en page un document, il suffit de connaître quelques commandes de base pour décrire la structure logique du document
- Des structures complexes telles que des notes de bas de page, des renvois, la table des matières ou les références bibliographiques sont produits facilement
- LaTeXencourage les auteurs à écrire des documents bien structurés

Inconvénients
- La mise au point d'une présentation entièrement nouvelle est difficile et demande beaucoup de temps
- Écrire des documents mal organisés et mal structurés est très difficile

3.2 UML

Nous avons utilisé le formalisme UML pour concevoir notre projet. C'est le logiciel libre Dia [1] qui nous a permi de dessiner les diagrammes.

3.3 Eclipse

Eclipse est un environnement de travail permettant de développer des applications Java. Nous utilisons ce logiciel dans le cadre des cours de programmation, nous l'avons donc choisi pour sa simplicité d'utilisation et son ergonomie.

3.4 Google Documents

Google Documents est un outil permettant de partager des documents en ligne. Nous avons utilisé cet outil afin que chacun puisse consulter les avancés du projet en temps réel à tout moment.

Chapitre 4

Analyse du moteur

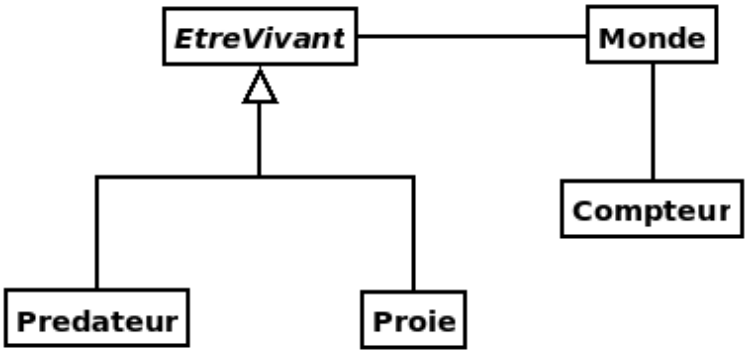


FIG. 4.1 – Les classes

4.1 Aspect statique

4.1.1 Modélisation

Le principal objectif étant de se représenter le rôle de chacune des classes (voir Fig. 4.1), notamment celui du monde par rapport aux êtres vivants. Nous avions deux choix principaux pour cette représentation, deux visions des choses totalement différentes.

La première donnant à la classe Monde peu de "pouvoir" car elle serait utilisée uniquement dans un but de représentation des êtres vivants et ne s'occuperait pas des interactions entre ceux-ci. Ils se chargeraient eux-mêmes de leur reproduction, déplacement, etc... Envoyant à chaque fois les nouvelles données au Monde pour qu'il les enregistre.

La seconde consistant à donner au contraire le rôle le plus important à la classe Monde. Celle-ci serait comme un contrôleur tout puissant qui peut tout gérer que ce soit les interactions entre les êtres vivants, le temps, les

déplacements, etc... Les êtres vivants s'occuperaient uniquement de ce qui leur est propre comme leur date de mort ou de reproduction mais rien d'autre.

Nous avons choisi cette deuxième solution car elle est beaucoup plus simple que la première à mettre en oeuvre et elle permet d'éclaircir le code. Il y a donc une classe Monde qui contrôle tout, et de celle-ci dépend le temps (la classe Compteur) et les êtres vivants.

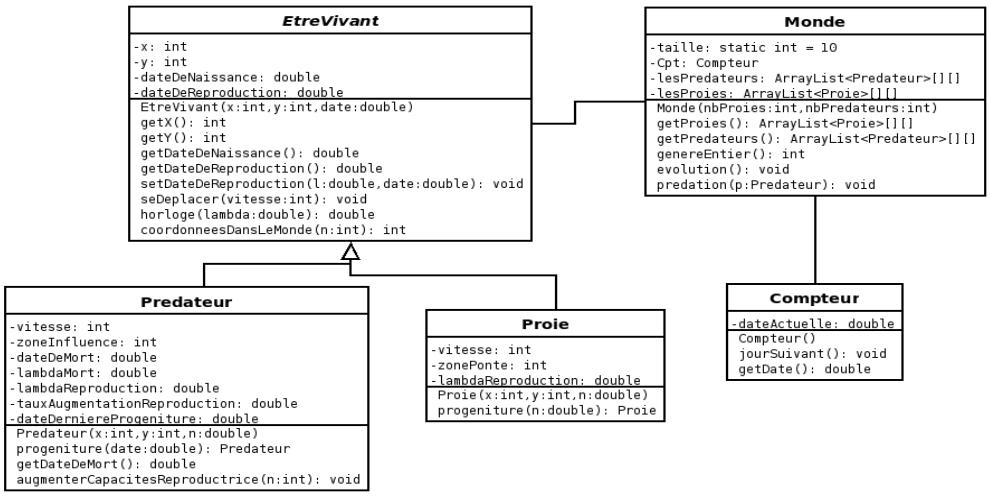


FIG. 4.2 – Le diagramme de classe du moteur

4.1.2 Implémentation

Il y a donc cinq classes en tout dans le moteur. Nous distinguons deux ensembles, le premier étant celui qui comporte une classe abstraite EtreVivant, puis une classe Prédateur et une classe Proie qui héritent toutes les deux de la classe EtreVivant. La seconde est composée du monde et de la classe compteur.

Chaque être vivant dispose d'une coordonnée en x et en y (voir Fig. 4.2), d'une date de naissance et d'une date de reproduction. Un être-vivant se charge de calculer sa prochaine date de reproduction en utilisant la méthode *setDateDeReproduction()* et la méthode *horloge()* (voir chapitre 4.2.2). Il calcule ses nouvelles coordonnées avec *seDeplacer()* et se contente de donner au monde ces données grâce à des accesseurs.

Les prédateurs mangent les proies dans une certaine zone : c'est la zone d'influence. Chaque prédateur meurt à une date donnée, elle est calculée à leur naissance grâce à l'horloge de mort. Le prédateur possède donc toutes les méthodes et attributs des êtres vivants, mais il gère en plus sa date de mort. Il peut aussi 'créer' sa progéniture et gérer l'augmentation de sa reproduction à travers le lambda reproduction (voir chapitre 4.2.2).

Les proies peuvent pondre leurs œufs à une certaine distance : c'est la zone de ponte. Elles peuvent uniquement créer leur progéniture en fonction de leur zone de ponte.

L'ensemble de ces êtres vivants appartient à un Monde : une classe contenant une matrice d'ArrayList de Proies et une matrice d'ArrayList de Prédateurs

(voir chapitre 4.1.3). Cette classe fait évoluer ces deux matrices d'ArrayList dans la méthode *évolution()* (voir chapitre 4.2.3).

La classe Compteur permet d'initialiser une date et de passer au jour suivant (voir chapitre 4.2.1).

4.1.3 Représentation

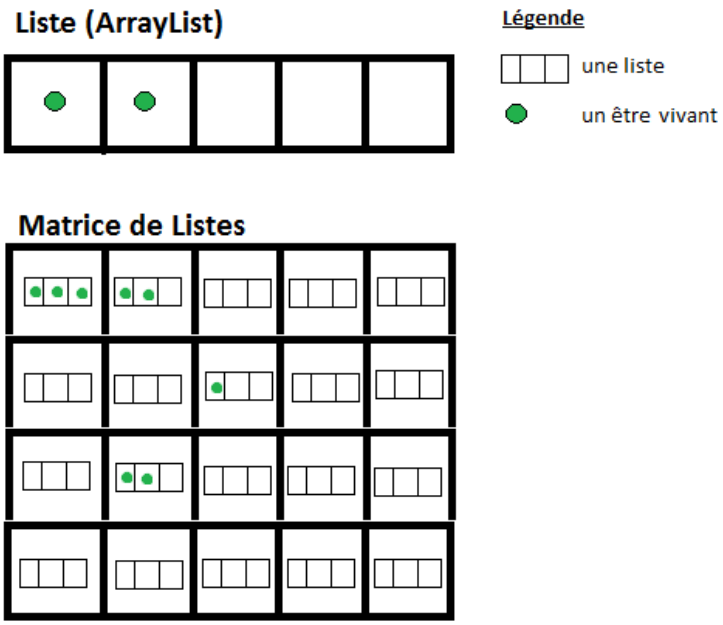


FIG. 4.3 – Comparaison entre liste et matrice de liste

Un des problèmes fondamentaux que nous avons rencontré est la représentation de l'ensemble des êtres vivants c'est-à-dire proies et prédateurs. En effet étant donné qu'il pouvait y avoir plusieurs êtres vivants sur la même case, la représentation c'est révélée complexe.

Au début nous pensions qu'il n'y avait qu'une seule solution possible, la représentation sous forme de listes, en java les ArrayList (voir Glossaire). Cette représentation n'étant pas spatiale tous les êtres vivants se suivent peu importe leurs coordonnées.

Cependant après réflexion une autre idée nous est venue à l'esprit. En effet, pour représenter les êtres vivants sur un plan, il nous faudrait une matrice mais dans celle-ci nous ne pourrions pas avoir plusieurs organismes sur la même cellule. La solution étant d'utiliser une matrice qui contient dans chacune de ses cellules une liste, ce serait donc une matrice d'arraylist. Nous pourrions donc ajouter ou supprimer des êtres vivants à notre guise dans chaque cellule de cette matrice.

Le gros problème a été de déterminer laquelle de ces solutions est la meilleure, c'est-à-dire celle qui est la plus simple à mettre en oeuvre mais qui permet aussi d'optimiser le code pour gagner du temps. En effet ce logiciel pouvant comporter un très grand nombre d'êtres vivants, il nous a fallu choisir la solution qui fait

le moins de calculs possibles. C'est très difficile à déterminer mais nous pouvons tenter une comparaison en regardant le nombre de parcours totaux nécessaires avec chacune de ces solutions.

Il nous est apparu que la deuxième solution évitait un certain nombre de parcours totaux, notamment lorsqu'il faut rechercher toutes les proies qui sont dans la zone d'influence du prédateur. Avec une liste nous aurions dû parcourir toute la liste et vérifier pour chaque proie, si ces coordonnées sont égales aux coordonnées de chaque case de la zone d'influence du prédateur. Au contraire dans les matrices il faut seulement supprimer directement toutes les proies qui sont dans la zone d'influence.

En revanche pour le déplacement une liste ne pose aucun problème mais dans une matrice nous devons éviter de déplacer deux fois le même être vivant. En effet si on déplace une proie sur la case suivante, nous allons repasser dessus puis la refaire se déplacer une autre fois au cours d'un seul tour il faut donc trouver une solution.

Au niveau de l'affichage, les listes sont plus économiques quand il n'y a pas un grand nombre d'êtres vivants mais en revanche dans le cas contraire ce sont les matrices qui prennent moins de temps. En effet pour le deuxième cas, même s'il y a plusieurs êtres vivants sur une même case on ne passe qu'une fois. Avec les listes s'il y avait deux mille proies sur une même case cela fait deux mille cellules sur lesquelles passer.

On peut donc dire que chaque méthode à ses inconvénients et ses avantages, mais nous avons choisi la deuxième c'est-à-dire la matrice d'arraylist car elle nous a semblé plus intéressante même si ce choix peut être contestable.

4.1.4 Gestion sphérique du monde

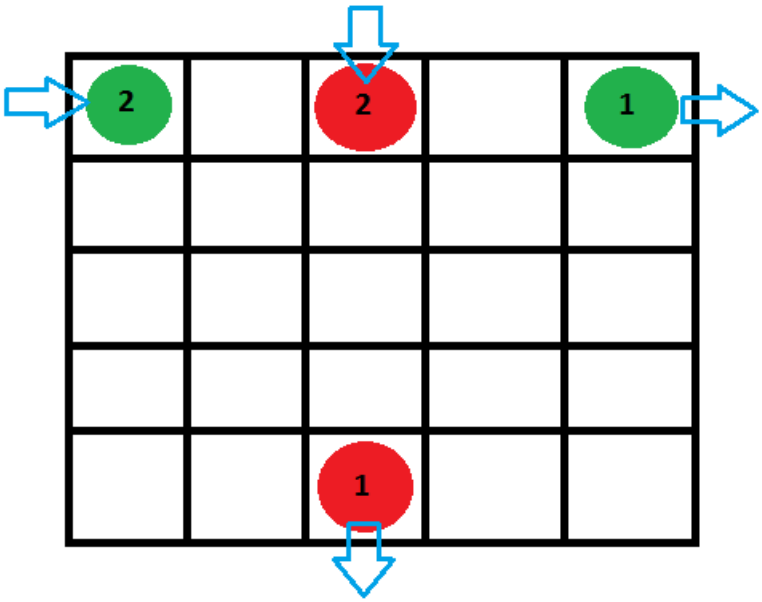


FIG. 4.4 – Gestion des déplacements en tore

Dans le cahier des charges nous avons vu que nous avions décidé de gérer le monde tel un environnement sphérique. Donc lorsqu'un être vivant se déplace en dehors des limites du monde, il réapparaît de l'autre côté (voir Fig.4.4). Nous avons donc implémenté une méthode permettant de vérifier si un être vivant se situait dans le monde ou si il ne s'y trouvait pas. Cette méthode n'est utilisé que dans deux cas.
Le premier cas est lorsque l'on déplace un être vivant, il faut s'assurer qu'après s'être déplacé il se trouve toujours dans le monde.
Le deuxième cas est lorsque les prédateurs mangent. Si un prédateur se situe au bord du monde, il doit pouvoir manger de l'autre côté du monde. Il nous faut donc utiliser la méthode *coordonneesDansLeMonde()* qui se charge à chaque fois de réajuster les coordonnées.

4.2 Aspect dynamique

4.2.1 Gestion du temps

Dans un premier temps, nous avions décidé que les êtres vivants avaient un âge et qu'ils devaient mourir ou se reproduire à un certain âge. Avec cette solution nous comparions l'âge de naissance de l'être vivant avec l'âge de sa reproduction ou de sa mort en fonction du temps passé entre sa naissance et cette observation. Notre tuteur nous a par la suite proposé de gérer ce système à l'aide d'une date. Avec cette solution nous comparons la date de reproduction de l'être-vivant avec la date actuelle. Cette solution permet de clarifier le code et permet aussi de mieux gérer les horloges (voir paragraphe 4.2.2). Nous avons donc employé un compteur qui génère une date. Dans la classe Compteur nous avons créé une méthode *jourSuivant()* qui incrémente la date actuelle, nous utilisons cette méthode dans la méthode *evolution()* de la classe Monde. Ainsi chaque évolution du système correspond à un jour passé et les modifications imposées par cette évolution sont effectuées sur les matrices d'ArrayList.

4.2.2 Les horloges

4.2.2.1 Théorie

Pour gérer les événements pseudo-aléatoire tels que la reproduction ou la mort nous allons utiliser ce que l'on nommera des "horloges".

On utilise la loi exponentielle de paramêtre λ :

$$\frac{-1}{\lambda} * \log(1 - \mu)$$

Le nombre μ est un nombre aléatoire entre 0 et 1.

Cette formule nous donne donc un nombre qui dépend du nombre aléatoire μ et de λ. C'est avec ce λ que l'on peut influencer l'horloge. Ainsi si on utilise un grand λ le résultat sera plus petit donc la période avant le prochain événement plus courte. En revanche si λ est petit alors le résultat sera grand donc la période avant le prochain événement plus longue. C'est donc ce λ que l'utilisateur va pouvoir modifier pour régler le taux de mort ou de reproduction des êtres-vivants.

4.2.2.2 Utilisation

A chaque naissance d'un nouveau prédateur nous devons donc calculer grâce à cette formule la période au bout de laquelle il va mourir.

A la naissance d'une proie nous calculons avec cette horloge la période avant sa prochaine reproduction. Puis après chaque reproduction nous recalculons cette période puis nous l'ajoutons à la date actuelle pour obtenir la prochaine date de reproduction de la proie.

Le calcul de l'horloge de reproduction s'effectue lors de la création des proies et des prédateurs. Cependant lorsqu'un prédateur mange une ou plusieurs proies, on recalcule cette horloge. En effet quand celui-ci mange on augmente le lambda de reproduction donc on diminue le temps moyen avant lequel le prédateur devrait se reproduire. On doit donc recalculer la période de reproduction pour prendre en compte ce nouveau paramètre.

4.2.3 Évolution

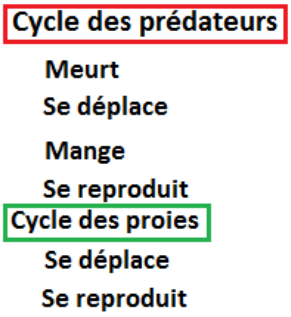


FIG. 4.5 – Déroulement de l'évolution

L'évolution du monde est la méthode la plus grande et la plus complexe à mettre en oeuvre. En effet il a fallu gérer l'imbrication des différents événements tels que la reproduction, le déplacement, etc... de manière à ce qu'il n'y ait pas de conflits entre ceux-ci.

Le premier problème est que nous ne pouvons pas faire un seul parcours de toute la matrice et faire tout dans cette unique parcours. En effet, il va y avoir des problèmes entre les proies et les prédateurs, que faire si une proie pond puis un prédateur mange juste après, etc.. Pour simplifier ce problème nous avons décidé de séparer le cycle des proies de celui des prédateurs.

Les prédateurs commencent donc par leur cycle, s'ils doivent mourir ils meurent sinon ils se déplacent, mangent et se reproduisent.

Ensuite quand tous les prédateurs ont fini, les proies se déplacent et se reproduisent (voir Fig.4.5).

4.2.3.1 Le déplacement

Pour gérer les déplacements nous utilisons une méthode *seDeplacer()* qui est dans la classe EtreVivant, cette méthode se charge uniquement de choisir une direction aléatoire puis de multiplier la direction choisie avec la vitesse pour

obtenir de nouvelles coordonnées qui remplacent les anciennes. C'est ensuite le monde qui se charge de déplacer la proie dans la matrice conformément à la décision prise dans le chapitre modélisation. Ce déplacement s'effectue exactement de la même façon pour les proies ou les prédateurs.

4.2.3.2 La prédation

C'est le monde qui se charge de cette prédation, les prédateurs se contentent uniquement d'incrémenter leur lambda de reproduction en fonction des proies mangées et de fournir leur zone d'influence au monde. Celui-ci regarde pour chaque case de la zone d'influence s'il y a des proies puis en détruit une. A la fin, il avertit le prédateur qu'il a mangé une proie.

4.2.3.3 La reproduction

Dans un premier temps, nous avions décidé de créer une méthode *seReproduire()* afin de créer un nouveau Prédateur ou une nouvelle Proie. Nous avons donc implémenté cette méthode. Elle consistait a vérifier si le Prédateur ou la Proie devait se reproduire, si c'était le cas on retournait un nouveau Prédateur ou une nouvelle Proie et si ce n'était pas le cas on retournait la valeur "null". Nous avons eu deux problèmes à cause de cette méthode.
Le premier problème était que nous ne pouvions pas ajouter la valeur "null" à la matrice d'ArrayList de Proies ou de Prédateurs. Nous avons donc décidé de créer une méthode *progeniture()* qui retourne un nouveau Prédateur ou une nouvelle proie. Cette nouvelle méthode serait appelée seulement si l'être vivant devait se reproduire, sinon on ne fait rien.
Le deuxième problème était que si nous déclarions la méthode abstraite dans EtreVivant il fallait donc retourner un être vivant. Or ajouter un être vivant à une matrice d'ArrayList de proies ou de prédateurs est impossible. Nous avons donc supprimé l'abstraction de la méthode et nous avons défini cette méthode *progeniture()* uniquement dans Proie et dans Prédateur. C'est donc le monde qui se charge d'appeler cette méthode si la proie ou le prédateur doit se reproduire.

4.2.3.4 La mort des prédateurs

Au début de l'analyse, les méthodes qui agissaient sur les proies et les prédateurs étaient dans la classe EtreVivant. Ainsi dans la classe Monde nous avions déclaré une matrice d'ArrayList d'êtres vivants. Cependant pour que cela fonctionne, il fallait déclarer les méthodes abstraites dans EtreVivant et les redéfinir dans Proie et dans Predateur. Ainsi nous devions créer une méthode abstraite *getDateDeMort()* dans EtreVivant et la redéfinir dans Proie et dans Predateur. Mais les Proies n'ont pas d'horloge de mort donc nous devions retourner une valeur qui ne serait jamais utilisée. Nous avons donc vu qu'il y avait une erreur d'analyse. Nous avons donc changé les matrices d'ArrayList d'êtres vivants par des matrices d'ArrayList de proies et de prédateurs. Maintenant c'est uniquement le prédateur qui calcule sa date de mort à sa naissance et la transmet au monde pour que celui-ci décide s'il doit mourir ou non.

Chapitre 5

Analyse de l'interface graphique

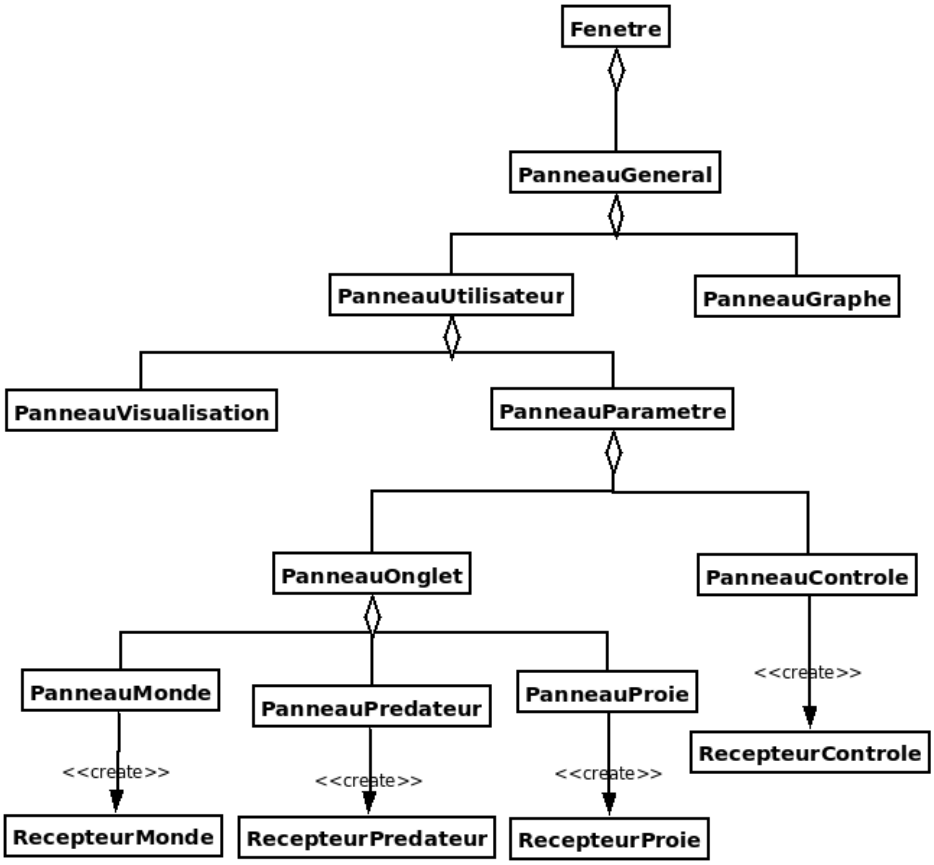


Fig. 5.1 – Le diagramme de classe de l'interface graphique

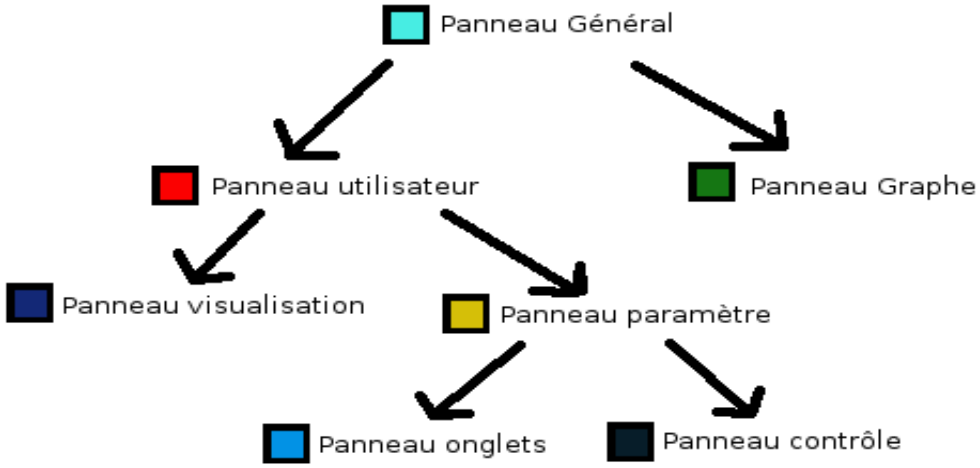


FIG. 5.2 – Architecture des panneaux sous forme d'arbre binaire

5.1 Architecture générale

Pour la conception de l'interface graphique nous avons décidé de diviser le plus possible la modélisation. Cela permet une meilleure maintenance et l'amélioration facile du code. Nous avons donc créé l'interface sous la forme d'un arbre binaire (voir Fig. 5.1 et Fig.5.2). On a créé une classe par panneau (voir Fig. 5.3) puis imbriqué ces panneaux suivant différents niveaux.

Tous les panneaux sont contenus dans la fenêtre mais pour séparer les fonctions qui incombent à la fenêtre des autres nous utilisons un panneau que l'on nommera panneau général qui contient tous les autres panneaux.

Ce panneau contient deux sous panneaux, le panneau utilisateur et le panneau graphe. Le panneau graphe contient le graphique qui permet de visualiser l'évolution des populations.

Le panneau utilisateur contient lui-même deux panneaux, le panneau visualisation et la panneau paramètre. Le panneau visualisation permet de voir l'évolution 'spatiale' des populations, pour observer certaines formations telles que des niches écologiques.

Le panneau paramètre est divisé en deux autres panneaux, le panneau contrôle qui contient tous les boutons de contrôle de l'évolution du monde et le panneau onglet.

Celui-ci contient trois onglets qui sont des panneaux. L'onglet monde qui permet de modifier les paramètres initiaux tels que le nombre de proies, de prédateurs ou la taille du monde. Puis les onglets proie et prédateur qui permettent de modifier les paramètres propres à chaque proie ou prédateur. Ceux-ci peuvent être modifiés au cours de la simulation.

Cette architecture sous forme d'arbre binaire permet une bonne gestion de la disposition des éléments dans la fenêtre. En effet notre première disposition divisait la fenêtre en trois panneaux, mais cela a entraîné de nombreux problèmes avec les gestionnaires de diposition de java.

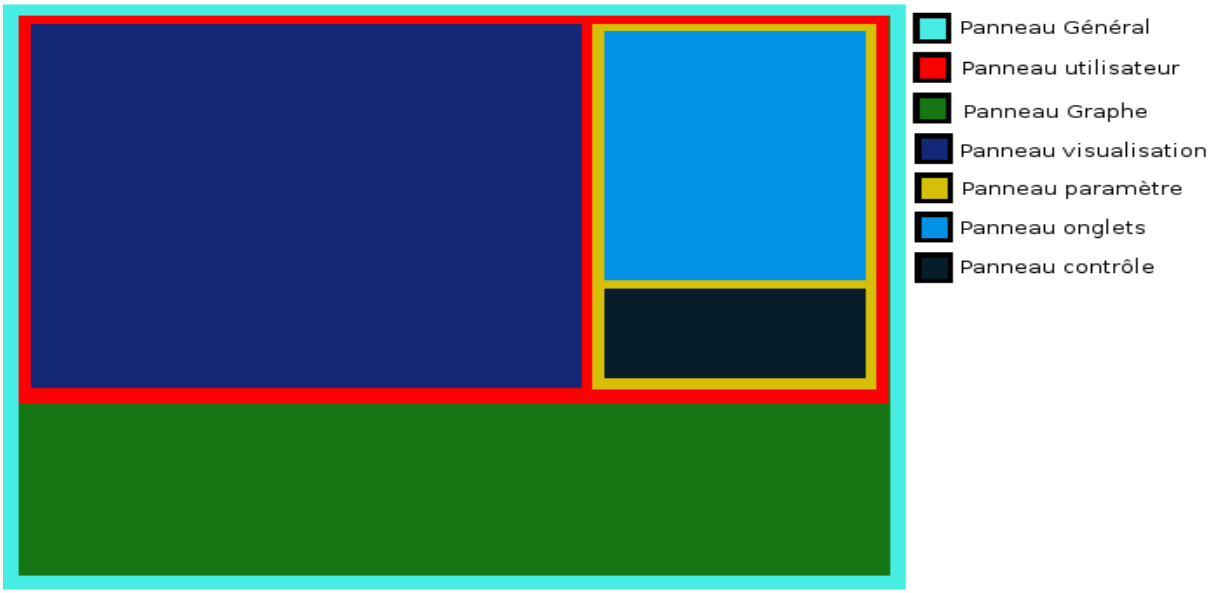


FIG. 5.3 – Disposition des panneaux

5.2 Gestion des événements

La gestion des évènements est un des points les plus complexes que nous avons eu a traiter. Plusieurs solutions s'offraient à nous, utiliser les classes préexistantes comme récepteur d'événement ou créer des classes chargées de ce travail. Nous avons donc décidé de séparer la gestion des événements du reste des classes pour permettre une meilleure clarté du code et donc une meilleure maintenance.

On a ensuite penser à faire un seul récepteur d'évènements qui les concentre tous. Mais étant donné la quantité relativement importante d'événements à gérer et la possibilité d'en rajouter plus tard, nous avons opté pour une autre solution, celle-ci consiste à créer plusieurs récepteurs d'événements, nous en avons créé quatre car nous avons pu identifier quatre groupes d'événements. Les événements générés par le panneau monde, par le panneau proie, par le panneau prédateur et enfin par le panneau de contrôle.

La troisième question fut à quel niveau dans l'architecture des panneaux fallait-il que l'on crée les récepteurs. En effet c'est une situation complexe car ces récepteurs ont besoin de beaucoup d'éléments, en particulier le panneau de contrôle qui a besoin du panneau graphe, du panneau visualisation, du monde ainsi que de toutes les commandes permettant de changer les paramètres. La première solution a été de créer tous les récepteurs à un niveau assez élevé c'est-à-dire dans le panneau utilisateur. Il a donc fallu créé aussi tous les boutons et commandes dans cette classe. Après réflexion, il nous a semblé plus logique et plus "esthétique" d'effectuer la création de ces récepteurs dans la classe pour laquelle ils réceptionnent les événements. Nous avons donc créé toutes les commandes à un niveau plus bas, c'est-à-dire dans le panneau paramètre puis nous les avons "distribuées" dans les classes pour lesquelles elles étaient concernées. En bout de chaîne nous avons donc toutes les données pour créer les récepteurs.

Cette disposition nous permet une certaine simplicité dans le cas d'ajout de boutons. Par exemple, si nous devions ajouter un composant au panneau monde pour régler un nouveau paramètre, il nous suffirait de créer ce composant dans le panneau paramètre puis de le transmettre aux classes panneau onglet et contrôle. Celles-ci l'enverraient au panneau monde et donc on pourrait créer les récepteurs contrôle et monde avec ce composant.

5.3 Aspects Ergonomiques

L'ergonomie a été un des principaux aspects de notre analyse. En effet nous nous sommes orientés vers une version plus ludique que scientifique de ce logiciel. Il a donc fallu produire une interface simple permettant une prise en main rapide de celui-ci.

5.3.1 Onglets

Les onglets permettent de regrouper les paramètres pour une meilleure accessibilité.

5.3.2 Images

On utilise des images pour les boutons telles que les classiques "lecture" représentées par un triangle et "pause" représentées par deux barres, mais aussi une flèche en rotation pour le bouton "réinitialiser" qui représente donc le retour en arrière. Ces boutons permettent une utilisation intuitive (voir Fig. 5.4).

FIG. 5.4 – Les boutons

5.3.3 Utilisation de listes déroulantes

Les listes déroulantes (voir Fig. 5.6) permettent plusieurs choix pré-paramétrés comme "petit" ou "grand" qui ne sont certes pas précis pour une utilisation d'ordre scientifique, mais qui pour une utilisation ludique permettent un choix facile. De la même façon les barres graduées (voir Fig. 5.5) permettent de choisir facilement des valeurs sans se préoccuper de l'échelle, mais bien sûr elles réduisent les possibilités.

FIG. 5.5 – Les barres graduées

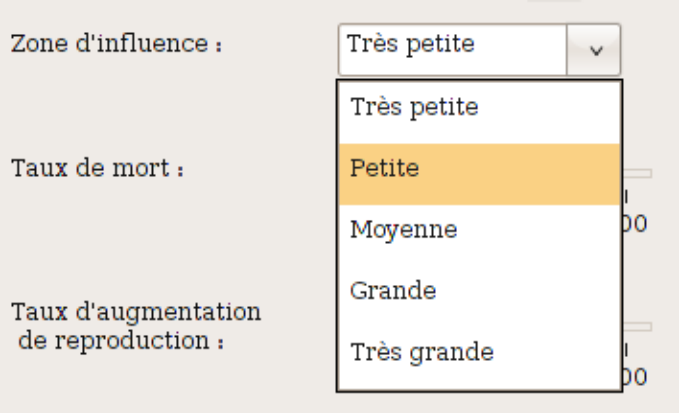


FIG. 5.6 – Les listes déroulantes

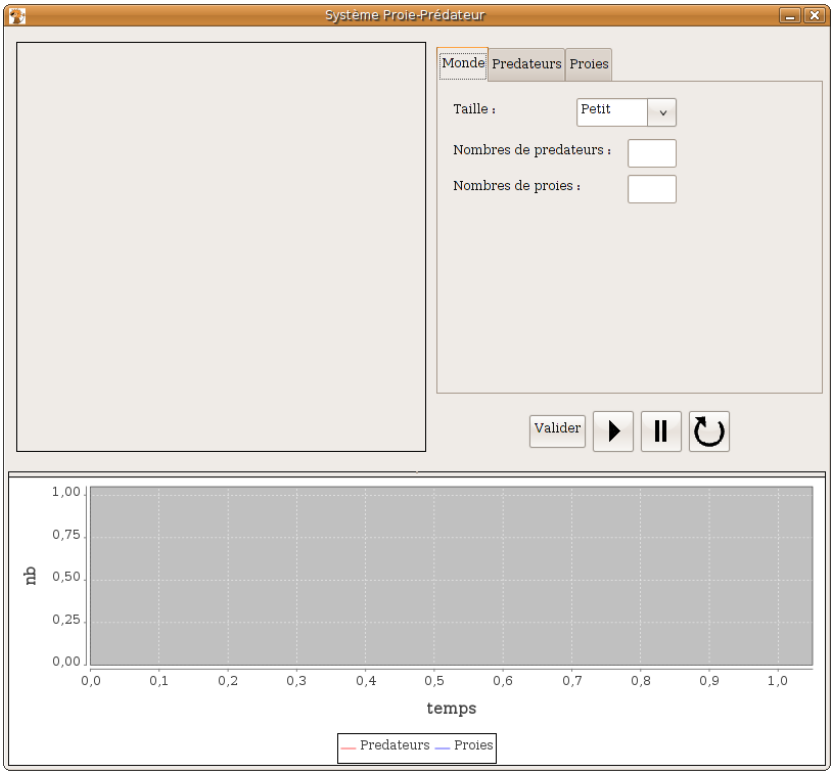


FIG. 5.7 – L'interface graphique

Chapitre 6

La programmation du moteur

6.1 La classe Monde

6.1.1 La méthode *evolution()*

Comme décidé dans l'analyse nous allons séparer le cycle des proies de celui des prédateurs.

Cependant, il y a un problème concernant les déplacements. En effet avec une matrice, comme il est dit dans le chapitre sur la représentation, nous risquons de repasser sur un être vivant. Plusieurs solutions s'offraient à nous, une première consistait à mettre en place une matrice intermédiaire pour les proies et une autre pour les prédateurs. Ces matrices contiendraient les êtres vivants qui se sont déplacés, de cette manière on ne les déplacerait pas dans la matrice originale donc on ne repasserait pas dessus. Après l'implémentation de cette méthode nous nous sommes rendus compte que celle-ci provoquait des lenteurs importantes dans le logiciel. Nous avons donc trouvé une deuxième solution. Dans un premier temps, on utilise la méthode *seDeplacer()* qui change uniquement les coordonnées du prédateur ou de la proie puis on effectue les autres actions comme la prédation ou la reproduction avec ces nouvelles coordonnées. Quand les deux cycles sont finis on en effectue un troisième pour changer la place physique de toutes les proies et de tous les prédateurs. De cette manière, on n'applique pas la méthode *seDeplacer()* dans le troisième cycle, donc même si on déplace un être vivant et que l'on repasse dessus ensuite, on ne le déplacera pas autre part car il ne change pas de coordonnées.

Nous commencons donc par le cycle des prédateurs, on doit donc parcourir la matrice des prédateurs pour cela il faut trois boucles imbriquées, les deux premières pour parcourir la matrice, et la troisième pour parcourir chaque arrayList en entier. Donc pour chaque prédateur on regarde s'il doit mourir, s'il meurt on le supprime de l'ArrayList. S'il ne meurt pas alors on le déplace, pour cela on utilise la méthode *seDeplacer()*. Ensuite on utilise la méthode *predation()* pour supprimer les proies qu'il mange, en prenant en compte ses nouvelles coordonées. Enfin on regarde s'il doit se reproduire en comparant sa date de reproduction à sa date actuelle, s'il doit se reproduire on utilise la méthode *progeniture()*

qui nous renvoie un prédateur que nous positionnons à l'endroit indiqué par les nouvelles coordonnées du prédateur.

Enfin pour le cycle des proies on effectue la même opération mais cette fois-ci sans la prédation.

Pour finir nous faisons un parcours simultané des deux matrices et l'on fait se déplacer "physiquement" les proies dans la matrice (voir l'annexe A).

6.1.2 La méthode *predation()*

Cette méthode permet de supprimer les proies que le prédateur mange. Pour cela on fait un parcours partiel de toute la zone d'influence du prédateur. On fait donc deux boucles "tant que" pour parcourir la matrice de la zone d'influence et si "n" qui est le nombre de proies mangées est égal à un alors on sort de ces boucles. Dans celles-ci, si une case comporte une proie alors on la supprime et on incrémente n. Cette construction permet un futur paramétrage du nombre de proies que peut manger un prédateur en une seule période. Il suffirait de changer la condition pour mettre par exemple "n=2", alors le prédateur mangerait deux proies à chaque tour. A la fin de ce parcours on envoie au prédateur le nombre de proies qu'il a mangé pour qu'il incrémente son "lambdareproduction" (voir l'annexe A).

6.2 Les classes EtreVivant, Predateur et Proie

6.2.1 La méthode *seDeplacer()*

Pour simuler un déplacement aléatoire, nous utilisons la méthode *Math.random()* qui permet d'avoir un nombre aléatoire entre zéro et un. Ensuite nous divisons un par huit, puis pour chaque tranche de la division il y a une direction. Alors on ajoute ou soustrait la vitesse suivant la direction. On finit par appliquer la méthode *coordonneDansLeMonde()* à chaque coordonnée pour être sûr que l'être vivant se trouve dans la matrice (voir l'annexe B).

6.2.2 La méthode *progeniture()*

Pour les prédateurs, cette méthode se contente de créer une nouvelle date de reproduction en fonction de la date actuelle puis de renvoyer un prédateur qui a les mêmes coordonnées que le prédateur courant, mais sa date de naissance est la date actuelle. Pour les proies la manoeuvre est identique sauf que la proie n'est pas créée au même endroit, on choisit un x et un y dans la zone ponte aléatoirement. Ensuite on ajuste ces coordonnées avec la méthode *coordonneDansLeMonde()* (voir les annexes C et D).

Chapitre 7

La programmation graphique

7.1 La classe PanneauGraphe

Dans ce panneau, nous utilisons la librairie JFreeChart. Elle consiste en la création d'un graphique. Dans notre cas, il représentera l'évolution des proies et des prédateurs en fonction du temps.
Nous créons un objet "dataset" étant une série de collections. Puis, nous lui ajoutons les séries de proies et de prédateurs après les avoir créées.

```
this.dataset = new XYSeriesCollection();
this.predateurs = new XYSeries("Predateurs");
this.proies = new XYSeries("Proies");
this.dataset.addSeries(this.predateurs);
this.dataset.addSeries(this.proies);
```

Cette classe implémente trois méthodes. Les deux premières *ajoutPointProies()* et *ajoutPointPredateurs()* consistent en l'ajout de la nouvelle population de proies et de prédateurs dans le graphique.

On invoque la méthode *add()* sur les séries de proies et de prédateurs. Celle-ci prend deux paramètres, le temps (en abscisse) et l'effectif de la population (en ordonnée).

```
this.predateurs.add((double) this.m.getDateActuelle(),
(double) this.m.tailleTotalePred());

this.proies.add((double) this.m.getDateActuelle(),
(double) this.m.tailleTotaleProie());
```

Ces deux méthodes sont appelées dans la classe RecepteurControle.
La troisième méthode *setMonde()* sert pour la réinitialisation du graphique. Elle prend en paramètre un monde et remplace l'ancien monde par le nouveau. Ainsi on efface le graphique et si on relance la simulation, à celui-ci sont affectées les valeurs du nouveau monde.

7.2 La classe PanneauVisualisation

Ce panneau sert à la visualisation du système en temps réel. Il prend en paramètre le monde, car dans cette classe il s'agira de dessiner les proies et les prédateurs qui se trouvent dans les matrices d'ArrayList.

La méthode *paintComponent(Graphics g)* va nous permettre de réaliser cette opération. Cette méthode est constituée de deux boucles qui vont parcourir les populations de proies et de prédateurs. Ils seront dessinés en fonction de leur position. Les proies seront dessinées en bleu et les prédateurs en rouge. Nous avons choisi les mêmes couleurs pour le graphique, ainsi l'utilisateur pourra identifier dans le panneau Visualisation et dans le panneau Graphe les proies et les prédateurs de la même couleur.

Exemple pour les proies :

```
for (int i=0; i<this.m.getProies().length; i++)
{
        for (int j=0; j<this.m.getProies()[i].length; j++)
        {
                if (this.m.getProies()[i][j].size() != 0)
                {
                        g.setColor(Color.BLUE);
                        g.fillRect(i*tailleCelulle+5, j*tailleCelulle+5,
                        tailleCelulle, tailleCelulle);
                }
        }
}
```

Ensuite, nous dessinons quatre lignes faisant le tour du panneau de visualisation, afin d'avoir un rendu plus esthétique.

```
g.setColor(Color.BLACK);
g.drawLine(5, 5, 5, 405);
g.drawLine(5, 5, 405, 5);
g.drawLine(405, 5, 405, 405);
g.drawLine(5, 405, 405, 405);
```

7.3 La classe RecepteurControle

7.3.1 La méthode *actionPerformed()*

La classe RecepteurControle écoute les principaux événements, c'est-à-dire les clics sur les quatre boutons de contrôle qui sont "valider", "lecture", "pause" et "réinitialiser". Cette classe a accès à tous les composants permettant de changer les paramètres, ainsi qu'au panneau du graphique et au panneau visualisation.

Si on clique sur "valider" alors ce récepteur récupère les nombres entrés dans les boites de texte, si ce ne sont pas des nombres qui sont entrés, on déclenche une exception qui affiche une erreur. Ensuite s'il n'y a pas d'erreur, on crée le monde avec les valeurs récupérées. On envoie ce monde au panneau visualisation

et au panneau graphe. On actualise le panneau graphe en ajoutant le nombre de proies et de prédateurs initial dans les données. Puis on utilise *repaint()* sur ces deux panneaux pour actualiser l'affichage.

En appuyant sur lecture cela désactive tous les composants permettant la sélection des paramètres puis on lance une tâche qui s'occupe de faire évoluer le jeu (voir section 7.3.2).

Quand on appuie sur le bouton "pause" alors on arrête la tâche, puis on réactive tous les paramètres que l'utilisateur peut changer au cours d'une même simulation.

Enfin, si on actionne le bouton "réinitialiser" alors on arrête la tâche, on réactive tous les paramètres, puis on remet le monde à zéro. On envoie ce monde aux panneaux graphe et visualisation puis on réactualise ceux-ci (voir annexe E).

7.3.2 La méthode *run()*

Cette méthode *run()* permet de créer une tâche mais pourquoi en créer une ? Lors de la première implémentation, nous ne l'avions pas créée mais lors du lancement cela faisait "planter" le logiciel ceci était du à la boucle infinie. Le programme bouclait et n'affichait rien. En mettant un nombre de tour de boucle défini à l'avance, nous nous sommes aperçus que le programme calculait tout, puis affichait uniquement le résultat de la dernière évolution. La seule solution est de créer une tâche qui s'occupe de la boucle, indépendamment du reste du programme. Cela permet donc d'avoir un affichage de l'évolution des populations en temps réel, car cet affichage est géré dans une tâche indépendante de la boucle. Elle n'attend donc pas que la boucle soit terminée pour afficher les changements (voir annexe E).

Chapitre 8

Discussion

Résultats obtenus et résultats attendus

Ce projet remplit la totalité des fonctionnalités prévues au début de celui-ci. Listons ces fonctionnalités.

- Implémentation d'un noyau correspondant à la simulation du modèle de Lotka-Volterra
- Implémentation graphique permettant à l'utilisateur de voir l'évolution du système en temps réel
- Implémentation d'une interface utilisateur qui lui permet de modifier des paramètres agissant sur le système
- Implémentation d'un graphique permettant de visualiser l'évolution des proies et des prédateurs en temps réel

Erreur commise

Dans un premier temps, nous avions établi une analyse préliminaire qui nous a rapidement permis de travailler sur la programmation. Or cette analyse s'est vue incomplète. Nous avons donc, dû reprendre celle-ci et établir les solutions aux problématiques qui se sont présentées à nous. Ensuite, nous avons pu reprendre la programmation sur une analyse solide et nous avons vu que cela nous facilitait la tâche. Cette erreur a été commise car nous manquions de recul et d'expérience. Si nous devions réaliser un autre projet, nous commencerions par consacrer le temps qui est nécessaire à l'analyse avant de commencer la programmation.

Optimisations possibles du projet

A la fin de ce projet, notre tuteur nous a proposé de changer la façon dont les prédateurs mangent. Il s'agissait de ne leur faire manger qu'une seule proie dans leur zone d'influence. Nous avons implémenté cette modification, mais on aurait pu changer l'interface de l'utilisateur afin qu'il paramètre le nombre de proie qu'un prédateur peut manger.
Dans un deuxième temps, nous avons pensé permettre à l'utilisateur de modifier la couleur des proies et des prédateurs. Or il fallait modifier les couleurs de ceux-ci pour l'évolution du système ainsi que sur le graphique. Nous n'avons pas souhaiter faire ces modifications par manque de temps.

Conclusion

Bilan personnel

Dans un premier temps ce projet nous a apporté des techniques de travail. En effet, nous avons appris à partager nos opinions et à les synthétiser afin de d'obtenir un résultat satisfaisant.

Dans un deuxième temps, nous avons appris à gérer notre temps pour ce projet. C'est-à-dire que nous avons évalué la durée de chaque tâche et fourni un travail en conséquence. Nous avons rempli nos objectifs dans les temps impartis et ainsi nous avons pu mener à terme ce projet. Cela nous tenait à cœur car nous voulions voir le résultat final de ce projet.

Continuité du projet

Nous avons créé ce projet de telle sorte qu'un utilisateur lambda puisse comprendre l'ensemble des paramètres à modifier. Celui-ci n'a pas besoin de réfléchir quant aux valeurs à donner aux paramètres. Il doit simplement agir sur un bouton que ce soit une liste déroulante ou une barre graduée. Cette implémentation est un choix volontaire de notre part. En revanche, nous pourrions proposer une deuxième version qui serait utilisée par exemple par des scientifiques. Dans ce cas-là, les paramètres devront prendre des valeurs très précises pour permettre à ceux-ci d'exploiter les résultats obtenus. Dans cette optique, nous aurions créé des boîtes de dialogue afin de permettre à ces scientifiques de rentrer la valeur exacte des paramètres qu'ils souhaitent modifier.

Ce genre de logiciel pourrait avoir des applications éducatives en matière d'écologie. En effet, on peut se rendre compte en l'utilisant que l'équilibre établi dans un écosystème est très fragile et que le changement d'un seul paramètre peut entraîner la destruction de celui-ci. Une version plus scientifique et plus proche de la réalité pourrait permettre aux chercheurs de prévoir les effets de certains changements dans un écosystème.

Bibliographie

[1] Dia. http ://www.live.gnome.org/Dia.

[2] Les equations de lotka-volterra. http ://fr.wikipedia.org/wiki/Equations_de_Lotka-Volterra.

[3] Corentin. Beamer, 2006. http ://www.math.unicaen.fr/∼pontreau/Documents/sem-jeuns.pdf.

[4] Konrad Florczak. Formation latex, 2005.

Annexe A

La classe Monde : les méthodes *evolution()* et *predation()*

```
/**
* @action
*       Permet de faire evoluer le monde donc de faire se deplacer,
manger, se reproduire et mourir les predateurs.
*       Ensuite de faire se deplacer et se reproduirent les proies.
*/
public void evolution()
{
        //periode suivante
        cpt.jourSuivant();

        for (int i=0; i<this.lesPredateurs.length; i++)
        {
                for (int j=0; j<this.lesPredateurs[i].length; j++)
                {
                for (int k=this.lesPredateurs[i][j].size()-1; k>=0; k--)
                {
                Predateur p = lesPredateurs[i][j].get(k);

                //soit il meurt
                if(cpt.getDate() >= p.getDateDeMort())
                {
                        this.lesPredateurs[i][j].remove(k);
                }
                else //sinon il vit
                {
                        //il se deplace
                        p.seDeplacer(Predateur.vitesse);

                        //il mange
                        this.predation(p);
```

```
                        //se reproduit
                if (cpt.getDate() >= p.getDateDeReproduction())
                {
                this.lesPredateurs[p.getX()][p.getY()].add(
                p.progeniture(cpt.getDate()));
                }
                }
                }
                }
        }

//les proies
for (int i=0; i<this.lesProies.length; i++)
{
        for (int j=0; j<this.lesProies[i].length; j++)
        {
                for (int k=this.lesProies[i][j].size()-1; k>=0; k--)
                {
                        Proie o = lesProies[i][j].get(k);

                        //elle se deplace
                        o.seDeplacer(Proie.vitesse);

                        //et se reproduit
                        if (cpt.getDate() >= o.getDateDeReproduction())
                        {
                        this.lesProies[o.progeniture(cpt.getDate()).getX()]
                        [o.progeniture(cpt.getDate()).getY()]
                        .add(o.progeniture(cpt.getDate()));
                        }
                }
        }
}

//On les met a leur places reelle dans la matrice
for (int i=0; i<this.lesPredateurs.length; i++)
{
        for (int j=0; j<this.lesPredateurs[i].length; j++)
        {
                for (int k=this.lesPredateurs[i][j].size()-1; k>=0; k--)
                {
                        //les predateurs se deplacent
                        Predateur p = lesPredateurs[i][j].get(k);
                        lesPredateurs[i][j].remove(k);
                        lesPredateurs[p.getX()][p.getY()].add(p);
                }
                for (int k=this.lesProies[i][j].size()-1; k>=0; k--)
                {
                        //les proies se deplacent
                        Proie p = lesProies[i][j].get(k);
                        lesProies[i][j].remove(k);
                        lesProies[p.getX()][p.getY()].add(p);
                }
        }
```

```
}
}

/**
* @param lesProies
* @return
*       supprime les proies que le predateur a manger.
*/
void predation(Predateur p)
{
        int x = p.getX()-Predateur.zoneInfluence;
        int x2 = p.getX()+Predateur.zoneInfluence;
        int n = 0;

        while (x <= x2 && n!=1)
        {
        int y = p.getY()-Predateur.zoneInfluence;
        int y2 = p.getY()+Predateur.zoneInfluence;
        while (y <= y2 && n!=1)
        {
        if (lesProies[p.coordonneeDansLeMonde(x)]
        [p.coordonneeDansLeMonde(y)].size() !=0)
        {
                n++;
                lesProies[p.coordonneeDansLeMonde(x)]
                [p.coordonneeDansLeMonde(y)].remove(0);
        }
        y++;
        }
        x++;
        }
        if (n!=0)
        {
                p.augmenterCapacitesReproductrice(n);
        }
}
```

Annexe B

La classe EtreVivant : les méthodes *seDeplacer()* et *coordonneeDansLeMonde()*

```
/**
* @param vitesse
* @action
*       change les coordonnees suivant une vitesse
donnee et un sens choisi aleatoirement.
*/
void seDeplacer(int vitesse)
{
        double r = Math.random();
        int i = this.x;
        int j = this.y;
        if (r < 0.125) //en haut a gauche
        {
                i = i - vitesse;
                j = j - vitesse;
        }
        else if (r < 0.25) //en haut
        {
                i = i - vitesse;
        }
        else if (r < 0.375) //en haut a droite
        {
                i = i - vitesse;
                j = j + vitesse;
        }
        else if (r < 0.5) //a gauche
        {
                j = j - vitesse;
        }
        else if (r < 0.625) //a droite
        {
                j = j + vitesse;
```

```
        }
        else if (r < 0.75) //en bas a gauche
        {
                i = i + vitesse;
                j = j - vitesse;
        }
        else if (r < 0.875) //en bas
        {
                i = i + vitesse;
        }
        else //en bas a droite
        {
                i = i + vitesse;
                j = j + vitesse;
        }

        this.x = (this.coordonneeDansLeMonde(i));
        this.y = (this.coordonneeDansLeMonde(j));
}

/**
* @param n
* @return
*       Si l'entier n n'est pas inclu entre 0 et la taille du monde
-1 alors on l'ajuste pour qu'il y soit inclue.
*/
int coordonneeDansLeMonde(int n)
{
        while (n < 0)
        {
                n = n+Monde.taille;
        }
        while (n >= Monde.taille)
        {
                n = n-Monde.taille;
        }
        return n;
}
```

Annexe C

La classe Proie : la méthode *progeniture()*

```
/**
* @return
*       retourne une proie dont les coordonnees sont comprises
*       dans la zone de ponte et dont la date de naissance
*       correspond a la date actuelle.
*/
Proie progeniture(double date)
{
        this.setDateDeReproduction(Proie.lambdaReproduction, date);
        int higher = Proie.zonePonte+1;
        int lower = -Proie.zonePonte;
        int i = (int)(Math.random()*(higher-lower)) + lower;
        int j = (int)(Math.random()*(higher-lower)) + lower;
        return new   Proie(this.coordonneeDansLeMonde(this.getX()+i),
        this.coordonneeDansLeMonde(this.getY()+j), date);
}
```

Annexe D

La classe Predateur : la méthode *progeniture()*

```
/**
* @return
*       retourne un predateur avec les memes coordonnees
et la date actuelle comme date de naissance.
*/
Predateur progeniture(double date)
{
this.dateDerniereProgeniture = date;
this.setDateDeReproduction(this.lambdaReproduction, date);
return new Predateur(this.getX(),this.getY(),date);
}
```

Annexe E

La classe RecepteurControle : les méthodes *actionPerformed()* et *run()*

```
/**
* receptionne les actions sur les boutons
*/
public void actionPerformed(ActionEvent e)
{
        if (e.getActionCommand() == "valider")
        {
                try
                {
                //On recupere les valeurs des JTextFields
                RecepteurControle.nbPredateurs =
                Integer.parseInt(this.boiteNbPredateurs.getText());
                RecepteurControle.nbProies =
                Integer.parseInt(this.boiteNbProies.getText());
                PanneauMonde.textErreur.setVisible(false);

                //On cree le monde puis initialise toutes les vues.
                this.leMonde = new Monde(RecepteurControle.nbProies,
                RecepteurControle.nbPredateurs);
                this.pVisualisation.setMonde(this.leMonde);
                this.pGraphe.setMonde(this.leMonde);
                this.pGraphe.ajoutPointPredateurs();
                this.pGraphe.ajoutPointProies();
                this.pVisualisation.repaint();
                this.pGraphe.repaint();
                }
                catch (NumberFormatException x)
                {
                //Si ce n'est pas un nombre qui est rentre
```

```
            //on affiche une erreur
            PanneauMonde.textErreur.setVisible(true);
            }
    }
    else if (e.getActionCommand() == "play")
    {
            //On desactive tous les parametres
            this.valider.setEnabled(false);
            this.listeTaille.setEnabled(false);
            this.listeVitessePredateur.setEnabled(false);
            this.listeVitesseProie.setEnabled(false);
            this.listeZoneDePonte.setEnabled(false);
            this.listeZoneDInfluence.setEnabled(false);
            this.tauxAgmentationDeReproduction.setEnabled(false);
            this.tauxDeMort.setEnabled(false);
            this.tauxDeReproduction.setEnabled(false);
            this.boiteNbPredateurs.setEnabled(false);
            this.boiteNbProies.setEnabled(false);
            //On lance la tache
            bool = true;
            es.execute(this);
    }
    else if (e.getActionCommand() == "pause")
    {
            //On arrete la tache
            bool = false;
            //On reactive certains parametres
            this.listeVitessePredateur.setEnabled(true);
            this.listeVitesseProie.setEnabled(true);
            this.listeZoneDePonte.setEnabled(true);
            this.listeZoneDInfluence.setEnabled(true);
            this.tauxAgmentationDeReproduction.setEnabled(true);
            this.tauxDeMort.setEnabled(true);
            this.tauxDeReproduction.setEnabled(true);
    }
    else if (e.getActionCommand() == "reinit")
    {
            //On desactive la tache
            bool = false;
            //On reactive tous les parametres
            this.valider.setEnabled(true);
            this.listeTaille.setEnabled(true);
            this.listeVitessePredateur.setEnabled(true);
            this.listeVitesseProie.setEnabled(true);
            this.listeZoneDePonte.setEnabled(true);
            this.listeZoneDInfluence.setEnabled(true);
            this.tauxAgmentationDeReproduction.setEnabled(true);
            this.tauxDeMort.setEnabled(true);
            this.tauxDeReproduction.setEnabled(true);
            this.boiteNbPredateurs.setEnabled(true);
            this.boiteNbProies.setEnabled(true);
            //On reinitialise le monde
            this.leMonde = new Monde(0, 0);
            this.pVisualisation.setMonde(this.leMonde);
```

```
                this.pGraphe.setMonde(this.leMonde);
                this.pVisualisation.repaint();
                this.pGraphe.repaint();
        }
}

/**
* @action
*       permet l'evolution continue des etre-vivants
*/
public void run()
{
        while (bool)
        {
                this.leMonde.evolution();
                this.pGraphe.ajoutPointPredateurs();
                this.pGraphe.ajoutPointProies();
                this.pVisualisation.repaint();
                this.pGraphe.repaint();

                try
                {
                        Thread.sleep(100);
                } catch (InterruptedException e)
                {
                        e.printStackTrace();
                }
        }
}
```

Résumé

Pendant la Première Guerre Mondiale, l'intensité de la pêche a diminué, pourtant on a observé une diminution de la population des sardines. Pourquoi n'ont-elles pas proliféré ? Deux chercheurs Lotka et Volterra ont mis en place un modèle qui explique ce phénomène : dans un premier temps les sardines ont proliféré, leurs prédateurs qui ne manquaient plus de nourriture ont mangé la majorité de celles-ci et ont ainsi pu se développer rapidement.

Dans le cadre de ce projet, nous avons créé un système proie-prédateur afin d'observer l'évolution des proies et des prédateurs dans un écosystème en ayant la possibilité de changer un certain nombre de paramètres sur celui-ci.

Mots-clés : Lotka, Volterra, système proie-prédateur, java

Summary

During the First World War, the intensity of fishing decreased, however one observed a reduction in the population of sardines. Why didn't they proliferate ? Two researchers Lotka and Volterra have set up a model which explains this phenomenon : initially the sardines proliferated, their predatory which didn't miss any more food, ate the majority of this one and thus could develop quickly.

Within the framework of this project, we created a prey-predatory system in order to observe the evolution of the preys and the predatory ones in an ecosystem by having the possibilities of changing a certain number of parameters on this one.

Key-words : Lotka-Volterra, prey-predatory system, java