Mélanie Bats - Cyril Coquilleau
Thomas Petazzoni - Julien Rosener

Approche réactive pour le traitement du problème Proie(s)/Prédateurs, simulations avec MadKit

IA54 - GL53

Automne 2003

Suiveur IA54 : M. Olivier Simonin
Suiveur GL53 : M. Thomas Lissajoux

Table des matières

Introduction

Notre projet avait pour point de départ une simulation de type *proie / prédateur* présente dans MadKit et utilisant le jeu de classes TurtleKit.

Ce projet étant couplé avec l'UV GL53, l'objectif du projet était double :

- améliorer la stratégie des proies et des prédateurs en se limitant à des agents purement réactifs sans communication et donner la possibilité de paramétrer les simulations pour comparer différentes stratégies
- repenser l'interface graphique afin de donner une représentation cohérente de la simulation et des possibilités de paramétrage

Chapitre 1

Modèle RIO

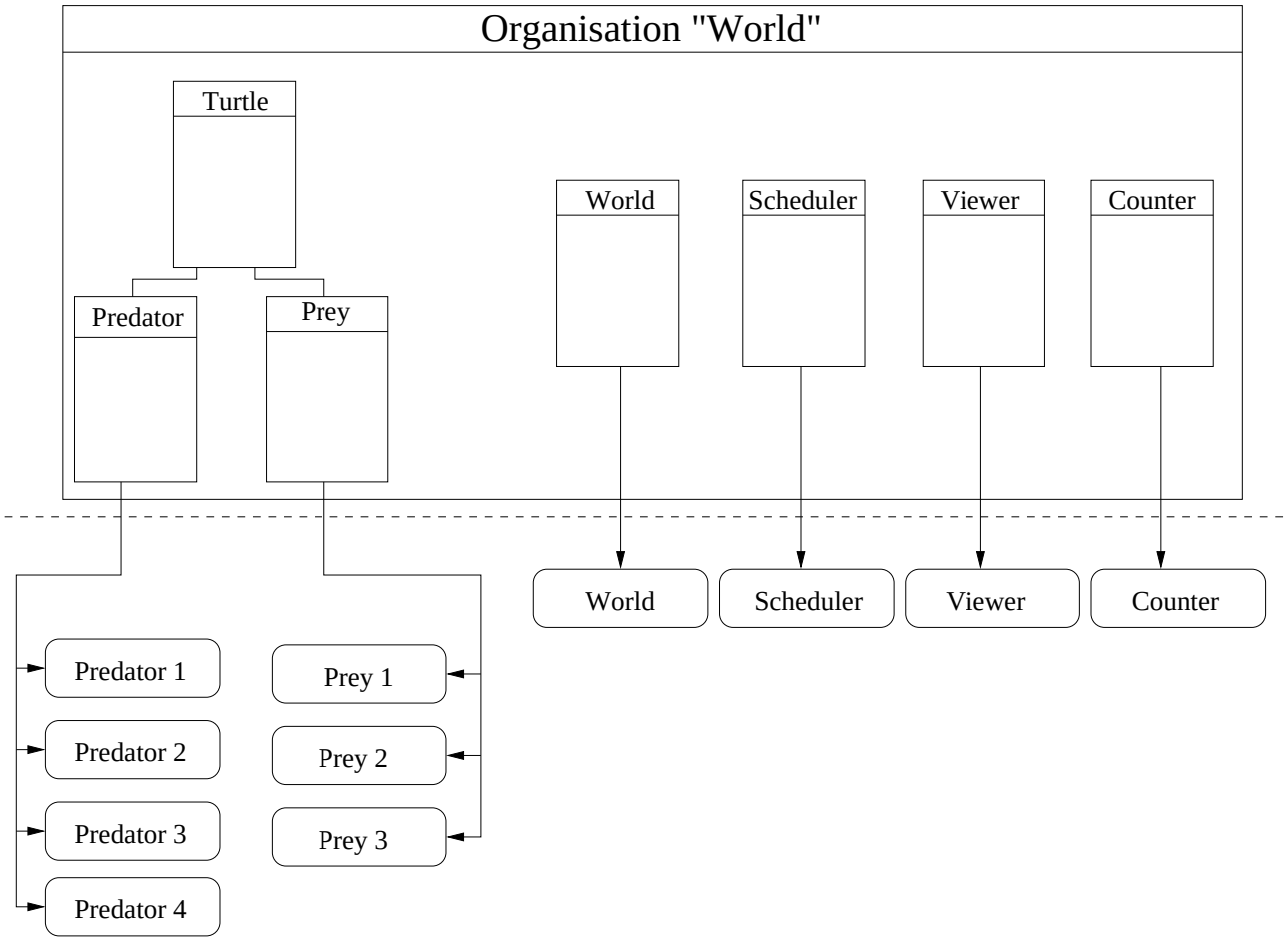

FIG. 1.1 – Modèle RIO du SMA

Notre système multi-agents comporte différents rôles :

- Le rôle *Turtle* regroupe les caractéristiques communes aux proies et prédateurs. Les rôles *Prey* et *Predator* héritent de ce rôle et correspondent aux agents qui nous intéressent dans la simulation : les proies et les prédateurs. Ils ne communiquent pas entre eux : ils ont un simple comportement réactif par rapport à ce qui se passe dans l'environnement.
- Le rôle *World* correspond à l'agent s'occupant de gérer l'environnement : emplacement des *turtles* et leur déplacement dans cet environnement.

– Le rôle *Scheduler* correspond à l'agent s'occupant d'ordonnancer la simulation.
– Le rôle *Viewer* correspond à l'agent chargé de représenter de manière graphique l'environnement.
– Le rôle *Counter* correspond à l'agent chargé d'extraire de la simulation des statistiques sur le nombre de proies et prédateurs.

Chapitre 2

Stratégies des proies et prédateurs

2.1 Stratégies des proies

2.1.1 Stratégie *Fabien Michel*

La stratégie des proies de *Fabien Michel* est très sommaire : elles se déplacent au hasard, sans tenir compte de la présence des prédateurs qui pourraient les poursuivre.

Nous avons décidé de transférer intégralement cette stratégie dans notre panel de stratégies afin de pouvoir la comparer avec les autres que nous avons mises au point.

2.1.2 Stratégie fuite

Fuire avec un aléatoire pur – méthode utilisée par *Fabien Michel* pour ses proies – nous semblant quelque peu faible, nous avons décidé d'implémenter une seconde stratégie intégrant une notion de fuite face aux prédateurs. Dans ce mode de fuite, la proie fuira à l'opposé du prédateur le plus proche de lui. D'autre part cette fuite peut permettre, au gré de l'utilisateur avec l'aide d'une option, un regroupement des proies pour former un groupe compact. Ceci a pour objectif de réduire la dispersion des proies sur le monde et ainsi réduire les chances des proies d'être repérées par les prédateurs.

L'algorithme adopté est développé dans le listing **Algorithm 1**.

Cet algorithme exploite une machine à états (composée ici de trois états : *promenade*, *groupe* et *fuite*), chaque changement d'état étant provoqué par une modification de l'environnement.

L'algorithme exploite deux procédures, *calculeVecteurRepulsion(ensemble de tortues)* et *calculeVecteurAttraction(ensemble de tortues)* qui retournent respectivement le vecteur répulsion qu'effectue la tortue la plus proche et d'autre part le vecteur attraction qu'effectue la tortue la plus proche. Ces deux procédures sont donc très similaires.

L'algorithme de la procédure *calculeVecteurRepulsion(ensemble de tortues)* se trouve dans le listing **Algorithm 2**.

2.2 Stratégies des prédateurs

2.2.1 Stratégie *Fabien Michel*

Pour la stratégie adoptée par les prédateurs, *Fabien Michel* a décidé d'utiliser une stratégie basée sur la mise en accord de tous les prédateurs. En effet, ceux-ci évoluent tous dans une même direction quand aucune proie n'est dans leur rayon de vision et changent de comportement en étant attirés vers la proie lorsqu'ils en aperçoivent une.

Cette mise en accord semble sortir du cadre d'une architecture réactive et sans communication (qui était le cadre de notre projet) car il est difficile dans la réalité de s'accorder sans communication préalable.

Algorithm 1 Stratégie de fuite pour les proies

```
1: perceptionEnvironnement()                ▷ Detecte les proies et predateurs dans le rayon de vision
2: if etat = promenade ou etat = groupe then
3:     if taille(predateurs) > 0 then
4:         vecteurrepulsion ← calculeVecteurRepulsion(predateurs)
5:         etat ← fuite
6:     else if taille(proies) > 0 et modegroupe = oui et taille(proies) ≤ maxproiespargroupe then
7:         vecteurattraction ← calculeVecteurAttraction(proies)
8:         etat ← groupe
9:     else
10:        etat ← promenade
11:    end if
12: else
13:    if etat = fuite then
14:        vecteurrepulsion ← calculeVecteurPredateurs(predateurs)
15:    else
16:        etat ← promenade
17:    end if
18: end if
19: if etat = promenade then                ▷ à partir d'ici, on oriente la proie
20:    rotationAleatoire(entre −50 a 50 degres)
21: else if etat = groupe then
22:    positionneAngle(obtientAngle(vecteurattraction))
23: else
24:    positionneAngle(obtientAngle(vecteurrepulsion))
25: end if
```

Algorithm 2 Procédure calculeVecteurRepulsion : Calcule le vecteur entre la tortue courante et la plus proche d'elle

```
1: procedure CALCULEVECTEURREPULSION(tortuecourante, ensemble de tortues)
2:     distmin ← 99999
3:     for i ← 0, taille(ensemble de tortues) do
4:         if ensemble de tortues[i] ≠ tortuecourante then
5:             dist ← norme(tortuecourante,ensemble de tortues[i])
6:             if dist < distmin then
7:                 vecteurmin ← vecteur(tortuecourante,ensemble de tortues[i])
8:                 distmin ← dist
9:             end if
10:        end if
11:        i ← i + 1
12:    end for
13:    retourne(vecteurmin)
14: end procedure
```

Nous avons tout de même choisi de garder cette stratégie pour la comparer aux notres, qui seront expliquées ci-dessous.

Nous avons noté plusieurs choses à propos de cette stratégie :

- en monde non torique, les prédateurs se bloquaient tous d'un même côté du monde (ceci étant dû à la mise en accord de tous les prédateurs sur une même direction).
- la performance de la stratégie est due au fait que tous les prédateurs vont dans la même direction : quand une proie est tuée par un groupe, celui-ci devient un bloc compact qui évolue dans le monde rapidement, et dès qu'une proie est en vue elle est instantanément tuée par les multiples prédateurs.

2.2.2 Stratégie basique

Cette stratégie est la plus simple de celles qui ont été faites pour les prédateurs. Ceux-ci auront l'opportunité de se réunir pour former des groupes si désiré et ainsi venir plus rapidement à bout des proies que s'ils étaient isolés. Le nombre de prédateurs par groupe est modifiable.

Cette stratégie exploite une machine à états (*promenade*, *poursuite*, *groupe* et *fuite*). La poursuite intervenant lorsqu'une proie est aperçue et donc pourchassée alors que la fuite intervient lorsque le nombre de prédateurs dans un groupe devient trop important.

Chaque prédateur possède une direction fixe (parmi les 8 points cardinaux) qu'il suivra dans l'état *promenade* et *fuite*. Cela lui évitera notamment de rester sur place à cause du *mouvement brownien*.

Une procédure *vecteurMeilleureProie(predateurs,proies)* a été mise en place pour trouver la proie la plus intéressante pour les prédateurs. Après l'essai de différentes méthodes comme la somme des vecteurs d'attraction pondérés en fonction de la distance ou bien la recherche de la plus proche proie, il s'est avéré que la solution la plus efficace était de faire aller le prédateur vers la proie qui a autour d'elle le plus de prédateurs, afin que celle-ci soit tuée le plus vite possible. L'algorithme de cette fonction ressemble sensiblement à celui de la recherche de la plus proche proie.

2.2.3 Stratégie avec *timeout*

Cette stratégies est une évolution de la stratégie précédente.

Elle corrige le fait que certains prédateur souffrent de problèmes de *deadlocks*. Ce phénomène apparaît quand un prédateur poursuit trop longtemps une proie alors qu'il est seul. Cela risque de bloquer l'évolution du monde si les prédateurs sont en nombre inférieur par rapport aux proies car chacun risquerait d'etre "obnubilé" par sa cible et de ce fait empêchant que 4 prédateurs puissent se réunir et tuer la proie.

Pour cela une notion de *timeout* (temps maximal) a été introduite. Ceci consiste à laisser le prédateur chasser une proie pendant un certain laps de temps (géré par un compteur *nbToursEtatCourant*), puis une fois celui-ci dépassé, le prédateur quitte son état de *poursuite* pour passer à l'état *fuite* et ainsi délaisser son objectif pour en trouver un autre plus facile. Des seuils maximum à ne pas dépasser ont été mis en place :

- *nbToursMaxGroupe* : le nombre maximum de cycles durant lequels l'agent restera dans l'état *groupe*. Cela permet à un groupe statique de s'exploser et de repartir en mode *promenade*.
- *nbToursMaxFuite* : le nombre de cycles que doit observer l'agent en cas de fuite.
- *nbToursMaxPoursuite* : le nombre de cycles total durant lesquels l'agent a le droit de poursuivre sa proie avant de basculer dans l'état *fuite* pour lui éviter d'être obnubilé par sa proie.

Les deux stratégies ont leurs avantages et inconvénients, car un *timeout* trop petit anihilerait toute chance de voir des proies tuées, les prédateurs quittant les proies après un court temps. Cela dépend des différents paramètres de la simulation comme entre autres le nombre de proies et prédateurs.

2.2.4 Stratégie avec *timeout* et direction aléatoire

Cette stratégie est une évolution plus poussée de la stratégie avec timeout. Auparavant un prédateur possédait une direction fixe (on l'appellera *absoluteDirection*) de déplacement parmi les 8 points cardinaux en mode *promenade* et n'en changeait pas durant la simulation. Ce mode permet aux prédateurs de changer aléatoirement de direction et de favoriser éventuellement la capture de proies.

Ce changement intervient lors des changements d'état du prédateur effectués à l'aide de procédure *changeEtat(etat)*. Cette procédure remet également automatiquement à 0 le compteur *nbToursEtatCourant*

L'algorithme de cette stratégie (englobant aussi les stratégies du mode basique et du mode avec *timeout*) est développé dans le listing **Algorithm 3**.

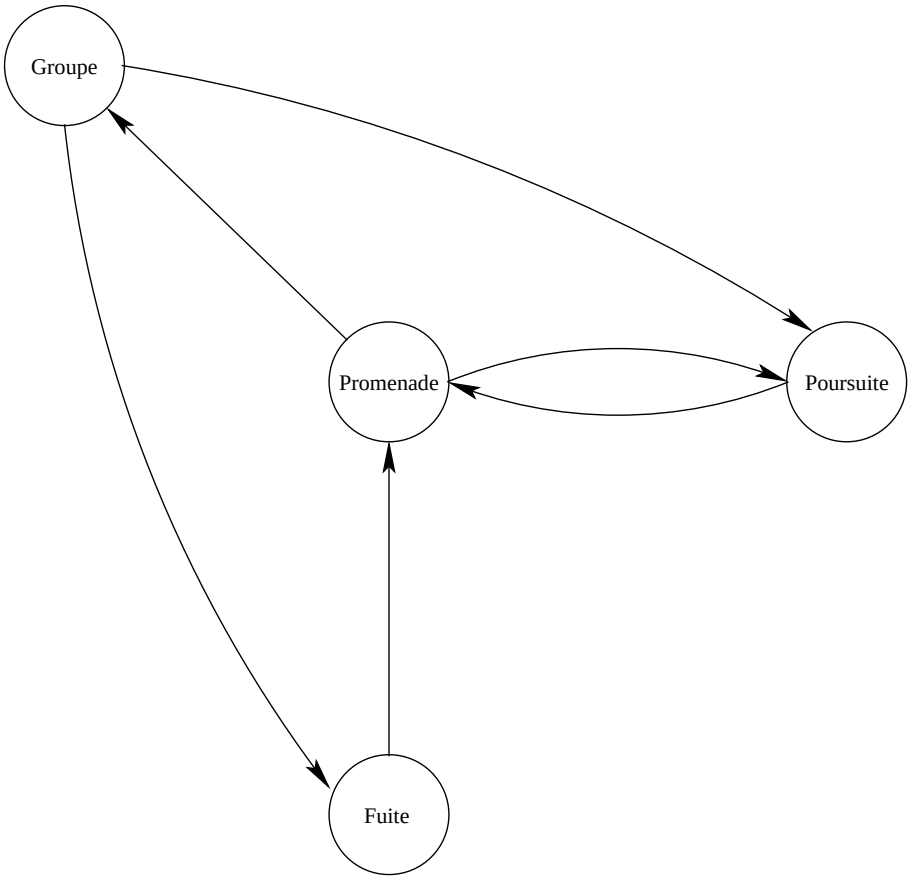

FIG. 2.1 – Graphe à états du prédateur

2.3 Propositions pour une stratégie plus efficace

2.3.1 Élection d'un chef par immobilité

Le problème de stratégies sans communication est l'impossibilité de faire s'accorder les différents agents afin d'être les plus efficaces possible. Ainsi l'élection d'un chef de troupe qui guiderait tous les autres éléments est quasi-impossible. Une solution semble toutefois réalisable. Celle-ci consisterait à laisser immobiles tous les prédateurs dans un certain rayon, puis le premier à se déplacer, vu de tous les autres prédateurs, serait élu chef de groupe. Ensuite les autres prédateurs pourraient suivre le chef de manière simple puisqu'il serait identifié.

Algorithm 3 Stratégie pour les prédateurs

```
1: perceptionEnvironnement()                ▷ Detecte les proies et predateurs dans le rayon de vision
2: nbToursEtatCourant ← nbToursEtatCourant + 1
3: if etat = promenade then
4:     if taille(proies) > 0 then
5:         vecteurattraction ← vecteurMeilleureProie(predateurs, proies)
6:         changeEtat(poursuite)
7:     else if taille(predateurs) ≥ 0 et modegroupe = oui et taille(predateurs) ≤
           maxpredateurspargroupe then
8:         vecteurattraction ← calculeVecteurAttraction(predateur)
9:         changeEtat(groupe)
10:    else
11:        etat ← promenade
12:    end if
13: else if etat = poursuite then
14:    if taille(proies) > 0 then
15:        vecteurattraction ← vecteurMeilleureProie(predateurs, proies)
16:        if nbToursEtatCourant ≥ nbToursMaxPoursuite et taille(predateurs) <
               2 et (strategie = strategietimeout ou strategie = strategietimeoutdiraleat) then
17:            changeEtat(fuite) ▷ Change d'état si la troupe est trop petite et le timeout est dépassé
18:        end if
19:    else
20:        changeEtat(promenade)
21:    end if
22: else if etat = groupe then
23:    if taille(proies) > 0 then
24:        vecteurattraction ← vecteurMeilleureProie(predateurs, proies)
25:        changeEtat(poursuite)
26:    else if taille(predateurs) ≥ 0 et modegroupe = oui et taille(predateurs) ≤
           maxpredateurspargroupe then
27:        vecteurattraction ← calculeVecteurAttraction(predateur)
28:        if nbToursEtatCourant ≥ nbToursMaxGroupe then
29:            changeEtat(promenade)
30:        end if
31:    else
32:        changeEtat(promenade)
33:    end if
34: else                                    ▷ Correspond au dernier état restant : celui de la fuite
35:    if nbToursEtatCourant ≥ nbToursMaxFuite then
36:        changeEtat(promenade)
37:    end if
38: end if
39: if etat = promenade then                ▷ à partir d'ici, on oriente le prédateur
40:    rotationAleatoire(entre −1 a 1 degres)
41: else if etat = fuite then
42:    positionneAngle(absoluteDirection)
43: else                                    ▷ ici l'état est groupe ou poursuite
44:    positionneAngle(obtientAngle(vecteurattraction))
45: end if
```

2.3.2 Communication inter-prédateurs

La solution de *l'élection d'un chef par immobilité* permet d'élire un chef de troupe, mais malheuresement celui-ci ne peut communiquer avec les prédateurs le suivant. Si une communication par messages avait pu être mise en place, ceci aurait permis la mise en place d'ordres plus complexes comme un encerclement des proies, la décision d'abandon d'une poursuite ou bien le ralliement à une autre troupe. Ceci aurait certainement pu améliorer grandement l'efficacité du système.

Chapitre 3

Réalisation

3.1 Démarche

Notre projet se base sur la simulation *HuntLauncher* disponible par défaut dans MadKit. Cette simulation repose sur l'ensemble de classes TurtleKit qui permet de réaliser simplement des simulations d'agents situés dans un environnement à deux dimensions.

Au départ, nous avons donc repris TurtleKit et les classes de *HuntLauncher* que nous avons adapté, d'une part du point de vue de l'interface graphique et d'autre part du point de vue de la stratégie des proies et prédateurs. Nous avons travaillé de cette manière jusqu'à fin décembre, en personnalisant TurtleKit et en modifiant la stratégie des agents de *HuntLauncher*.

La personnalisation de TurtleKit ne fut pas facile, car nous souhaitions réaliser diverses choses qui n'étaient pas prévues au départ. Par exemple, la personnalisation du *scheduler* fut particulièrement délicate, et passer des informations durant la simulation aux agents proies et prédateurs était difficile. Nous devions user de méthodes abstraites peu élégantes. En bref, la façon dont était structuré le programme ne nous satisfaisait pas pleinement.

A la fin décembre, suite à une rencontre avec les responsables du projet de GL53, nous nous sommes aperçus que l'interface réalisée jusqu'alors ne satisfaisait pas les attentes de l'UV GL53. Nous avons donc décidé de reprendre complètement le projet.

Le projet n'utilise plus le *desktop* de MadKit : il utilise directement le noyau MadKit. Nous avons réutilisé certaines classes de TurtleKit et créé notre propre interface graphique. Ceci nous a permis de structurer le programme de manière plus cohérente par rapport à ce que nous souhaitions obtenir, et de réaliser une interface graphique bien pensée.

Tout au long du projet, nous avons rencontré quasiment toutes les semaines notre suiveur, M. Olivier Simonin. Ces rencontres ont permi de nous guider dans le travail à accomplir.

3.2 Répartition du travail

Mélanie Bats et Thomas Petazzoni se sont concentrés sur l'aspect interface graphique et la restructuration du programme, tandis que Julien Rosener et Cyril Coquilleau ont travaillé l'aspect stratégie des proies et prédateurs.

Toutefois, les échanges entre les deux équipes ont été fréquents et réguliers, à la fois au sujet de problèmes conceptuels que de problèmes techniques de réalisation.

3.3 Aspects techniques

3.3.1 Utilisation du noyau MadKit

Le noyau MadKit a été utilisé en dehors du *desktop MadKit*, afin de créer une interface graphique entièrement personnalisée. Pour chaque simulation, nous créons un nouveau noyau MadKit, dans lequel nous éxécutons ensuite les agents :

Listing 3.1 – Noyau MadKit et agents

```
theKernel = new Kernel("simPredatorPrey");

env = new TurtleEnvironment(x, y, "hunt");
theKernel.launchAgent(env, "environnement", this, false);

pp = new PlotPanel("Proies", x*cellSize, nbPrey);

envPanel = new EnvironnementPanel(this, cellSize, withImages);

for (int i = 0; i < nbPrey; i++)
{
    Prey prey = new Prey(env,
                         (int) (Math.random()*(double)x),
                         (int) (Math.random()*(double)y));
    preyList.add(prey);

    theKernel.launchAgent(prey, "bete", env, false);
}

for (int j = 0; j < nbPredator; j++)
{
    Predator predator = new Predator(env,
                                     (int) (Math.random()*(double)x),
                                     (int) (Math.random()*(double)y));

    predatorList.add(predator);

    theKernel.launchAgent(predator, "bete", env, false);
}

viewer = new Viewer("hunt", envPanel, false);
theKernel.launchAgent(viewer, "viewer", this, false);

counter = new Counter("hunt", pp);
theKernel.launchAgent(counter, "counter", this, false);
```

3.3.2 Classes

Interface Cette classe héritant de *JFrame* est la fenêtre principale de l'application.

NouveauMondeDialog Cette classe, héritant de *JDialog*, est la boîte de dialogue permettant de paramétrer une nouvelle simulation.

CommandPanel Cette classe est un élément graphique héritant de *JPanel*. Il contient l'ensemble des composants permettant de contrôler la simulation. C'est le panneau affiché dans la partie droite de l'interface lorsqu'une simulation est en cours.

EnvironnementPanel Cette classe héritant de *JPanel* permet de représenter le monde.

PlotPanel Cette classe héritant de *JPanel* permet de tracer une courbe donnant le nombre de proies encore vivantes en fonction du temps.

MainPanel Cette classe héritant de *JPanel* est chargée de regrouper en un seul panneau les panneaux *PlotPanel*, *EnvironnementPanel* et *CommandPanel*.

Simulation Cette classe représente la simulation en cours. Elle créé le noyau MadKit, tous les agents, et permet à l'interface d'intéragir avec la simulation.

Patch Cette classe représente une unité de l'espace dans lequel se déplacent les proies et les prédateurs.

TurtleEnvironnement Cette classe implémente l'agent chargé du maintien des informations concernant le monde, en particulier la grille de *Patch* représentant l'espace.

Counter Cette classe héritant de *Watcher* permet d'observer le monde pour déterminer le nombre de proies et de prédateurs présents, et réaliser à partir de ces informations des statistiques graphiques.

Viewer Cette classe est un agent qui permet d'envoyer à *EnvironnementPanel* les requêtes de dessin des proies et prédateurs.

TurtleProbe Cette classe héritant de *Probe* permet à l'agent *viewer* de récupérer la liste de toutes les proies et de tous les prédateurs en activité.

TurtleActivator Cette classe héritant de *Activator* permet au *Scheduler* de réaliser à chaque tour de simulation, l'appel à la fonction `live()` de toutes les proies et de tous les prédateurs.

TurtleScheduler Cette classe héritant de *Scheduler* est chargée d'ordonnancer l'éxécution des agents.

Turtle Cette classe regroupe les fonctionnalités communes aux proies et aux prédateurs, notamment en ce qui concerne le déplacement dans l'environnement.

Predator Cette classe héritant de *Turtle* implémente l'agent ayant le comportement *prédateur*.

Prey Cette classe héritant de *Turtle* implémente l'agent ayant le comportement *proie*.

3.3.3 Diagramme UML

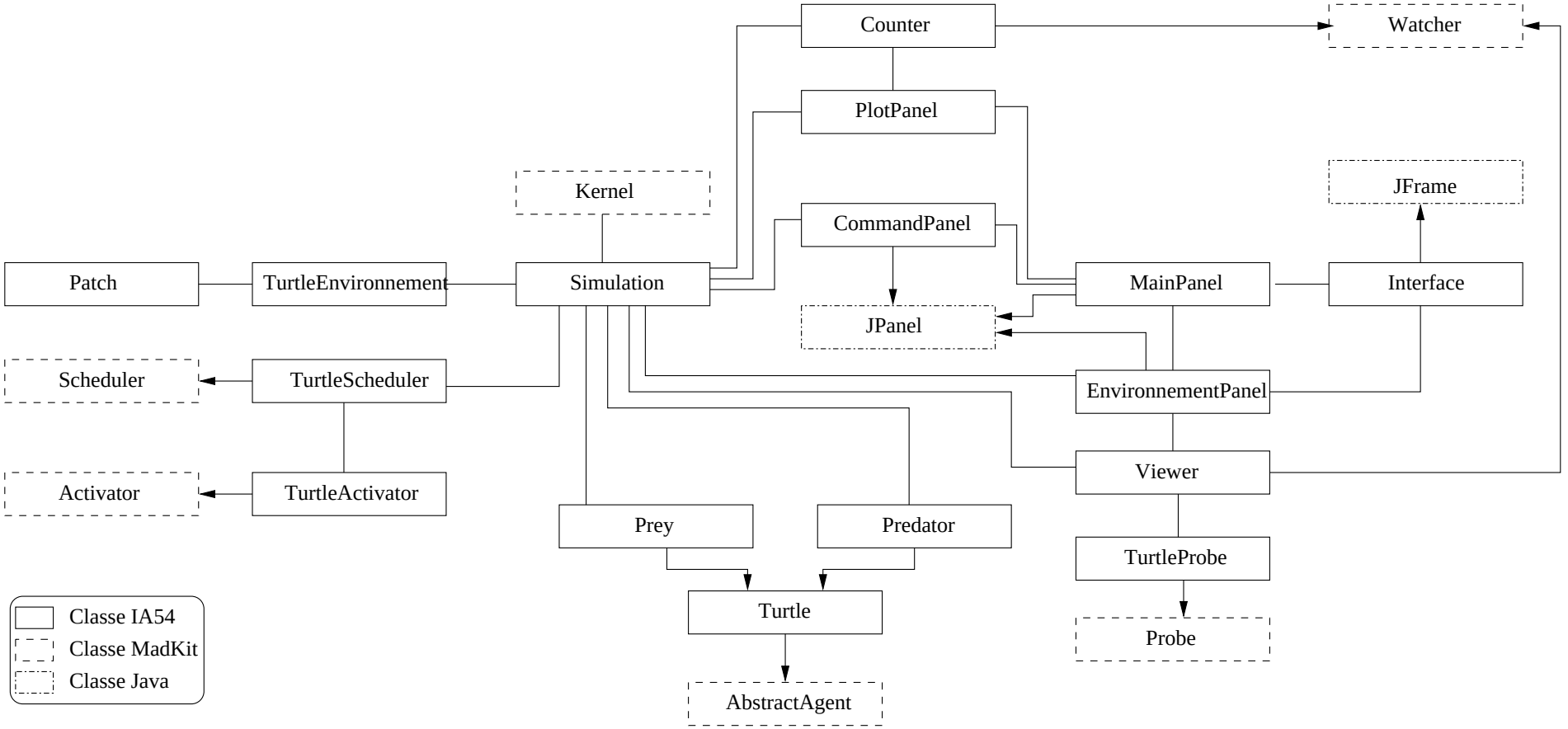

FIG. 3.1 – Diagramme UML

Chapitre 4

Interface graphique

4.1 Profil de l'utilisateur

Le logiciel réalisé est destiné à être utilisé par les personnes qui implémentent des stratégies de proies ou de prédateurs. L'utilisateur est donc un informaticien confirmé : l'interface doit donc être efficace, mais également proposer de nombreuses options pour tester correctement la simulation.

4.2 Fonctionnalités souhaitées

Pour le programmeur qui teste ses stratégies, il faut pouvoir créer une nouvelle simulation en donnant les paramètres statiques ne pouvant pas changer au cours de l'éxécution de la simulation. Ces paramètres sont la taille du monde (hauteur, largeur), le nombre de proies et de prédateurs initialement positionnés de manière aléatoire dans le monde et la taille des cellules de la représentation graphique.

Une fois la simulation créée, on veut pouvoir :

- visualiser la simulation : position de chaque proie et de chaque prédateur
- ajouter des proies et des prédateurs à l'aide de la souris dans l'environnement
- supprimer des proies et des prédateurs de l'environnement
- avoir des informations statistiques sur l'état de la simulation
- agir sur la simulation, par exemple en la mettant en pause, en changeant sa vitesse, ou en modifiant le comportement des proies et des prédateurs

Étant donné le profil utilisateur défini en section 4.1, l'interface doit fournir l'ensemble de ces informations en une seule fenêtre, de manière à ce qu'elles soient toutes disponibles en un clic souris. L'interface sera donc relativement chargée.

Il serait également souhaitable de pouvoir sauvegarder l'état d'une simulation pour le recharger plus tard, afin de réaliser plusieurs simulations identiques avec des stratégies différentes.

4.3 Démarche

Au début du projet, nous étions partis de l'interface fournie par défaut par TurtleKit. Nous l'avions remodelé afin qu'elle réponde aux besoins exposés ci dessus. Toutefois, elle ne répondait pas exactement aux attentes : le desktop MadKit prenait en charge une partie de l'affichage, et ne permettait pas de personnaliser l'interface comme nous le souhaitions. Notamment, au lieu de disposer d'une seule fenêtre uniforme et cohérente, il y avait une fenêtre pour la gestion de la simulation, une fenêtre pour la visualisation de l'environnement et une autre pour les statistiques.

L'interface ne répondait donc pas à nos attentes, c'est la raison pour laquelle nous avons décidé de construire notre propre interface, en dehors du desktop MadKit. Nous avions donc toute lattitude pour sa réalisation. Afin de réaliser une interface cohérente et lisible, nous avons commencé par reprendre les

besoins de l'utilisateur, puis nous avons réalisé sur papier une maquette de l'interface. Cette maquette a guidé la suite du développement.

4.4 Solution implémentée

L'interface, de même que l'ensemble du projet, est réalisée en langage Java. Elle utilise les composants de l'API *Swing* (voir [3]). L'interface a été réalisée sans environnement de développement intégré, en utilisant uniquement les classes standard de *Swing*.

4.4.1 Interface principale et menus

La figure 4.1 présente l'interface lors du lancement du logiciel, lorsqu'aucune simulation n'est lancée.

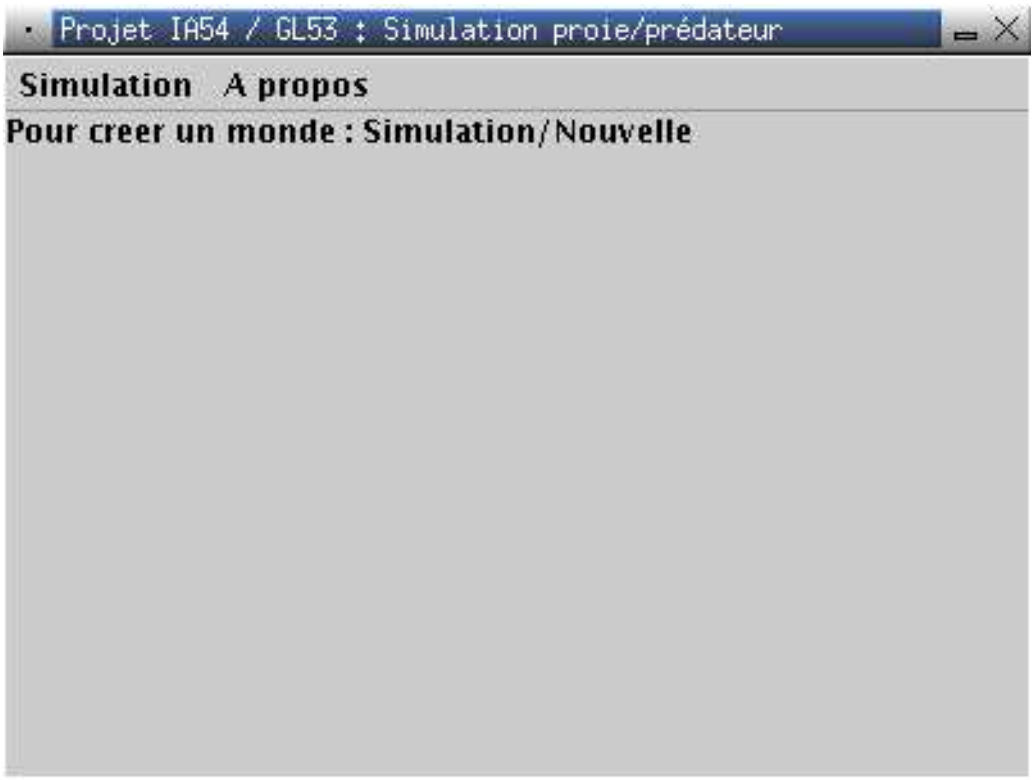

FIG. 4.1 – L'interface après son lancement

Notre interface dispose d'une barre de menu avec deux menus : *Simulation* (voire figure 4.2) et *A propos* (voir figure 4.3). Le menu *Simulation* permet de créer une nouvelle simulation, d'enregistrer la simulation, ou de charger une simulation précédemment sauvegardée, d'arrêter la simulation en cours et de quitter le logiciel. Le menu *A propos* donne des informations sur le logiciel. Le menu est implémenté au sein de la classe principale du programme, `Interface`.

FIG. 4.2 – Le menu *Simulation*

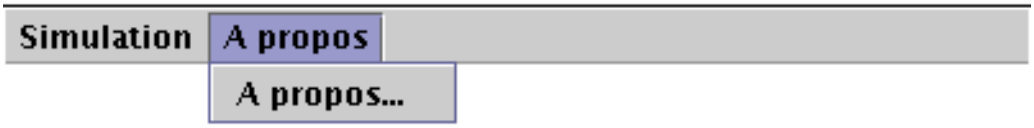

FIG. 4.3 – Le menu *A propos*

4.4.2 Lancement d'une nouvelle simulation

Pour lancer une nouvelle simulation, on utilise donc l'option *Nouvelle* du menu *Simulation*. Ceci a pour effet d'afficher la boîte de dialogue (voir figure 4.4) qui permettra de donner les paramètres statiques de la simulation, comme nous l'avons vu en section 4.2. Pour chaque paramètre numérique, la boîte de dialogue possède un label de description et un *spinner* de sélection de la valeur. L'option *Représentation imagée*, sous forme d'une boîte à cocher, permet de sélectionner si l'on souhaite une représentation graphique avec des images. Un texte explicatif est également présent en bas de la boîte de dialogue : il explique comment seront positionnées les proies et prédateurs, et précise qu'il sera possible d'en ajouter par la suite à l'aide de la souris.

Cette boîte de dialogue est simple et efficace : elle permet en un clin d'oeil de visualiser tous les paramètres disponibles et de les configurer.

Cette boîte de dialogue est implémentée au sein de la classe `NouveauMondeDialog`.

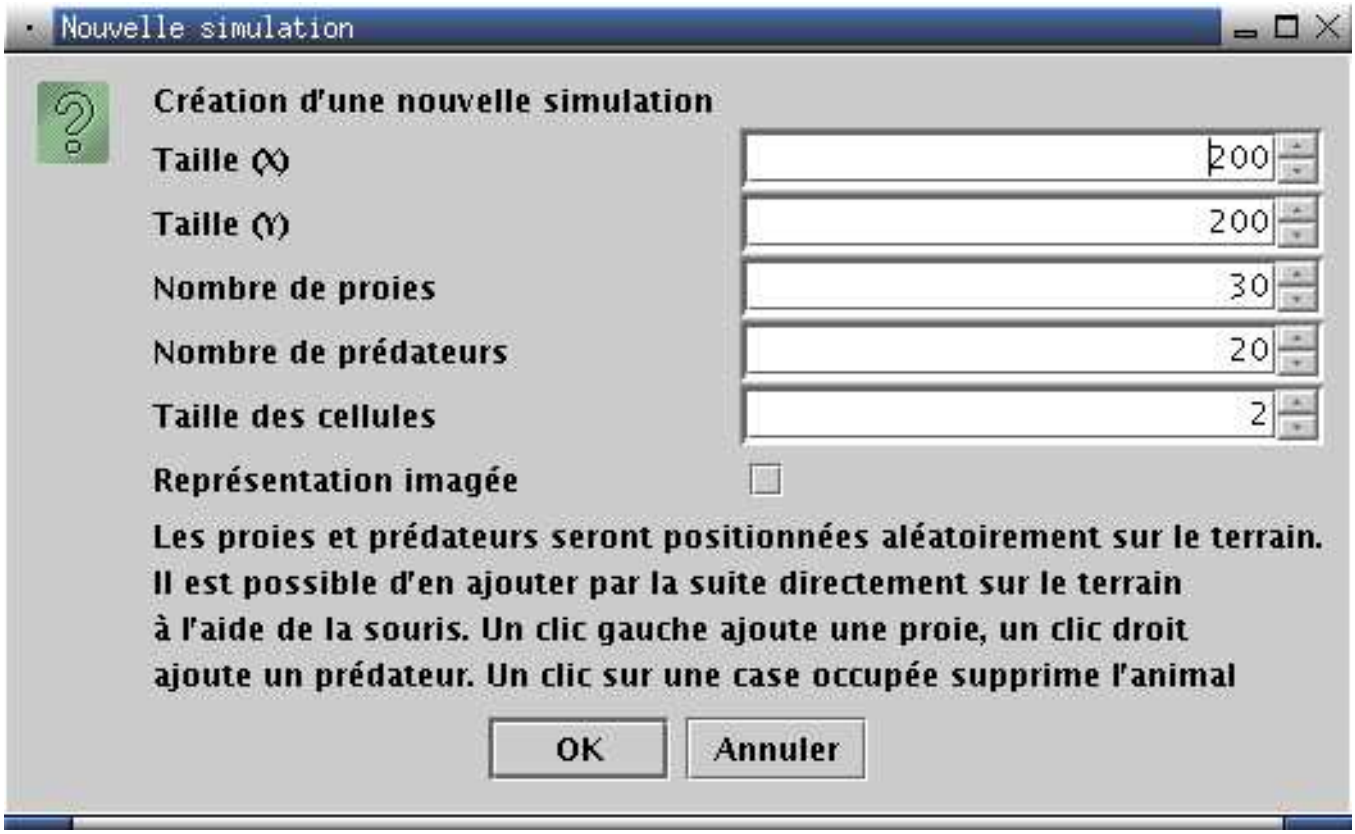

FIG. 4.4 – La boîte de dialogue de nouvelle simulation

A noter que la gestion des erreurs est prévue dans cette boîte de dialogue : si l'utilisateur entre une valeur trop importante pour le monde et que celle-ci dépasse la capacité mémoire de la machine, un message d'erreur explicite est affichée sous forme de boîte de dialogue :

FIG. 4.5 – Traitement des erreurs

4.4.3 Simulation en cours

Une fois la simulation créée, l'interface principale du programme s'affiche. La fenêtre se redimensionne automatiquement pour laisser place à l'environnement de simulation :

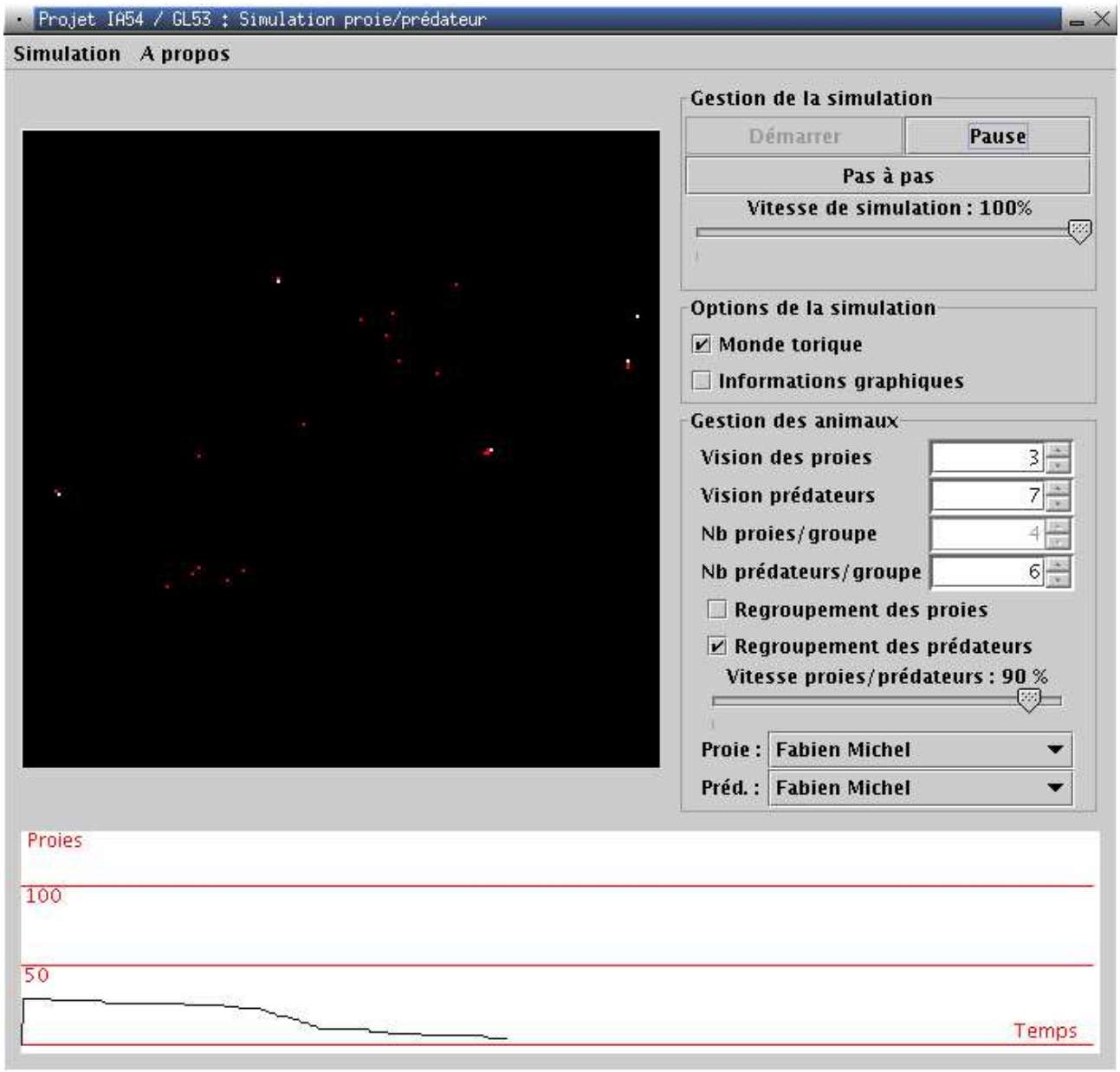

FIG. 4.6 – Simulation en cours

L'interface se décompose en 3 parties : le panneau de visualisation, le panneau de contrôle et le panneau de statistiques.

Le panneau de visualisation

Le panneau de visualisation permet d'observer la position de chaque proie et de chaque prédateur dans l'environnement. Il est implémenté dans la classe `EnvironnementPanel`, qui hérite de `JPanel`. Les fonctions classiques de dessin de la classe `Graphics` sont utilisées.

Deux modes d'affichage sont proposés : la représentation de chaque animal par un simple carré de couleur (rouge pour les prédateurs, blanc pour les proies), ou la représentation par une petite image. Le second mode n'offre pas une très bonne qualité de visualisation, la représentation classique est préférable, c'est donc la valeur par défaut.

Une fonctionnalité intéressante de ce panneau est qu'il permet de rajouter ou de supprimer des

proies et des prédateurs de l'environnement. Avant le lancement de la simulation, un clic gauche sur une zone vide ajoute une proie, un clic droit ajoute un prédateur. Un clic sur un animal supprime cet animal. Ceci permet de tester certaines stratégies par rapport à des positions d'animaux précises.

Le panneau de contrôle

Le panneau de contrôle de la simulation est l'élément le plus complexe de l'interface graphique. Il est implémenté au sein de la classe `CommandPanel` qui hérite de `JPanel`. Tous les paramètres configurables via ce panneau sont effectifs en temps réel sur la simulation.

Il se décompose en 3 parties, séparées par des bordures pour plus de lisibilité :

- Un panneau *Gestion de la simulation* qui permet de contrôler l'état de la simulation. Il permet de lancer la simulation (bouton *Démarrer*), de la mettre en pause (bouton *Pause*) ou de l'éxécuter en pas à pas (bouton *Pas à Pas*). D'autre part, un *slider* permet de configurer la vitesse de la simulation.
- Un panneau *Options de la simulation* qui permet de modifier deux options de la simulation. La première option est la toricité du monde, c'est à dire si les animaux à gauche du monde peuvent passer à droite, et ainsi dans toutes les directions. La seconde option, *Informations graphiques* permet d'afficher des informations sur chaque proie et chaque prédateur, en particulier leur cercle de vision.
- Un panneau *Gestion des animaux* qui permet de configurer le comportement des animaux. Il est ainsi possible de régler la taille du cercle de vision des proies et des prédateurs, de dire si l'on souhaite activer le regroupement des proies ou des prédateurs en donnant le nombre d'animaux par groupe. On peut également y configurer la vitesse des proies par rapport à la vitesse des prédateurs, ainsi que la stratégie de chacun des types d'animaux.

Le panneau de contrôle utilise donc de nombreux éléments graphiques, des *spinner*, des cases à cocher, des *sliders*, des boutons, des listes déroulantes. Elle est donc complexe, mais permet d'accéder en un clic de souris à toutes les options de la simulation. En effet, c'est ce que souhaite l'utilisateur : dans ce type d'applications, l'efficacité prime sur l'esthétique.

Le panneau de statistiques

Le panneau de statistiques est un graphe statistique du nombre de proies vivantes en fonction du temps. Il permet de mesurer la qualité des stratégies adoptées. Il est implémenté dans la classe `PlotPanel`, qui hérite de `JPanel`, puis les fonctions classiques de dessin de la classe `Graphics`.

4.5 Chargement et sauvegarde d'une simulation

Afin de faciliter les comparaisons de stratégies, nous avons implémenté une fonctionnalité permettant de sauvegarder l'état du monde dans un fichier XML puis de le recharger ultérieurement. Nous utilisons l'API DOM de Java pour lire et écrire le document XML.

D'un point de vue graphique, nous utilisons la classe Java `JFileChooser` pour sélectionner le fichier à lire ou à écrire.

4.6 Difficultés rencontrées

Java dispose d'un modèle objet clair qui rend la création d'interfaces graphiques et la création de nouveaux éléments très simple. Ainsi, créer les différentes boîtes de dialogue, créer le panneau de représentation de l'environnement et le panneau de statistiques n'a pas posé de problèmes.

En revanche, l'aspect problématique a été le positionnement des éléments graphiques. Les différents gestionnaires de positionnement (*Layout managers*) sont complexes à utiliser, et nous avons éprouvé des difficultés à obtenir la disposition que nous avions défini dans la maquette.

Conclusion

Du point de vue de l'UV IA54, ce projet nous a permis d'approfondir nos connaissances dans le fonctionnement de MadKit. Ainsi, nous avons pu étudier le fonctionnement des *scheduler* et des *probes*. La réalisation de la stratégie des agents a permis de réfléchir dans un cadre très simple à différentes solutions pour faire progresser nos prédateurs. Toutefois, l'absence de communication a limité les possibilités de stratégie, et nous n'avons pu obtenir de stratégie réellement satisfaisante bien que plusieurs voies aient été explorées. L'ajout de communication ou d'apprentissage par renforcement permettrait sans doute d'ouvrir de nouvelles perspectives stratégiques.

Du point de vue de l'UV GL53, ce projet nous a permis de concevoir une interface à partir des besoins de l'utilisateur et à travers la réalisation d'une maquette. Cette démarche nouvelle que nous avons pu expérimenter nous a permis de concevoir une interface simple, claire et efficace par rapport à l'utilisation prévue.

Bibliographie

[1] A simple solution to pursuit games, Richard E Korf, University of California, Los Angeles, Février 1992

[2] *Java 2 Platform, Standard Edition, v 1.4.1, API Specification*, Sun Microsystems, http://java.sun.com/j2se/1.4.1/docs/api/

[3] *Java Tutorial*, http://java.sun.com/docs/books/tutorial/index.html

[4] *MadKit Development Guide*, Jacques Ferber, Olivier Gutknecht, Fabien Michel, http://www.madkit.org/madkit/doc/devguide/devguide.html

[5] *MadKit API Documentation*, Madkit Development Team, http://www.madkit.org/madkit/doc/api/index.html

[6] *An introduction to TurtleKit*, Fabien Michel, http://www.madkit.org/madkit/doc/agents/turtlekit/t1.html