UNIVERSITE BLAISE PASCAL
UFR SCIENCES EXACTES ET NATURELLES
DEPARTEMENT DE MATHEMATIQUES
ET INFORMATIQUE
63177 AUBIERE CEDEX

Maîtrise d'Informatique

RAPPORT DE STAGE

Présenté par

Nicolas DUMOULIN

Sujet : Modélisation d'une colonie de chauve-souris vampires en vue de réaliser une application ludo-éducative.

Lieu du stage : Cemagref, groupement de Clermont-Ferrand
Unité de Recherche LISC
24, Avenue des Landais – BP 50085
63172 Aubière Cedex

Durée du stage : 5 mois

Remerciements

Je tiens tout d'abord à remercier **Frédéric Amblard**, mon maître de stage, pour ses conseils pertinents et sa confiance, et pour m'avoir motivé (sans le vouloir ;-) à poursuivre mes études vers un DEA.

Merci de manière générale à toute l'équipe du **LISC** pour sa bonne humeur permanente, et pour leur chaleureux accueil.

Je tiens également à remercier les membres de la « javalist » de l'Université de Strasbourg (**www.sogid.com/javalist/**) et ceux du Forum Hardware.fr (**forum.hardware.fr**), pour leur précieuse aide en langage Java.

I. Sommaire

II. Résumé :

Le but de cette étude est de modéliser une colonie de chauve-souris vampires dans le but de déterminer par la simulation les caractéristiques idéales pour réaliser un jeu, ce jeu étant à destination d'une exposition au Muséum National d'Histoire Naturelle et devant mettre en évidence les enjeux de la coopération chez les vampires. Le but de ce calibrage est de mettre le joueur dans des situations proches de la réalité, mais de faire apparaître régulièrement les événements exceptionnels qui génèrent un besoin de coopération.

III. Abstract :

This study aims at modelling a vampire bat's colony, in order to find by simulations the ideal characteristics of the corresponding game. This game is dedicated to an exposition at the « Muséum National d'Histoire Naturelles » in France, and must highlight the cooperation's stakes by vampire bats. This calibration must on the one hand put the player in situations near of the real cases, and on the other hand, it must also force the appearance of exceptional events, which generate a need for cooperation.

IV. Introduction

Ce rapport a pour objectif de rendre état de la première partie de mon stage de Maîtrise d'Informatique, qui s'est déroulé sur le site clermontois du CEMAGREF (Centre national d'Etudes sur le Machinisme Agricole, le Génie Rural, les Eaux et Forêts), au sein du Laboratoire d'Ingénierie pour les Systèmes Complexes (LISC). Ce stage est exceptionnellement prolongé, et durera 5 mois au lieu de 3 mois.

Le but de ce stage est de concevoir une application ludo-éducative permettant d'illustrer l'aspect primordial de la coopération dans les sociétés de vampires (chauves-souris). Cette application sera ensuite utilisée par le Muséum d'Histoire Naturelle de Paris en vue d'une exposition sur la coopération chez les animaux. La stratégie utilisée ici, est de développer dans un premier temps une plate-forme de simulation qui permettra de caler le jeu, afin de faire apparaître des cas intéressants.

V. Présentation de l'entreprise

1. Le CEMAGREF

Institut de recherche pour l'ingénierie de l'agriculture et de l'environnement, le Centre national d'Etudes sur le Machinisme Agricole, le Génie Rural, les Eaux et Forêts, le Cemagref, est un établissement public à caractère scientifique et technologique (EPST). Il a été créé en 1981 et est placé sous la double tutelle du ministère de l'Education nationale, de l'Enseignement Supérieur, de la Recherche et de la Technologie et du ministère de l'Agriculture, de la Pêche et de l'Alimentation.

Outre la contribution au progrès des connaissances, le CEMAGREF diffuse des méthodes de diagnostic et de contrôle, il développe des outils de gestion et de négociation, il conçoit des technologies innovantes (exemple : la conduite automatique de tracteurs, détection du potentiel d'inondation d'un cours d'eau) et apporte son expertise aux services publics et aux entreprises. Les méthodologies scientifiques constituent un apport essentiel à la recherche en ingénierie. Modélisation, information géographique et informatique scientifique sont privilégiées.

Ces recherches contribuent au développement durable des territoires. Elles aident à protéger et gérer les hydrosystèmes et les milieux terrestres, à dynamiser les activités qui les valorisent et à prévenir les risques qui leur sont associés.

Ses objets d'étude sont donc le plus souvent des systèmes complexes, en relation avec des questions de société et sa démarche est presque toujours interdisciplinaire.

Les activités du CEMAGREF

Ses orientations scientifiques sont regroupées en 4 départements et concernent :

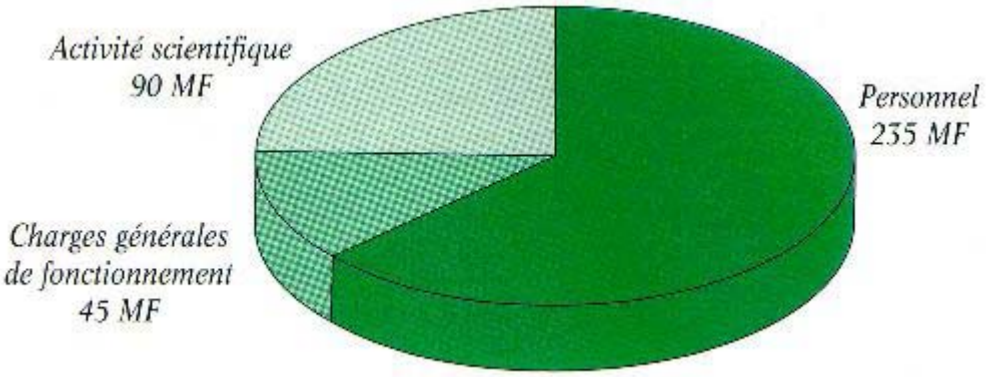
- le fonctionnement des hydrosystèmes,
- le génie des équipements et services pour l'eau et les déchets,
- la gestion des territoires à dominante rurale,
- le génie des équipements dans le secteur agricole et alimentaire.

Les hydrosystèmes et leurs relations avec l'agriculture d'une part, la gestion des risques naturels d'autre part, sont deux thématiques scientifiques prioritaires pour les années à venir. Les méthodologies scientifiques constituent un apport essentiel à la recherche en ingénierie.

1.1. Le budget

Le budget consolidé s'élève à 56,5 M€ (370 MF) dont plus de 15 M€ (100MF) de recettes de résultats. L'activité scientifique est donc négociée avec les nombreux partenaires qui assurent l'essentiel des moyens de fonctionnement des unités de recherche. Chaque année, plus de 150

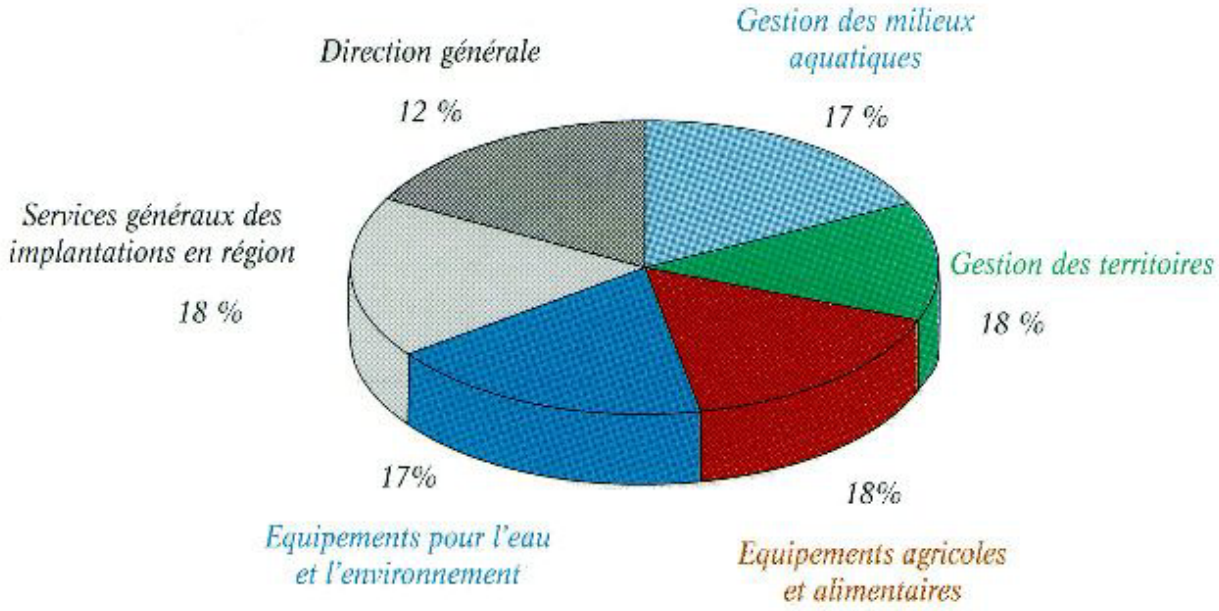
contrats de recherche et de transferts sont signés.



Le budget

1.2. Les ressources humaines

Le CEMAGREF emploie un peu moins de mille personnes à titre permanent. Deux tiers des emplois sont financés par le budget civil de la recherche et du développement. Le dernier tiers est mis à disposition principalement par le ministère de l'agriculture, de la pêche et de l'alimentation.



Les effectifs

2. Le CEMAGREF en région Auvergne-Limousin

Le CEMAGREF de Clermont-Ferrand est une des 9 délégations régionales de l'Institut, celle de la Région Auvergne-Limousin. Au même titre que le CNRS, l'INRA ou l'INSERM, il fait partie, en tant qu'EPST, du paysage de la recherche publique en Auvergne et s'applique à développer ses partenariats scientifiques, d'enseignement supérieur et formation ou de transfert dans cette région.

2.1. Trois Unités de Recherche et d'Expertise

Les activités de recherche du CEMAGREF en région Auvergne sont conduites sur 2 sites : Aubière et Montoldre. Le CEMAGREF de Clermont-Ferrand (Aubière) accueille 4 Unités de Recherche et d'Expertise (URE) :

- DFCF : Dynamiques et Fonctions des espaces ruraux
- TSCF : Technologies, Systèmes d'information et procédés pour l'agriculture et l'agro-alimentaire
- LISC : Laboratoire d'Ingénierie des Systèmes Complexes

3. Le Laboratoire d'ingénierie des systèmes complexes (LISC)

L'unité de recherche participe à des projets scientifiques en collaboration avec les quatre départements scientifiques du Cemagref. Elle a pour mission d'évaluer et de développer des méthodes de modélisation de systèmes complexes. Ce travail est mené en collaboration étroite avec les équipes thématiques de l'établissement et des partenaires extérieurs au travers de projets communs, souvent interdisciplinaires.

Le thème de recherche de l'unité, MAGESTYC (Modèles pour l'Aide à la GESTion des sYstèmes Complexes), vise à élaborer, tester, valider des méthodes de modélisation génériques utilisables dans les domaines d'activité du Cemagref : écosystèmes, dynamiques sociales et interactions avec le milieu, interactions entre équipements et milieux naturels. Le travail de recherche se concentre particulièrement sur l'utilisation de modèles géométriques dynamiques, qui permettent de définir l'évolution locale de formes, à diverses échelles. Ces modèles sont élaborés dans une démarche incrémentale, en étroite liaison avec des experts du terrain et des domaines abordés. Leur confrontation aux historiques de données disponibles et à l'appréciation des acteurs est une question particulièrement sensible. Les applications principalement visées sont des outils de gestion de la connaissance, généralement destinés aux experts d'un domaine, ou directement des outils d'aide à la décision ou à la négociation pour les acteurs de terrain.

Dans ce cadre, le LISC s'efforce de répondre aux questions suivantes lorsqu'elles se rapportent aux domaines thématiques des équipements agro-alimentaires, des aspects "qualité alimentaire", de l'eau, de la forêt et de la gestion de l'espace rural :

- Recueillir l'information pertinente pour un problème à partir de bases de données (statistiques, data mining)
- Mettre en place et valider des simulations de systèmes constitués d'agents en interaction, et rechercher des méthodes de résolution de problèmes distribués (systèmes multi-agents)
- Intégrer des connaissances expertes après leur recueil et leur formalisation (systèmes à base de connaissances)
- Intégrer des systèmes d'information cohérents, utilisables, et ouverts, raisonnés par rapport à une perspective d'aide à la décision ou de négociation

VI. Présentation de l'étude

1. Le sujet

Le Muséum d'Histoire Naturelle de Paris prépare une exposition pour 2003 portant sur la coopération dans le domaine animal. Une partie de l'exposition est consacrée à l'altruisme réciproque, et particulièrement au cas observé dans les sociétés de vampires (chauve-souris). En effet, les vampires se nourrissent de sang chaque soir, mais il arrive qu'ils rentrent bredouilles de leur chasse. Or, s'ils restent deux soirs sans réussir à se nourrir de sang, ils meurent d'inanition. C'est pour cela, que les vampires s'entraident en régurgitant du sang à ceux qui en ont besoin. Ce phénomène de coopération est décrit théoriquement par le dilemme du prisonnier itéré en théorie des jeux (Von Newman et Morgenstern, 1947). Le Muséum désire pour cette partie, disposer d'une application ludo-éducative installée sur des bornes interactives, afin que les visiteurs puissent apprécier les enjeux de la coopération chez les animaux. Le LISC, laboratoire situé au Cemagref à Clermont-Ferrand, a donc décidé de répondre à cette demande en en proposant la réalisation à un stagiaire.

Le but du stage est donc de réaliser l'application demandée sur une période de 5 mois (de Avril à fin Août). Cette application comportera deux objectifs. Tout d'abord, elle devra permettre de simuler l'évolution d'une colonie de vampires en fonction de paramètres spécifiques. Cette simulation permettra alors déterminer les paramètres saillants pour la réalisation du jeu. La seconde partie sera de réaliser le jeu lui-même, permettant d'illustrer aux joueurs l'importance de la coopération.

La plate-forme de simulation devra être conviviale. Elle proposera différentes boîtes de dialogue, pour ajuster les paramètres de la simulation, et éditer les propriétés des vampires et de la colonie. Elle proposera également un graphe permettant de représenter les échanges faits entre les vampires en temps réel.

La plate-forme de simulation et le jeu devront être réalisé en Java, afin que l'application soit entièrement portable notamment sur les bornes et qu'elle laisse la possibilité de proposer le jeu sur une page web. L'application à réaliser comportera un habillage graphique sommaire et permettra aux commanditaires de modifier l'interface graphique de manière simplifiée.

Jusqu'à maintenant, rien n'avait été encore développé dans ce cadre. Il n'y a donc pas d'existant. Rien n'était imposé quant aux logiciels à utiliser pour la conception et le développement.

2. Planning du stage

Premier mois	• Bibliographie sur l'espèce étudiée (le vampire *desmodus rotundus*) • Recherche d'outils technologiques de modélisation et de développement • Documentation sur les systèmes multi-agents. • Analyse et spécification du modèle à exploiter. • Implémentation du modèle.
Deuxième mois	• Poursuite de l'implémentation du modèle. • Développement de l'interface graphique. • Développement d'un mode de visualisation des relations entre les vampires.
Troisième mois	• Exploration des simulations, pour trouver les espaces de paramètres intéressants pour le jeu. • Poursuite du développement de l'interface graphique. • Rédaction du rapport de stage.
Quatrième mois	• Poursuite de l'exploration des simulations. • Spécification du scénario du jeu avec le Muséum d'Histoire Naturelle. • Conception du jeu.
Cinquième mois	• Développement du jeu. • Développement d'outils dédiés à la modification du contenu du jeu (graphisme et textes).

VII. Descriptif des outils acquis

Durant ce stage, j'ai surtout approfondi mes connaissances en modélisation UML, et en Java. J'ai manipuler des logiciels permettant de construire des modèles UML (Poseidon), et de développer en Java (JEdit). J'ai également découvert le domaine des systèmes multi-agents.

1. Le langage Java

Java est un langage orienté objet qui a été créé par la société Sun en 1995. Malgré des difficultés liées surtout au manque de performances (qui ont tendance à s'effacer), ce langage est aujourd'hui largement utilisé à travers le monde. Ses points forts sont :

- sa portabilité, qui est son principal atout. En effet, les programmes développés en Java sont ensuite exécutés par une machine virtuelle, qui sert d'interface entre l'application et le système d'exploitation. Ainsi, ces programmes peuvent être exécutés sur n'importe quelle plate-forme et sous n'importe quel système d'exploitation, pourvu qu'une machine virtuelle Java y soit installée.

- sa fiabilité. Le langage Java est fortement typé (plus que le C++), et n'offre pas la possibilité de manipuler des pointeurs, ce qui supprime bon nombre d'erreurs générées inévitablement dans d'autres langages.

- le grand nombre de classes fournies par le JDK (Java Development Kit). Parmi celles-ci, on trouve celles qui permettent de manipuler des flux (fichiers, socket, données provenant d'internet...), des bases de données, des structures de données, etc. Un ensemble de classes permet également de développer rapidement des interfaces graphiques très complètes.

- sa proximité syntaxique du langage C++. Celle-ci permet de passer de l'un à l'autre sans trop de mal.

Ses points faibles sont :

- La présence obligatoire de la machine virtuelle sur la machine utilisatrice.

- L'absence de gestion de la mémoire pour le programmeur, ce qui empêche toute optimisation, celle-ci étant complètement gérée par la machine virtuelle.

- L'impossibilité d'implémenter les héritages multiples, ceci en vue d'éviter des erreurs de programmation. Néanmoins Java offre une autre possibilité pour contourner ce problème qui est l'utilisation d'interfaces.

Dans le cadre de mon étude, le modèle ne présentant pas des proportions élevées, l'utilisation de la machine virtuelle n'a posé aucun problème. J'ai surtout apprécié ce langage pour sa facilité d'utilisation et la documentation exhaustive des classes fournies. La version du JDK que j'ai utilisée est la 1.4, la 1.3 est encore largement utilisée, mais la 1.4 apportait de nouvelles fonctionnalités fortes intéressantes comme l'amélioration de la gestion des composants Swing (interface graphique), une classe dédiée aux entrées sorties d'images (imageio), la prise en charge de la molette des souris, et bon nombre de nouvelles méthodes dans diverses classes.

2. Les logiciels utilisés

N'ayant pas de logiciels imposés pour la conception et le développement, j'ai donc choisi de faire mes propres recherches et trouver les outils qui seraient les mieux adaptés pour mon contexte.

Poseidon (www.gentleware.com) est un logiciel en shareware qui permet de construire des modèles UML, et de générer le code Java correspondant au modèle (classes, attributs, méthodes, accesseurs/modificateurs d'attributs, commentaires JavaDoc). Il possède également la capacité de faire du "reverse engineering", c'est-à-dire qu'à partir d'un ensemble de fichiers Java existants, il reconstruit un diagramme de classes UML correspondant.

JEdit (www.jedit.org) est un Environnement de Développement Intégré (EDI) en freeware. Il permet de faciliter le développement de gros projets en offrant une saisie semi-automatique du code, et des scripts de compilation générés automatiquement. Ce logiciel permet également l'intégration de modules supplémentaires qui existent en très grand nombre. Parmi ceux-ci, on pourra trouver un explorateur de classe qui permet de naviguer plus aisément dans le code, ou encore un enjoliveur de code, qui automatise l'indentation selon les préférences définies par l'utilisateur.

3. Les systèmes multi-agents

3.1. Définitions

Un agent est une entité qui peut être vue comme percevant et agissant de façon autonome sur son environnement. En voici une définition (Ferber, 1995) :

> *On appelle agent une entité physique ou virtuelle*
>
> *a. qui peut être capable d'agir dans un environnement,*
>
> *b. qui peut communiquer directement avec d'autres agents,*
>
> *c. qui est mue par un ensemble de tendances (sous la forme d'objectifs individuels ou d'une fonction de satisfaction, voire de survie, qu'elle cherche à optimiser),*

d. *qui possède des ressources propres,*

e. *qui est capable de percevoir (mais de manière limitée) son environnement,*

f. *qui ne dispose que d'une représentation partielle de cet environnement (et éventuellement aucune),*

g. *qui possède des compétences et offre des services,*

h. *qui peut éventuellement se reproduire,*

i. *dont le comportement tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont elle dispose, et en fonction de sa perception, de ses représentations et des communications qu'elle reçoit.*

Un système multi-agents est donc un système composé d'agents, de ressources et d'opérateurs qui ont pour rôle de gérer la coordination des tâches de chaque agent et l'influence que chacun des agents peut avoir sur l'environnement. Les agents interagissant également entre eux, ils vont créer un réseau de relations. Ces relations peuvent être des relations d'accointances (reconnaissances d'agent) ou des relations de communicabilité.

La dualité agent/environnement est au cœur des systèmes multi-agents, puisque l'environnement va structurer le comportement des agents (contraintes), et les agents vont également modifier leur environnement.

3.2. Modélisation par systèmes multi-agents

Le besoin d'utiliser les systèmes multi-agents comme support de simulation vient des limites des techniques de simulation traditionnelles (modèles déterministes et stochastiques). Les méthodes de simulation classiques représentent les individus de manière agrégée, tandis que la simulation par système multi-agents considère chaque individu, ce qui apporte plus de cohérence dans le modèle et une plus grande souplesse et modularité des individus.

Pour (Bousquet, 1994), élaborer un système multi-agents revient à reproduire un monde artificiel ressemblant au monde observé, en ce sens qu'il est composé de différents acteurs, de façon à y mener des expériences diverses et variées.

A travers cette approche, il est possible de représenter directement les individus d'un système, leurs comportements et leurs interactions. C'est-à-dire, qu'au lieu de modéliser les systèmes en s'appuyant sur des équations différentielles, sur des matrices de transitions ou toute autre représentation formelle, une structure de donnée est proposée pour chaque entité existante dans le système (Campos, 2000).

VIII. L'étude

1. Le vampire desmodus rotundus

Gerald Wilkinson de l'Université de Maryland (USA), accompagné de Robin Weiss, Michael Jones et Terri Lamp a passé 26 mois (entre 1978 et 1983) au Costa Rica pour étudier le comportement des vampires *desmodus rotundus* (Wilkinson, 1990).

1.1. Présentation générale de l'espèce

Le sujet étudié est le vampire commun (*desmodus rotundus*), qui vit en Amérique Centrale et se nourrit exclusivement de sang qu'il se procure en incisant à l'aide de ses dents la peau des vaches et chevaux, et en léchant le sang qui sort ainsi de la plaie. Il pèse en moyenne entre 95 et 140 grammes, et son espérance de vie est comprise entre 14 et 20 ans.

Figure 1.1 *Un desmodus rotundus*

Pour rester en vie, le vampire doit consommer 2 à 30 millilitres de sang toutes les 60 heures, sinon il meure d'inanition. Il chasse une fois par jour (la nuit) en groupe. Ses chasses ne sont pas toujours fructueuses, et son taux de réussite varie en fonction de l'âge. Un adulte (à partir de 2 ans) a un taux de réussite à la chasse de 97%, tandis qu'un jeune vampire de 3 mois rentrera bredouille 3 fois sur 10. Ce taux de réussite croît donc de 70% à 97% entre 3 mois et 2 ans, qui correspond à l'apprentissage du jeune vampire.

1.2. Renouvellement de l'espèce

Les femelles sont en âge de procréer à partir de 10 mois. Leur période de gestation dure entre 6 et 8 mois. Les femelles ont une portée de un vampire par an. A la naissance les petits pèsent entre 5 et 7 grammes, et durant le premier mois de leur vie, ils seront nourris par le lait maternel. Au bout d'un mois, leur poids a doublé, il est passé à 12 grammes. De 1 mois à 3 mois, la mère

nourrit son petit de sang qu'elle régurgite. A partir de 3 mois, le petit commence à être autonome, il part lui-même à la chasse en quête de sang, mais conserve un lien privilégié avec sa mère. Ses chasses n'étant pas encore efficaces, il doit pouvoir demander du sang à sa mère. A 5 mois, il a terminé sa croissance, il a donc atteint un poids adulte. Il sera complètement sevré à 10 mois, lorsqu'il aura atteint sa maturité sexuelle.

1.3. Vie sociale

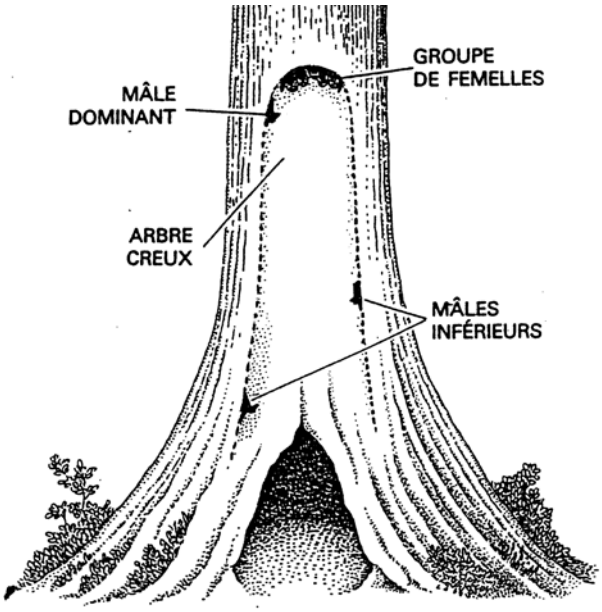


Figure 1.2 *Répartition des vampires dans la niche.* (extrait de *Pour la science* n°150 Avril 1990 p.62)

Les vampires vivent en colonies de 100 individus, et se répartissent en sous-groupes d'une quinzaine d'individus. Chaque sous-groupe loge dans une niche, leurs niches préférées étant les troncs d'arbres creux. Les femelles changent régulièrement de sous-groupe. Les mâles eux, défendent leur place dans la niche, et restent donc tout le temps dans la même. En général, les femelles sont regroupées avec leurs petits tout en haut de la niche, et les mâles eux se placent sur les parois un peu plus bas, et se battent pour être le plus haut possible. Ce comportement fait qu'une femelle voit plusieurs mâles.

Les vampires communiquent en émettant des sons très aigus. L'équipe de Gerald Wilkinson a montré que ces signaux étaient semblables d'un vampire à ses descendants, ce qui leur permet d'identifier un membre de leur lignée, et que les vampires étaient capables d'en identifier un autre grâce à ces sons, même après plusieurs années.

Lorsqu'il est rassasié, un vampire a une réserve de 60 heures de survie. Le lendemain d'une chasse infructueuse, il doit absolument se procurer du sang sinon il mourra avant la prochaine chasse. Au bout de deux jours sans réussite à la chasse, le vampire va « demander » à une femelle de lui régurgiter une partie du sang qu'elle possède dans son estomac, car seules les femelles acceptent de le faire. Celle-ci se réserve tout de même le pouvoir de refuser ou accepter selon plusieurs critères qui concerne les liens de parentés et les affinités qui se créent au fil du temps. Les régurgitations n'ont lieu que lorsque le vampire demandeur est réellement en danger de mort. Cela signifie également que les vampires sont capables de détecter si un autre individu est affamé ou pas.

Lors d'une régurgitation, le vampire perd moins que l'autre ne gagne. Pour mieux comprendre ce phénomène, observons la courbe de la figure 1.2. La courbe représente le nombre d'heures gagnées au cours de la digestion. On remarque que la courbe n'est pas linéaire, et que de ce fait, durant la première journée qui suit une chasse fructueuse le fait de céder un peu de sang ne fait perdre que trois heures de survie, tandis qu'au bout de deux jours sans approvisionnement une petite quantité de sang va alimenter beaucoup plus l'organisme du receveur.

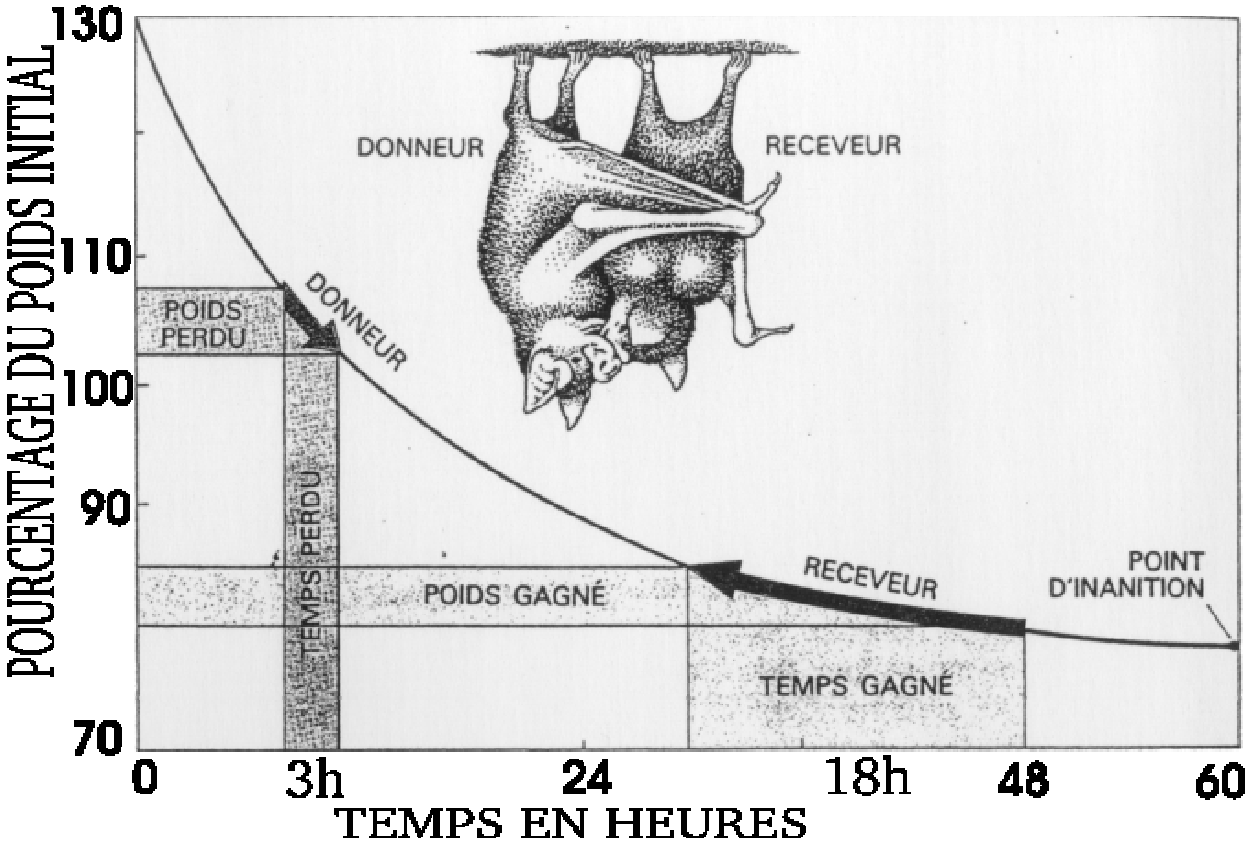


Figure 1.3 *Le vampire receveur est plus avantagé que le donneur n'est gêné.* (extrait de *Pour la science* n°150 Avril 1990 p.63)

2. Modélisation

2.1. Cadre théorique : le dilemme du prisonnier itéré

Le propos de la théorie des jeux est l'étude de toute situation présentant des caractéristiques semblables à celles des jeux de société, c'est-à-dire de situations où des individus font des choix en interaction, dans un cadre stipulé à l'avance.

Le « dilemme du prisonnier » est un type de jeu où des individus ont intérêt à s'entendre, plutôt qu'à ne pas s'entendre, mais où chacun peut gagner à ne pas respecter un éventuel accord, si les autres s'y tiennent. Tel est le cas lorsque deux suspects sont incités à se dénoncer l'un l'autre. Voici l'énoncé de ce jeu qui est exposé par (Axelrod, 1984) : P1 et P2 sont deux prisonniers qui

ont le choix entre coopérer et trahir, chacun d'entre eux ne connaissant pas la décision de l'autre.

Le tableau 2.1 présente les gains réalisés par les deux prisonniers en fonctions de leurs décisions. En observant les gains, on observe que, a priori, il est préférable de trahir car on gagne plus qu'en coopérant (5 ans de remise de peine ou 1 contre 3 ans ou 0), et ce quelque soit le choix de l'autre prisonnier. Hors si les deux prisonniers suivent ce raisonnement, il réalise un gain moins élevé que s'ils avaient coopéré tous les deux, d'où le dilemme.

		P2	
		Coopère	Trahit
P1	Coopère	(3,3)	(0,5)
	Trahit	(5,0)	(1,1)

Tableau 2.1 *Le dilemme du prisonnier*

On appellera stratégie l'ensemble des règles qui déterminent les choix faits par l'individus. Lorsque ce type jeu est itéré, le nombre de stratégie devient vite extrêmement grand (il croît exponentiellement avec le nombre de coups). Parmi ces stratégies, (Delahaye, 1992) en retient 12, dont :

- La stratégie "méchante" (toujours trahir).
- La stratégie "gentille" (toujours coopérer).
- La stratégie du "donnant-donnant", consistant à riposter par un trahison à toute trahison de l'autre.
- La stratégie "lunatique" consistant à faire son choix au hasard.

Dans le cas du dilemme du prisonnier itéré, où le nombre d'itération n'est pas connu à l'avance, la meilleure stratégie à adopter est le « donnant-donnant ».

Le contexte de la coopération chez les vampires illustre parfaitement le dilemme du prisonnier itéré. La stratégie adoptée par les vampires est le « donnant-donnant », qui leur permet d'éliminer des profiteurs, car les vampires favorisent ceux qui font partie de leurs bonnes relations.

2.2. Caractéristiques retenues pour notre modèle

Modéliser un écosystème est une tâche complexe d'une part, et d'autre part tous les aspects de la réalité ne nous intéresse pas dans le cadre de la réalisation du jeu.

Pour commencer, nous ne considérerons pas l'aspect spatial dans notre modèle, car ce sont l'état des relations de coopération entre les vampires dans un sous-groupe et l'évolution de la population qui nous intéressent. La répartition des vampires se fera donc juste au niveau de l'agrégat « sous-groupe ».

D'un point de vue démographique, on introduit dans le modèle une dynamique de population par un taux de natalité et un processus de mortalité, qui se compose de la mort naturelle et de la mort

par manque de nourriture (on ne prend pas en compte les accidents qui pourraient survenir lors des sorties nocturnes).

Les mâles adultes ne sont pas considérés, car ils ne participent pas aux échanges de sang. De cette manière, nous les prenons en compte pendant leur phase de sevrage, car à ce moment, ils participent aux régurgitations mère - petit. Notre modèle ne comportera donc les mâles qu'entre le moment de leur naissance et 10 mois.

Etant donné que durant les deux premiers mois, le petit vampire est complètement dépendant de sa mère et qu'il ne chasse pas encore, les petits seront inclus dans le modèle à partir de 2 mois. De plus, comme la gestation n'influence a priori pas le comportement de la mère, elle ne sera pas prise en compte. Donc tous les ans, une mère fera naître un petit de 2 mois.

2.3. L'approche multi-agents

Dans notre cas, il apparaît assez clairement que l'individu vampire est un agent de notre système. Il nous reste à définir son comportement, et également un ordonnanceur qui va déclencher les événements au niveau de la colonie et des sous-groupes.

L'ordonnancement se fera à plusieurs niveaux. Au niveau de la colonie seront déclenchées successivement les différentes phases d'une journée pour chaque sous-groupe. Et chaque sous-groupe déclenchera les étapes pour chaque vampire qu'il contient.

Chaque journée de la simulation se découpera effectivement en plusieurs phases :

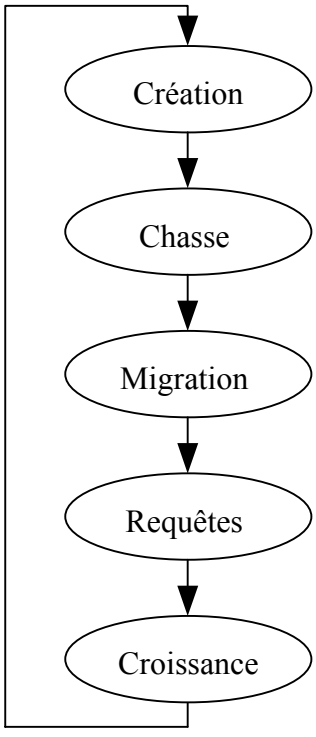


Figure 2.1 *Les phases d'une journée*

- Création : durant cette phase, les femelles vampires sont sollicitées pour créer de nouveaux vampires. Chaque femelle ayant ses propres caractéristiques, les naissances seront dispersées sur la durée de la simulation.
- Chasse : les vampires reçoivent leur ration de sang, en fonction d'un tirage aléatoire, qui dépend évidemment de leur taux de réussite.
- Migration : les femelles ont la possibilité de changer de sous-groupe, si elles n'ont pas de petits, et en fonction d'un facteur aléatoire.
- Requêtes : c'est à ce moment que les vampires affamés peuvent demander du secours auprès d'autres vampires.
- Croissance : tous les vampires vieillissent d'un jour, changent de classe d'âge si nécessaire ou meurent s'ils ont trop vieux, ou s'ils n'ont plus assez de réserves.

2.4. Le modèle de simulation

2.4.1. Modélisation du système

Notre système comportant une agrégation au niveau des formations de sous-groupes, nous allons retrouver également cette structure dans notre modèle. Nous allons avoir besoin d'une classe coordinatrice, qui contiendra tous les paramètres de la simulation et les méthodes qui permettront de les manipuler. Cette classe sera la classe `Simulateur`, que nous découvrons sur la figure suivante :

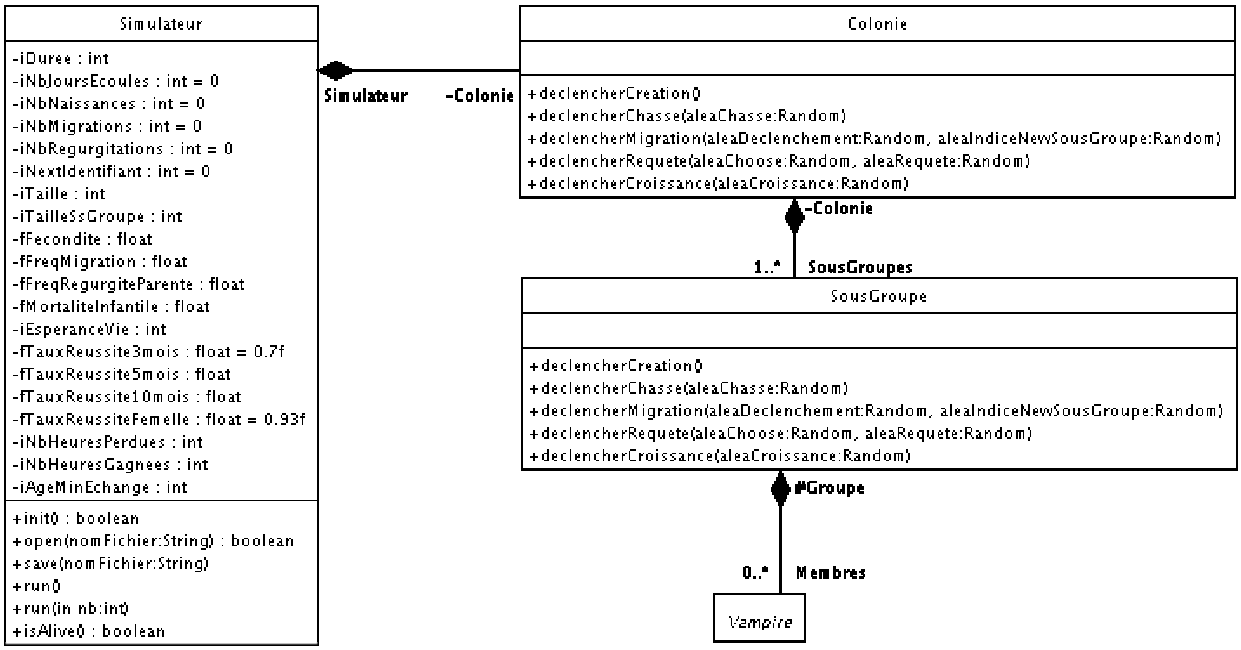


Figure 2.2 *Diagramme de classe UML : les différents niveaux d'agrégation*

Dans la classe `Simulateur`, nous retrouverons donc tous les paramètres pouvant initialiser une simulation et des indicateurs sur l'évolution de la colonie, qui sont :

- `iDuree` (le « i » signifie que la variable est un entier, selon la notation polonaise) : la durée en jours de la simulation.
- `iNbJoursEcoules` : le nombre de jours écoulés durant la (les) simulation(s).
- `iNbNaissances` : le nombre de naissances survenues.
- `iNbMigrations` : le nombre de migrations.
- `iNbRegurgitations` : le nombre de régurgitations faites entre vampires.
- `iNextIdentifiant` : identifiant pouvant être attribué à un nouveau vampire en respectant l'unicité. Ce nombre est incrémenté automatiquement lors d'une attribution d'un identifiant à

un vampire.

- `iTaille` : taille initiale de la colonie.
- `iTailleSsGroupe` : taille initiale moyenne des sous-groupes.
- `fFecondite` : proportion initiale de jeunes vampires par rapport à l'ensemble de la population.
- `fFreqRegurgiteParente` : proportion initiale des régurgitations mère - petit par rapport à l'ensemble des régurgitations.
- `fMortaliteInfantile` : probabilité pour un jeune vampire de moins de 10 mois de mourir.
- `iEsperanceVie` : Limite d'âge en jours pour un vampire.
- `fTauxReussite{3mois;5mois;10mois;VampireFemelle}` : taux de réussite à la chasse à l'âge correspondant. Entre deux paliers, ce taux de réussite suit une évolution linéaire et est recalculé tous les jours.
- `iNbHeuresPerdues` et `iNbHeuresGagnées` : nombres d'heures de survie perdues et gagnées lors d'une régurgitation.
- `iAgeMinEchange` : âge à partir duquel un vampire participe à des régurgitations autres que dans le cadre mère – petit.

Voici maintenant la description de ses méthodes :

- `init()` : qui comporte autant de paramètres que la classe possède d'attributs. Cette méthode va tout simplement initialiser une colonie de vampire en respectant les paramètres fournis. Le paramètre de retour sert à signaler si l'initialisation s'est déroulée correctement, ou si une incohérence dans les paramètres a été rencontrée (nombre de groupes nul, etc.).
- `open(String)` : permet d'initialiser une simulation à partir d'un fichier contenant l'état d'une simulation (conditions initiales et états des vampires).
- `save(String)` : permet de la même manière de sauvegarder l'état d'une simulation dans un fichier.
- `run()` : déclenche la simulation pendant autant de jours que précisé dans les conditions initiales.
- `run(int nb)` : déclenche la simulation sur `nb` jours. Cette fonction va notamment permettre de faire du pas-à-pas.
- `isAlive()` : renseigne si une simulation est en cours.

Le simulateur possède une colonie, qui elle-même possède plusieurs groupes qui sont composés de plusieurs vampires. La méthode `run()` de la classe `Simulateur` va appeler successivement les cinq méthodes de la classe `Colonie`, qui vont elles-mêmes appeler les méthodes correspondantes pour chacune des instances de la classe `SousGroupe`. Ces cinq méthodes correspondent aux cinq phases décrites plus haut (figure 2.1).

Lorsqu'un vampire meurt ou qu'un mâle est retiré de la simulation à 10 mois, on veut néanmoins pouvoir les garder en mémoire pour consulter leur état plus tard. C'est pour cela que deux associations ont été ajoutées entre la classe `Simulateur` et la classe `Vampire` comme on peut le voir sur la figure ci-dessous.

Figure 2.3 *Diagramme de classe UML : conservation des individus sortis du modèle*

2.4.2. Modélisation de l'agent vampire

Pour commencer, nous pouvons dégager quatre catégories d'âge pour nos vampires :

- Entre 3 et 5 mois, nous parlerons de « chasseur enfant ». Le vampire est alors en pleine croissance, et commence à chasser.
- Entre 5 et 10 mois, nous parlerons de « chasseur débutant ». Le vampire a terminé sa croissance, et améliore ses capacités de chasseur.
- Entre 10 mois et 2 ans, nous parlerons de « jeune femelle ». A cet âge, les mâles ne sont plus pris en compte, le vampire est sevré, mais il n'a pas encore les compétences d'un chasseur adulte.
- Plus de 2 ans, nous parlerons de « femelle ».

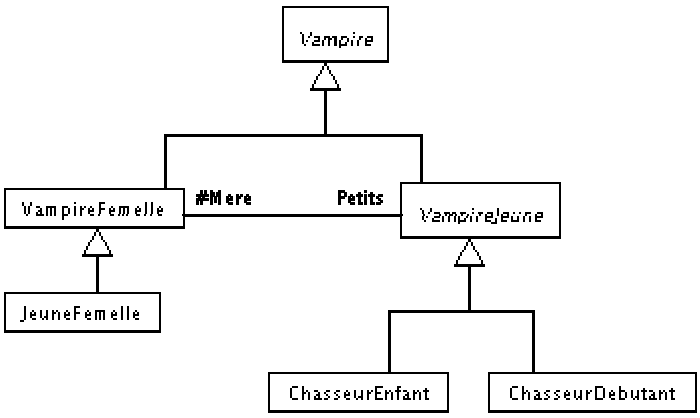


Figure 2.4 *Diagramme de classe UML : diagramme d'héritage des vampires*

`Vampire` et `VampireJeune` sont des classes abstraites, qui permettent de spécifier des propriétés et méthodes communes. Nous retrouvons bien nos quatre catégories de vampires : `ChasseurEnfant`, `ChasseurDebutant`, `JeuneFemelle` et `VampireFemelle`.

Approchons maintenant de plus près de la classe `Vampire` pour observer leurs propriétés et méthodes.

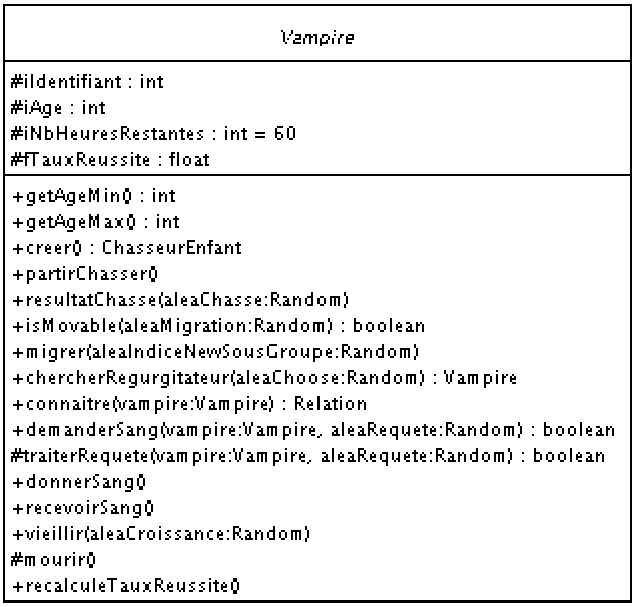


Figure 2.5 *La classe Vampire*

Un vampire a quatre propriétés :

- `iIdentifiant`: est l'identifiant unique du vampire, qui va permettre de suivre un individu.
- `iAge` : est l'âge du vampire
- `iNbHeuresRestantes` : est le nombre d'heures de vie restantes au vampire avant qu'il ne meure d'inanition.
- `fTauxReussite` : est son taux de réussite à la chasse.

Pour le nombre d'heures restantes, nous avions également la possibilité d'utiliser le contenu stomacal du vampire (qui s'exprimait alors en millilitres), mais dans ce cas, les quantités auraient différé d'un vampire à l'autre selon son âge. Tandis que cet indicateur a la même sémantique quelque soit l'âge du vampire.

Intéressons-nous ensuite à ses méthodes :

- `getAgeMin()` et `getAgeMax()` renseignent les bornes de la période d'existence du vampire, et serviront surtout à déterminer si un vampire change de catégorie d'âge.
- `creer()` va servir au renouvellement de l'espèce. Bien sûr, cette fonction ne sera effective que pour les vampires adultes.
- `partirChasser()` n'est pas implémentée pour l'instant, elle vise à reproduire graphiquement une chasse.
- `resultatChasse()` correspond au tirage aléatoire pondéré par le taux de réussite de chaque vampire qui va déterminer si le vampire à eu une chasse fructueuse ou non.

- `isMovable()` va déterminer si le vampire est apte à migrer vers une autre niche. On peut noter que les vampires qui ne sont pas des femelles de plus de 10 mois répondront toujours qu'ils ne sont pas aptes. La décision des femelles dépendra de la présence d'un petit à éduquer, et d'un facteur aléatoire qui est passé en paramètre de la fonction et dépend d'un paramètre initial propre à la simulation.

- `migrer()` fait migrer le vampire dans un nouveau groupe déterminé aléatoirement.

- `chercherRegurgitateur()` permet de trouver un vampire à qui demander du sang. Cette recherche se fait dans l'ordre qui suit : Le vampire cherche d'abord les vampires envers qui il a une dette, ensuite ceux avec qui il a déjà eu une relation, et enfin les autres femelles de sa nichée. Le choix d'un vampire dans l'une de ces catégories se fait aléatoirement.

- `connaitre()` renseigne si on a déjà eu affaire à un vampire dans le cadre de régurgitations.

- `demanderSang()` correspond à l'émission d'une requête à destination d'un donneur potentiel.

- `traiterRequete()` correspond au traitement d'une requête provenant d'un demandeur de sang. Retourne la réponse à la requête : acceptation ou refus.

- `donnerSang()` et `recevoirSang()` ne font qu'incrémenter et décrémenter le nombre d'heures du donneur et du receveur. La présence de ces fonctions vise aussi à reproduire graphiquement ces actions.

- `vieillir()` correspond à la dernière phase de la journée (voir 2.2 et figure 2.1).

- `mourir()` déplace le vampire de la `colonie` vers le cimetière (`VampiresMorts`, voir plus bas).

- `recalculeTauxReussite()` recalcule le taux de réussite en fonction des paramètres du modèle et de son âge.

La classe `VampireFemelle` possédera l'attribut supplémentaire `iNbJoursSansCreation` qui renseignera s'il est temps pour la femelle d'avoir une nouvelle progéniture, ou s'il est trop tôt.

Une femelle va également posséder une stratégie qui définira son comportement et sa prise de décision lors des demandes de régurgitation. Voici le diagramme :

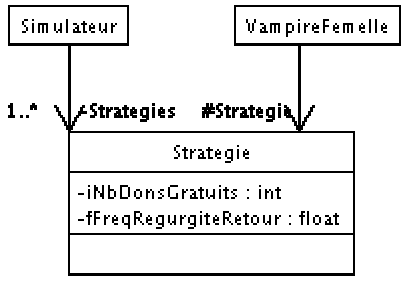


Figure 2.6 *Diagramme de classe UML : gestion des stratégies*

Les stratégies font partie des paramètres initiaux, on peut initialiser une simulation avec plusieurs stratégies et la méthode `init()` les attribuera aléatoirement aux femelles, ou bien on

peut initialiser une simulation avec une stratégie unique, et par la suite ajouter une stratégie uniquement pour un ou plusieurs individus. Un objet `Simulateur` a donc au moins une stratégie de définie et chaque femelle en possède une. Une stratégie comporte la limite de dons faits sans que l'autre vampire ne nous aide en retour, et une probabilité d'accepter finalement lorsque toutes les conditions sont réunies.

2.4.3. Gestion des relations

Nous allons maintenant nous intéresser à la manière dont sont entretenues les relations entre les vampires. Chaque relation contient une trace des événements survenue entre les deux vampires.

Il existe 4 événements que l'on représentera par des entiers, en voici les correspondances :

1. Le vampire a accepté de donner.
2. Le vampire a refusé de donner.
3. L'autre vampire de la relation a accepté de donner du sang au vampire.
4. Le vampire a subit un échec lors de sa requête.

L'événement 0 sera la représentation d'un non-événement.

Pour stocker cette trace, j'avais opté dans un premier temps pour un tableau ayant pour dimension la durée de la simulation, ainsi au i-ème élément du tableau correspondait l'événement du i-ème jour. Cette structure n'était pas du tout adaptée, et j'ai pu le vérifier en me heurtant à de multiples dépassements de capacité mémoire lors de simulation sur plusieurs années. En effet, le nombre d'événements entre deux vampires reste assez réduit. Je me suis donc ensuite tourné vers une table de hachage, pour qu'il y ait autant d'éléments que d'événements. L'événement du jour i reçoit tout simplement la clé de hachage i, on a ainsi une association jour/événement.

Voici le diagramme qui illustre tout ceci :

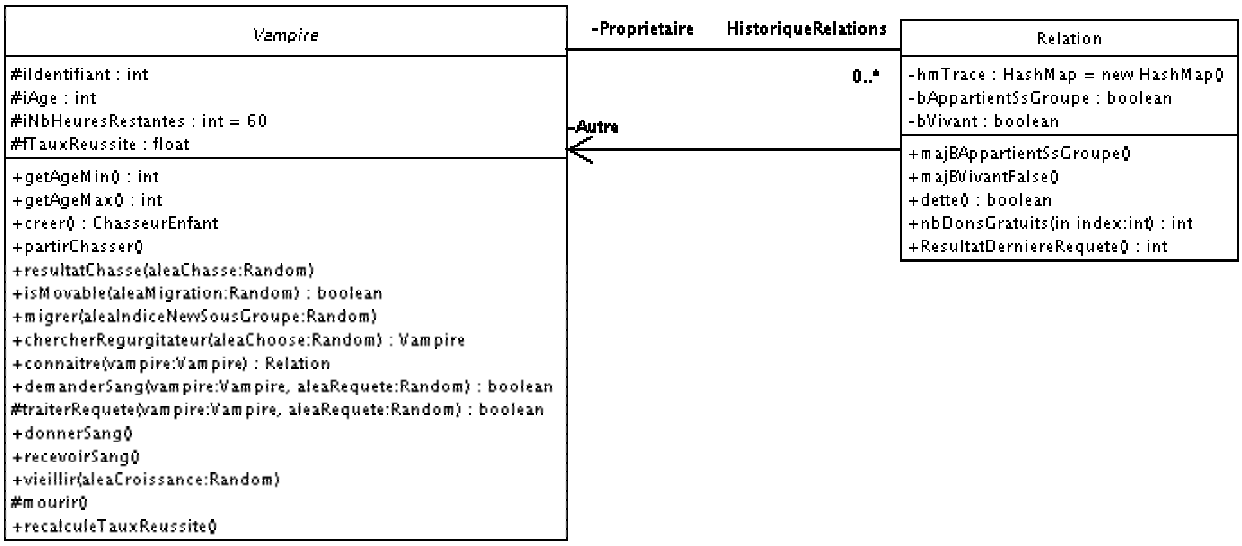


Figure 2.7 *Diagramme de classe UML : gestion des relations*

On retrouve ici les caractéristiques énoncées plus haut. Un vampire peut accéder à l'historique de ses relations. Depuis une relation, on peut atteindre son propriétaire et l'autre vampire concerné. Les attributs `bAppartientSsGroupe` et `bVivant` renseignent si l'autre vampire de la relation est dans le même sous-groupe, et s'il est toujours vivant. Voici ensuite quelques explications sur les méthodes :

- `majBAppartientSsGroupe()` et `majBVivant()` permettent de mettre à jour les attributs correspondants lorsque cela est nécessaire.
- `dette()` renvoie si le vampire a une dette envers l'autre vampire de la relation. C'est-à-dire si le dernier échange venait de l'autre vampire.
- `nbDonsGratuits(int index)` renvoie le nombre de dons qu'a fait le vampire à l'autre sans qu'il l'aide en retour depuis le jour `index`.
- `ResultatDernierRequete()` nous renseigne sur le résultat de la dernière requête fructueuse, c'est-à-dire, quel est le dernier vampire a avoir aidé l'autre.

2.4.4. Synthèse

Au final on obtient le diagramme suivant pour ce qui est du modèle de simulation d'une colonie de vampire :

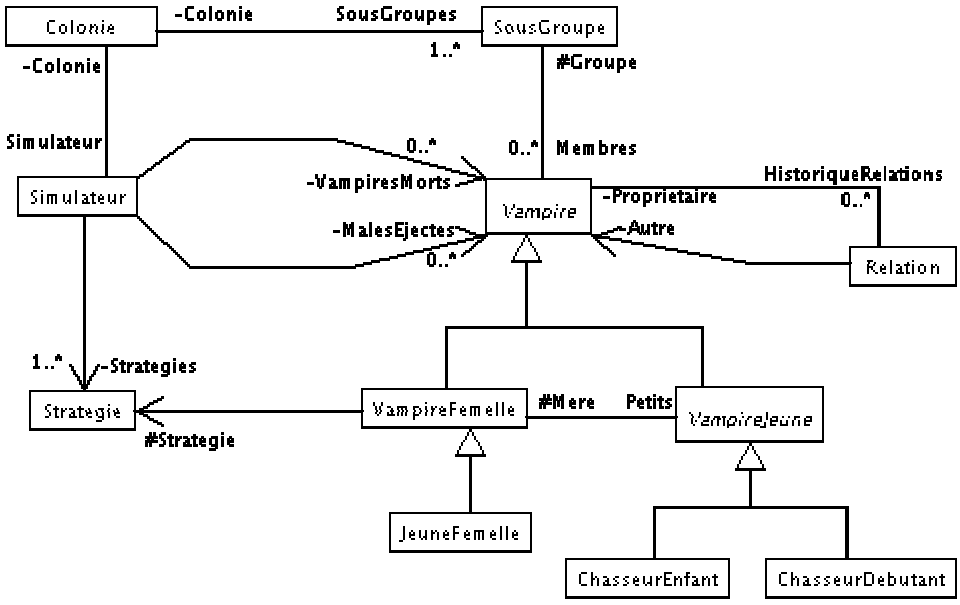


Figure 2.8 *Diagramme de classe UML : le modèle de simulation*

2.5. Représentation graphique des relations

Une des tâches durant ce début de stage fut de concevoir une structure permettant de visualiser les relations entre les vampires, afin d'observer directement l'ensemble des échanges ayant lieu

au sein de la colonie.

L'idée est de représenter les vampires d'un même sous-groupe sur un cercle, et de disposer ces mêmes cercles à nouveau sur un cercle. L'objectif que je me suis ensuite fixé était que cette structure puisse s'adapter à d'autres systèmes de manière générique. J'ai alors établi les spécifications suivantes :

- Le graphe pourra contenir plusieurs niveaux d'agrégation. Il pourra ainsi représenter une quelconque classification arborescente. Par la suite on notera A(CG) l'arbre associé au graphe circulaire CG. Par exemple :

  Soit trois groupes de vampires, avec deux groupes qui en contiennent 3 et le troisième qui en contient 4, on obtient le graphe CG suivant :

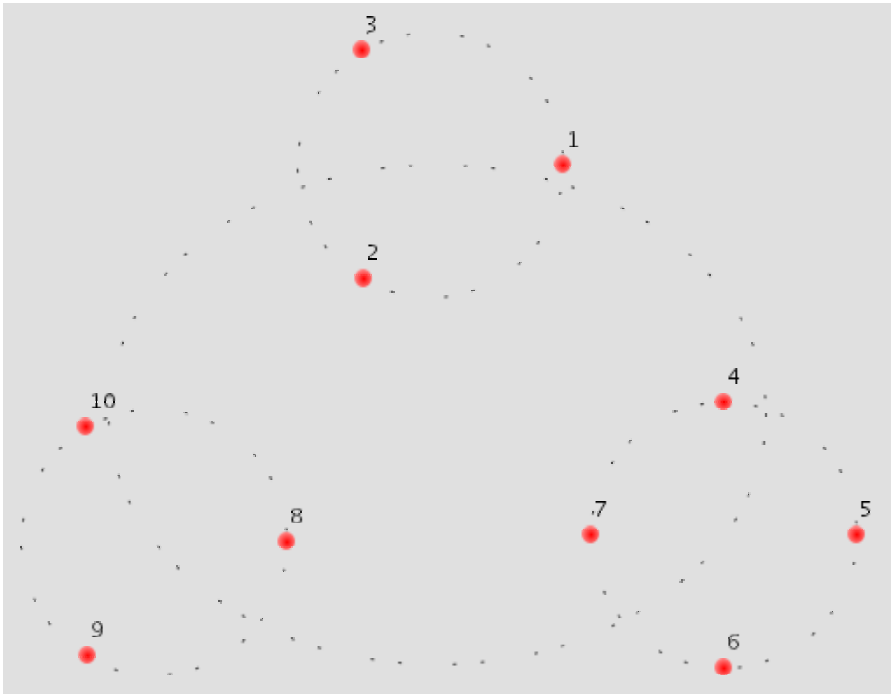
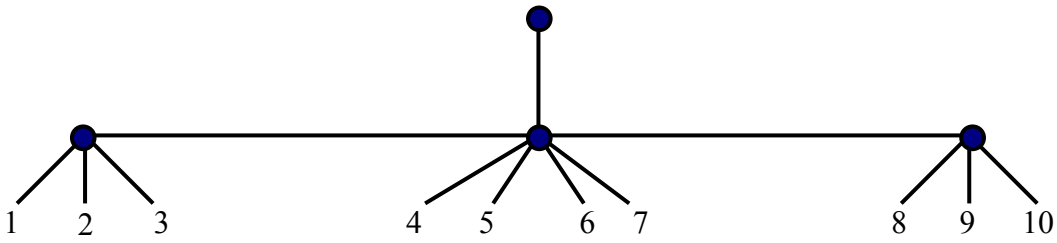


Figure 2.9 *Exemple de graphe circulaire*

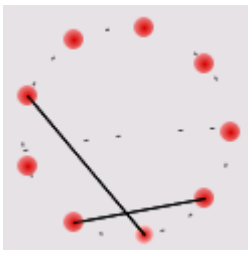
Et l'arbre A(CG) correspondant est :



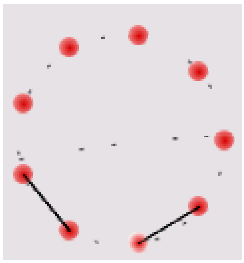
- Chaque nœud du graphe pourra être soit un nouveau graphe, soit une feuille, c'est-à-dire un des objets issus de la classification.
- Un lien pourra éventuellement être ajouté entre deux objets. Mais ce lien ne pourra être établi

qu'entre deux nœuds qui ont la même profondeur dans l'arbre

- On disposera d'une fonction qui permettra de rapprocher les objets qui sont le plus liés dans un même graphe de manière à clarifier la représentation graphique. Par exemple :



Avant



Après

Voici le diagramme de classe issu de ces spécifications :

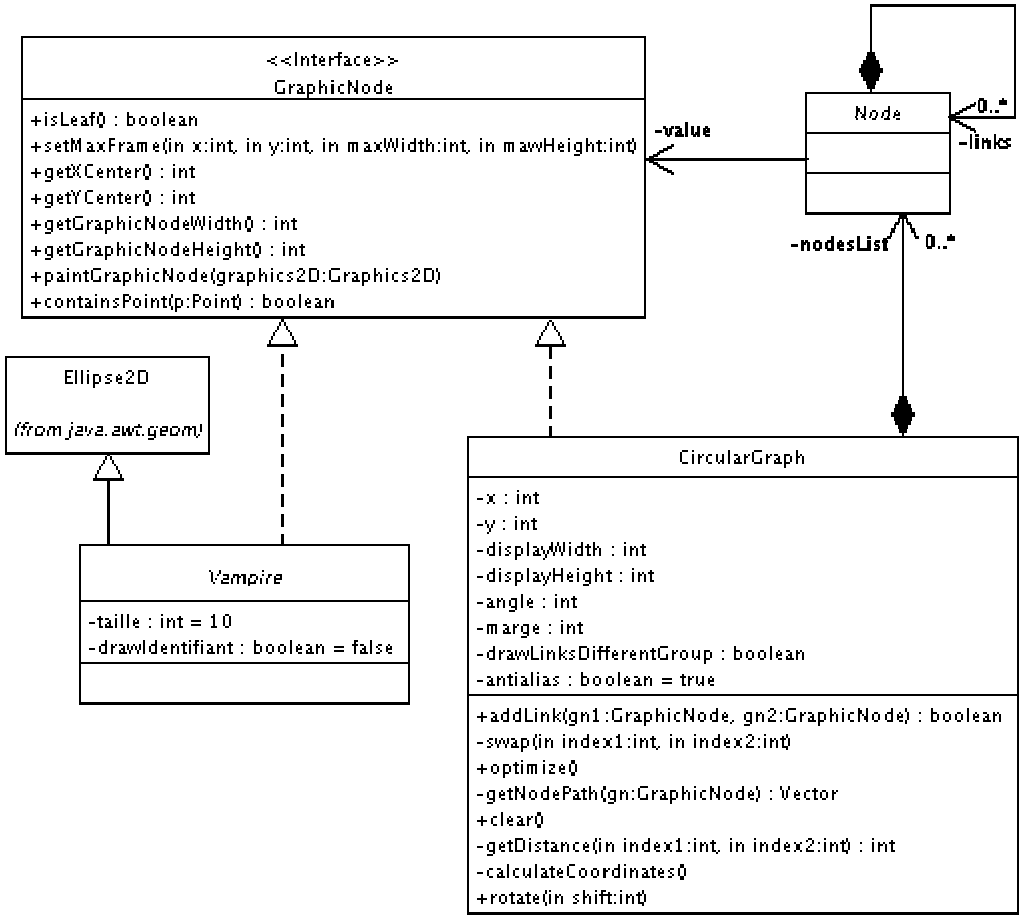


Figure 2.10 *Diagramme de classe UML : représentation graphique des relations*

La classe principale est `CircularGraph`, elle possède une liste de nœuds qui ont comme contenu soit une nouvelle instance de `CircularGraph`, soit un objet qui implémente l'interface `GraphicNode`, dans notre cas il s'agit d'un objet `Vampire`. Ce polymorphisme permet d'avoir

plusieurs niveaux d'agrégation, comme souhaité lors de nos spécifications. Un nœud peut comporter plusieurs liens vers d'autres nœuds qui ont la même profondeur dans l'arbre correspondant au graphe. L'interface `GraphicNode` contient les méthodes devant être implémentées par les objets à ajouter dans le graphe. Parmi ces méthodes, les plus importantes sont :

- `isLeaf()` renseigne si l'objet est une feuille (le cas échéant, il s'agit d'une instance de la classe `CircularGraph`).

- `paintGraphicNode()` permet de dessiner les objets contenus dans le graphe, ainsi lorsqu'un objet `CircularGraph` veut s'afficher, il appelle cette méthode pour tous les objets qu'il contient.

- `containsPoint()` nous indique si l'objet contient le point passé en paramètre. Cela servira notamment pour savoir à quel objet du graphe correspond un événement souris.

Notre classe `Vampire` implémente donc l'interface `GraphicNode` et hérite également de la classe Ellipse2D qui fait partie des classes fournies par le JDK, ce qui permettra de dessiner les vampires comme des cercles rouges (voir figure 2.9). Deux attributs lui ont également été ajoutés, `taille` définit le diamètre du cercle à dessiner et `drawIdentiant` définit si l'identifiant du vampire doit être affiché ou non (les afficher surcharge parfois l'affichage).

Revenons maintenant sur la classe `CircularGraph`, dont voici le détail de ses attributs et méthodes :

- `x`, `y`, `displayWidth`, `displayHeight` déterminent le rectangle dans lequel doit être dessiné le graphe à l'écran.

- `marge` est l'espace laissé sur le bord de la surface de dessin pour améliorer la lisibilité. Celle-ci va varier lorsqu'on ajoutera des nouveaux niveaux de graphes.

- `angle` est l'angle de rotation du graphe par rapport à sa position originale (voir méthode `rotate()` plus bas).

- `drawLinksDifferentGroup` indique si l'on doit dessiner les liens entre deux nœuds qui n'ont pas le même père (par exemple les nœuds 1 et 4 dans la figure 2.9).

- `antialias` sert à activer/désactiver l'option antialias dans le dessin des composants graphiques qui est géré par la machine virtuelle (fourni par le JDK). Cette option améliore sensiblement l'affichage en éliminant le crénelage qui peut apparaître mais peut parfois ralentir les performances.

- `addLink(gn1,gn2)` permet d'ajouter un lien entre deux nœuds. Cependant les deux nœuds doivent avoir la même hauteur dans l'arbre correspondant au graphe. Et cette méthode doit être appelée sur le graphe racine, car la méthode va également ajouter des liens (invisibles) entre les sous-graphes de même niveau qu'elle aura du parcourir, cela servira lors de l'optimisation du placement des nœuds (voir méthode `optimize()` plus bas).

- `swap(index1,index2)` permet tout simplement d'échanger le placement de deux nœuds, bien sûr ils doivent garder le même père donc cet échange ne peut se faire qu'entre deux nœuds qui ont le même père.

- `optimize()` est la fonction qui va permettre de repositionner au mieux les nœuds comme voulu dans les spécifications. La méthode utilisée est un algorithme que j'ai élaboré de manière empirique dont voici l'énoncé :

```
entrée : cg une instance de la classe Circulargraph

sortie : cg une instance de la classe Circulargraph optimisée

début

      pour chaque nœud node du graphe faire

            si node n'est pas une feuille alors node.optimize()

      fin pour

      pour chaque couple (nodei,nodej), nodei≠nodej, de cg.nodesList en
prenant d'abord les nœuds qui ont le plus grand ordre (nombre de liens) faire

            // on calcule la somme des distances avec les nœuds auxquels un

            // sommet est lié, et on soustrait la somme des distances en

            //cas d'échange des deux nœuds.

            gain_i = 0 ;

            gain_j = 0 ;

            pour chaque liens de nodei  avec linked_nodei faire

                  gain_i = gain_i + distance(node,linked_nodei) ;

                  gain_j = gainj – distance(node,linked_nodei) ;

            fin pour

            pour chaque liens de nodej  avec linked_nodej faire

                  gain_j = gain_j + distance(node,linked_nodej) ;

                  gain_i = gaini – distance(node,linked_nodej) ;

            fin pour

            // analyse des résultats

            si gain_i + gain_j > 0 alors

                  si (gain_i>0 OU nodei n'est pas marqué) ET

                  (gain_j>0 OU nodej n'est pas marqué) alors

                        échanger les deux sommets

                        marquer les sommets qui ont un gain strictement
                        positif

                  fin si

            fin si
```

```
      fin pour

fin
```

- `getNodePath(GraphicNode gn)` retourne un vecteur contenant le chemin complet de la racine jusqu'au nœud `gn`.

- `clear()` permet de vider entièrement le contenu du graphe.

- `getDistance(index1,index2)` retourne le nombre de pas qu'il y a entre deux nœuds qui sont sur un même graphe. Deux nœuds voisins auront une distance de 1, tandis que les nœuds 4 et 6 de la figure 2.9 auront une distance de 2.

- `calculateCoordinates()` recalcule toutes les coordonnées de tous les nœuds en fonction de l'état actuel du graphe.

- `rotate(int shift)` permet d'appliquer une rotation d'angle `shift` sur le graphe. Cette méthode est destinée à pouvoir modifier les angles de vue du graphe afin de le rendre plus clair.

3. L'application

3.1. Présentation générale

3.1.1. La fenêtre principale

Une fois le modèle implémenté, il fallut concevoir l'interface qui allait permettre de tester puis d'explorer le modèle. L'application se présente comme une fenêtre avec une barre de menu, un volet de visualisation avec deux onglets et des boutons pour un accès rapide aux fonctions importantes. Voici un aperçu lors du lancement de l'application :

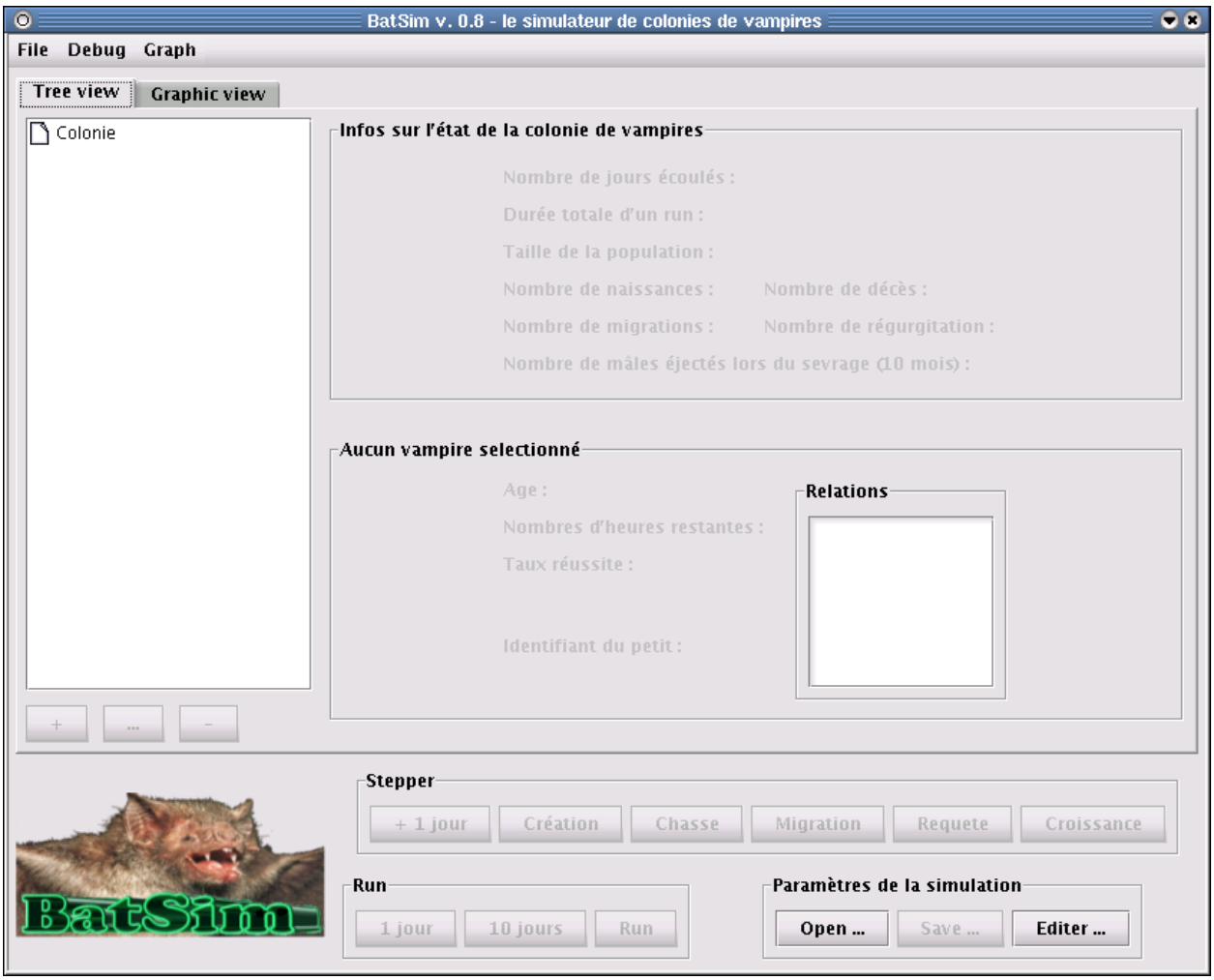


Figure 3.1 *L'application au démarrage*

On remarque la barre de menu qui permettra de charger/sauvegarder l'état d'une simulation, d'activer/désactiver l'affichage de la trace des actions effectuées (correction de bugs), et de paramétrer les options d'affichage du graphe circulaire. Nous retrouvons les deux onglets, le premier permettant de voir les principales caractéristiques de la colonie et d'accéder aux informations de chaque vampire en utilisant l'arbre qui est situé sur la gauche (qui est vide pour

l'instant), le second lui contient le graphe circulaire permettant de voir les échanges faits entre vampires. En dessous se trouve une rangée de boutons qui permettra de déclencher les étapes d'une journée une à une. Et encore en dessous, deux groupes de boutons, le premier permet de déclencher la simulation sur plusieurs jours, et le second permet de modifier/sauvegarder l'état de la simulation.

Pour l'instant, aucune initialisation n'a été faite, donc aucune information n'est encore affichée. Les seules actions possibles sont d'ouvrir un fichier, ou d'éditer les paramètres de la simulation afin d'initialiser une colonie de vampires.

3.1.2. Initialisation des paramètres

Il existe deux moyens pour initialiser les paramètres de la simulation :

- A partir d'un fichier, que nous choisissons à l'aide d'une boîte de dialogue (figure 3.2), à laquelle nous accédons par le menu « File→Open … » ou en cliquant sur le bouton « Open … ». La boîte de dialogue permet de visualiser un résumé du contenu du fichier sélectionné, car il n'est pas forcément évident de nommer un fichier de manière éloquente.

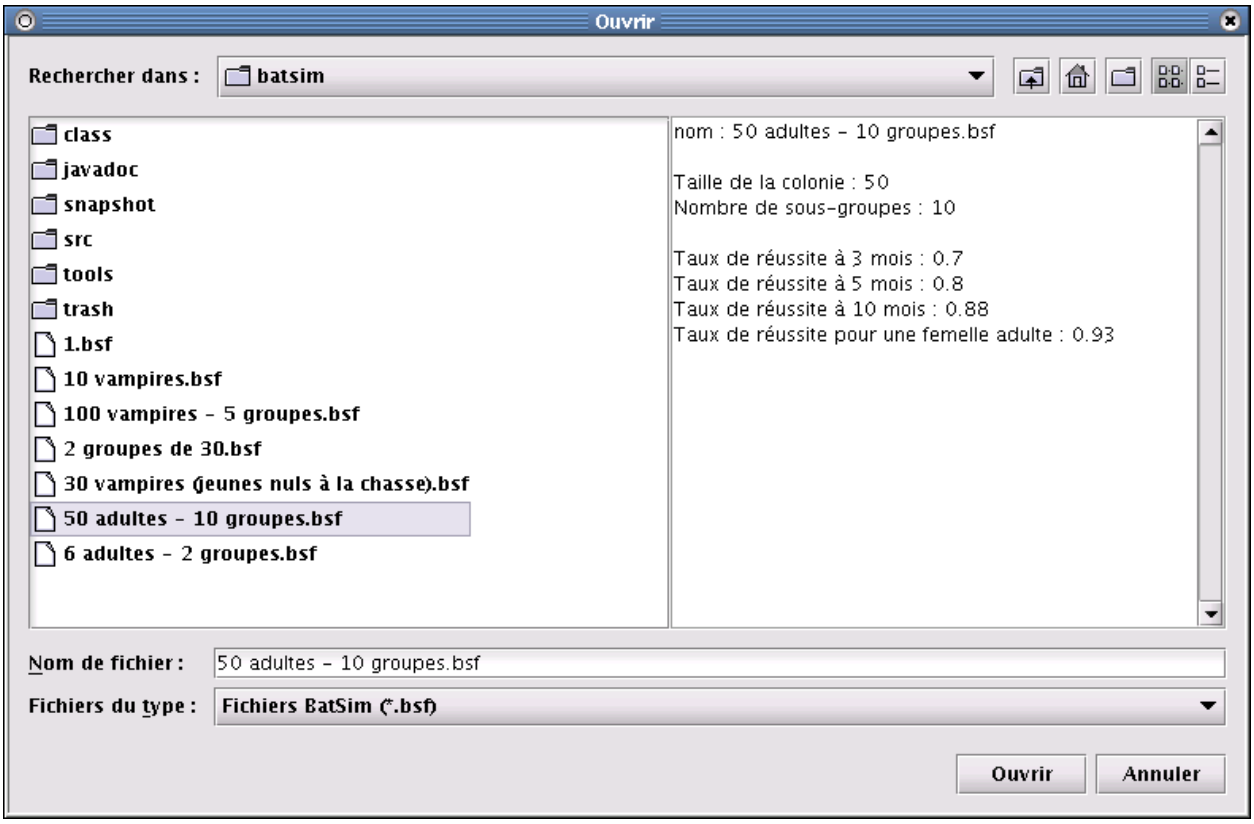


Figure 3.2 *Chargement d'un fichier*

Cette visualisation permet aussi de voir si un fichier n'est pas compatible (conflits de version, extension partagée par une autre application) comme illustré sur la figure 3.3.

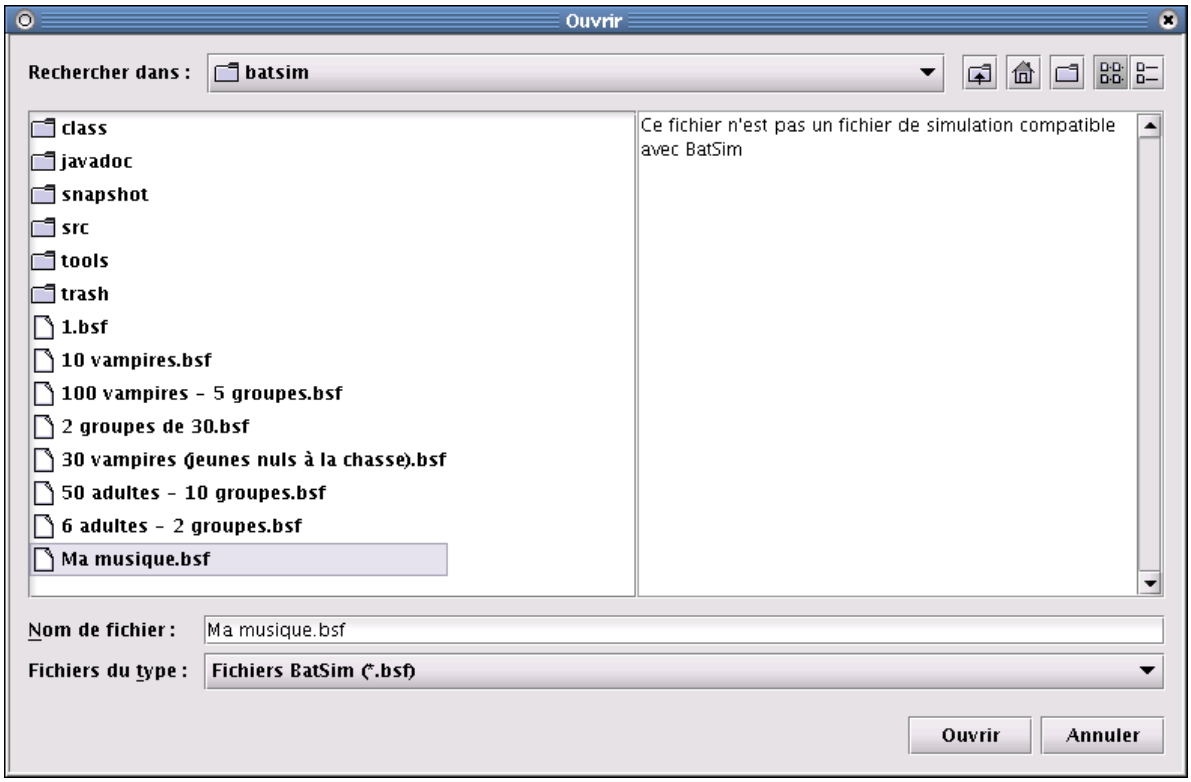


Figure 3.3 *gestion des fichiers non compatibles*

- En ajustant tous les paramètres à l'aide de la boîte de dialogue de la figure 3.4 qui est accessible par le bouton « Editer … ».

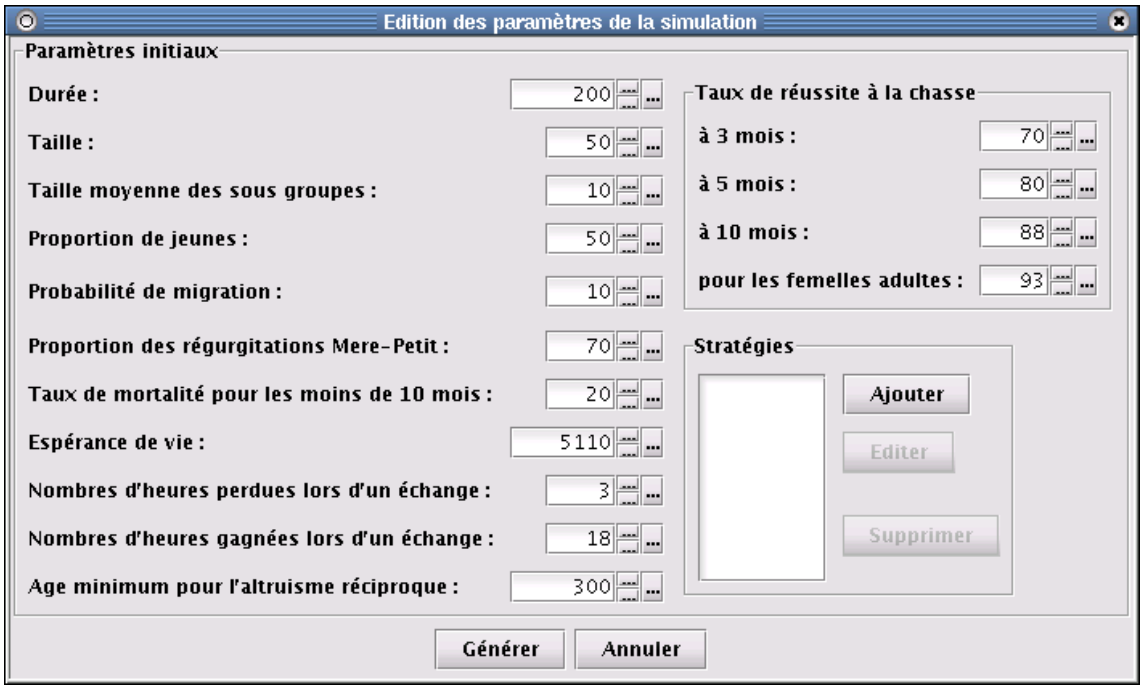


Figure 3.4 *Edition des paramètres de la simulation*

La case « Stratégies » va permettre d'ajouter/éditer/supprimer la liste des stratégies. Il est bien sûr possible d'initialiser une première fois le modèle, et ensuite de retoucher à quelques

paramètres.

3.1.3. Déclenchement de la simulation

Il se résume aux boutons déjà cités auparavant.

Figure 3.5 *Le « stepper »*

Le « stepper » (pas à pas) permet de déclencher une à une les étapes d'une journée afin d'observer avec précision les effets de ces différents événements.

Figure 3.6 *Les déclencheurs*

Ces trois boutons vont permettre de simuler une colonie sur 1 jour, 10 jours ou la durée prédéfinie lors de l'ajustage des paramètres.

3.2. Durant la simulation

Une fois la simulation initialisée, il existe de nombreux indicateurs de l'état du modèle qui permettent d'observer son évolution.

3.2.1. Indicateurs globaux

Ils sont regroupés dans la case « Infos sur l'état de la colonie de vampires » (figure 3.7) du premier volet (figure 3.1) et renseignent sur l'état global de la colonie et sur le nombre d'événements survenus.

Figure 3.7 *Les indicateurs globaux*

3.2.2. Indicateurs et modificateurs des individus

Toujours sur le premier volet, l'arbre situé à gauche permet de voir la répartition des vampires par catégories d'âge et par groupe (figure 3.8), d'ajouter/supprimer un vampire ou d'en éditer les propriétés.

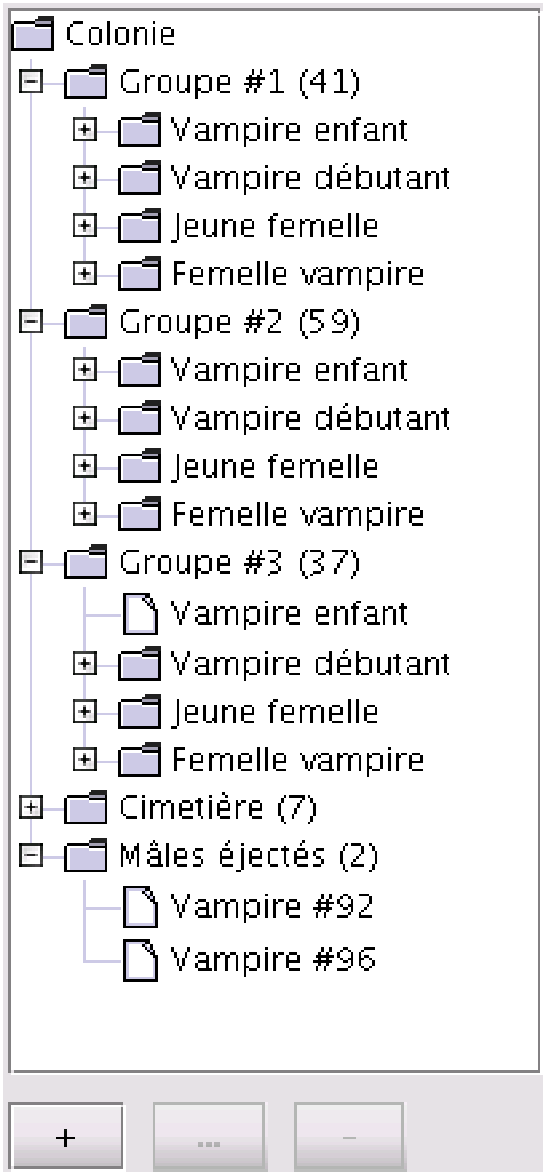


Figure 3.8 *Vue arborescente de la colonie*

Les vampires sont représentés par leur identifiant. Chaque groupe est affiché avec sa taille entre parenthèse, ce qui permet de voir rapidement la répartition des vampires dans la colonie.

Les boutons situés juste au-dessous permettent donc de manipuler les vampires. Grâce au premier nous allons pouvoir ajouter un nouveau vampire en spécifiant son âge et ses caractéristiques (voir figure 3.9), ces même caractéristiques vont pouvoir être modifier à l'aide du deuxième bouton (voir figure 3.10). Quant au troisième, il permet tout simplement de supprimer le vampire sélectionné.

Le fait de pouvoir éditer les propriétés d'un vampire est important, car, de cette manière, nous allons pouvoir introduire facilement des cas limites dans le modèle pour susciter des phénomènes plutôt difficiles à obtenir. Nous pourrons ainsi affiner la population de vampires qui a été générée de manière uniforme lors de l'initialisation.

Dans les propriétés d'un vampire, nous allons retrouver la liste de ses relations (figure 3.10), et de la même manière, que nous avons pu modifier les vampires, nous allons pouvoir modifier ces relations (figure 3.11). Une relation est représentée par l'identifiant de l'autre vampire, et ses événements par le couple [jour de l'événement] – [type de l'événement], la sémantique des types étant indiquée dans la boîte de dialogue correspondante (figure 3.11).

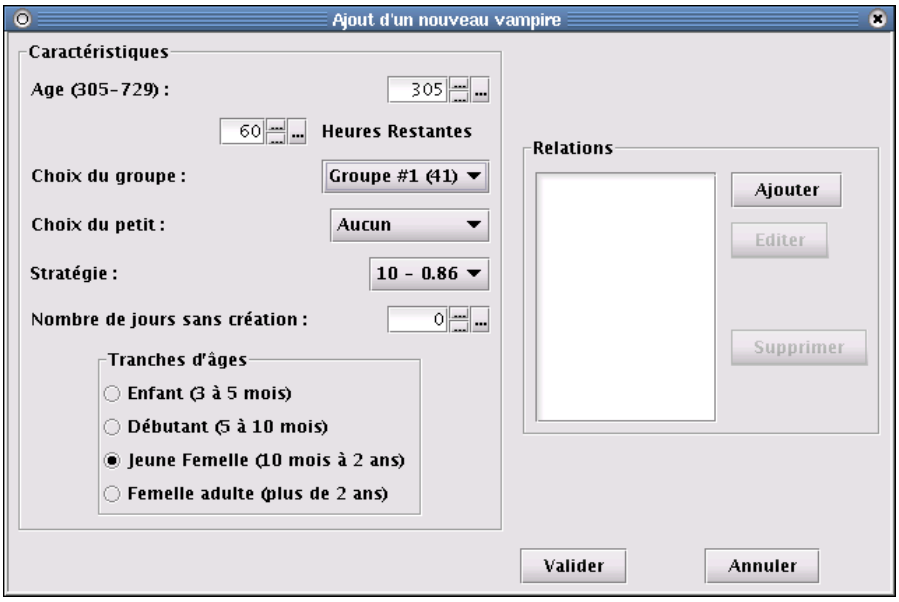


Figure 3.9 *Ajout d'un nouveau vampire*

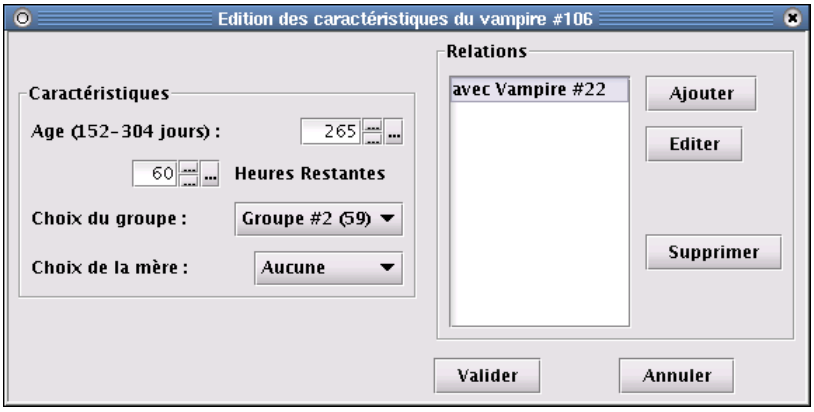


Figure 3.10 *Edition d'un vampire*

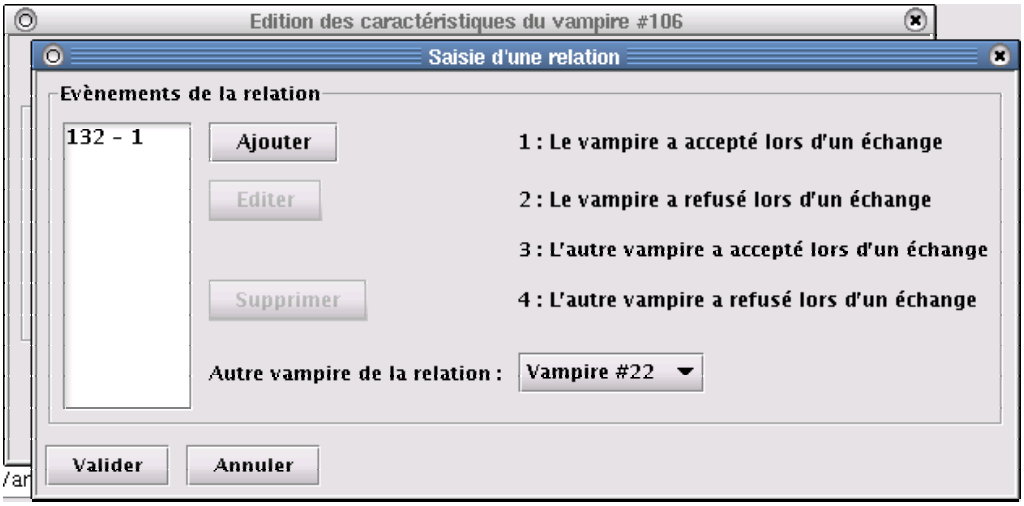


Figure 3.11 *Edition d'une relation*

3.2.3. Visualisation graphique

A l'aide de la structure développée au chapitre VIII.2.4, il est possible de visualiser directement la présence de « dialogues » entre les différents vampires. Ces derniers peuvent rapidement apparaître en grand nombre, l'algorithme précédemment exposé permet alors d'améliorer la lisibilité du graphe. Il est également possible d'appliquer une rotation sur chacun de graphes à l'aide de la souris, sur la figure 3.12 une rotation est appliquée sur le niveau du graphe représentant la colonie (la racine), et sur les figures 3.13 une rotation est effectuée uniquement sur un sous-groupe. L'option « Enable mouse zoom » permet d'activer/désactiver la fonction de zoom qui consiste à pouvoir élargir ou rétrécir le graphe à l'aide de la souris. Ces possibilités de pouvoir déformer le graphe ont pour but d'augmenter encore la lisibilité, en y ajoutant cette fois-ci une interactivité, en effet il est plus facile de percevoir un objet si l'on peut le manipuler.

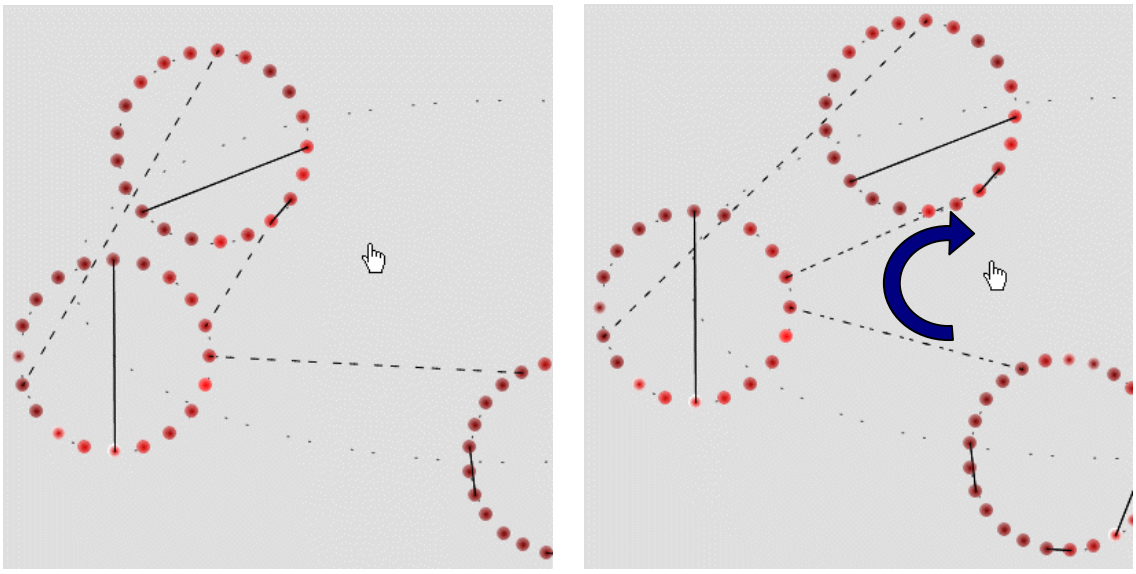


Figure 3.12 *Rotation au niveau de la racine du graphe*

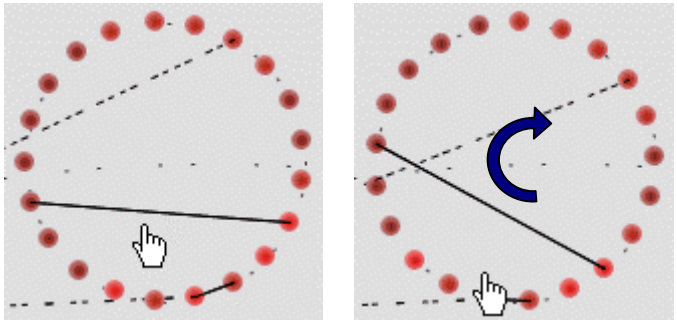


Figure 3.13 *Rotation d'un sous-groupe*

IX. Conclusion

1. Bilan

Le stage n'est pas terminé, compte tenu qu'il reste deux mois pour le réaliser. L'ensemble des objectifs prévus dans le planning a été atteint. Les principales difficultés rencontrées durant ce stage ont été :

- L'apprentissage de techniques de simulation multi-agents que je ne connaissais pas jusqu'alors.
- Le fait de mener ce projet seul. Néanmoins, le fait d'être seul a été particulièrement enrichissant, car cela m'a permis de couvrir tous les problèmes pouvant apparaître durant un projet de cette envergure.
- La gestion d'un aussi grand nombre de classes (la version actuelle de l'application comptabilise 42 classes et un peu plus de 6000 lignes de code).

La découverte de la technique de simulation par systèmes multi-agents qui est une technique largement utilisée dans la modélisation de systèmes complexes, a été un point fort de mon stage.

2. Perspectives

2.1. Exploration du modèle

Le modèle étant exploitable, il reste maintenant à l'explorer à la recherche de paramètres adéquats pour la colonie. Ces paramètres devront permettre des conditions de jeux idéales pour l'exposition, à savoir que les demandes de régurgitation doivent apparaître régulièrement et que la stratégie « donnant-donnant » reste la meilleure stratégie pour la survie de la colonie.

2.2. Réalisation du jeu

Un rendez-vous est prévu le 8 Juillet avec nos correspondants au Muséum d'Histoire Naturelle, afin de se mettre d'accord sur le scénario et le déroulement du jeu. Durant ce rendez-vous, nous élaborerons également les spécifications du contenu, et plus précisément des graphismes qui ne seront pas totalement livrés à cette date. De cette manière, je pourrais développer le jeu sur la base de ces spécifications, et prévoir des outils permettant de modifier tous les éléments graphiques du jeu, ainsi que les textes.

X. Références bibliographiques

Axelrod R. (1984) *The Evolution of Cooperation*. Basic Books, New York.

Bousquet F. (1994) *Des milieux, des poissons, des hommes : étude par simulations multi-agents.* Editions de l'ORSTOM, Paris.

Campos A. (2000) *Une architecture logicielle pour le développement de simulations visuelles et interactives individu-centrées : application à la simulation d'écosystèmes et à la simulation sur le Web.* Thèse de doctorat, Université Blaise Pascal Clermont-Ferrand II, 8 Septembre 2000.

Delahaye J.P. (1992) *L'altruisme recomposé ?.* Pour la science, n° 181.

Ferber J. (1995) *Les systèmes multi-agents. Vers une intelligence collective.* InterEditions, Paris.

Von Newman J. et O. Morgenstern (1947) *Theory of games and economic behavior.* Princeton University Press, Princeton.

Wilkinson G. (1990) *Food Sharing in Vampire Bats*. Univ. Press, Baltimore and London.