# Rapport projet JAVA

10 Janvier 2024

Clément Patrizio
clement.patrizio@ensta-bretagne.org
Tea Toscan du Plantier
tea.toscan@ensta-bretagne.org
Thibaut Vivier
thibaut.vivier@ensta-bretagne.org

## Résumé

Le but de notre projet est de modéliser un écosystème. Le cahier des charges nous imposait de modéliser les interactions entre deux animaux : des grenouilles (classe *Frog*) et des mouches (classe *Fly*), en utilisant les principes de programmation orientée objet, et *Java* comme langage de programmation. Nous avons ainsi modélisé ce qui était demandé, tout en allant plus loin : nous avons ajouté un troisième animal (un renard, classe *Fox*), et avons modélisé cet écosystème sous la forme d'un jeu, duquel l'utilisateur est maître. Ainsi, notre travail possède une interface graphique (classes *GameButton*, *GameFrame*, *GamePanel*), ainsi que des tests unitaires pour à peu près toutes les méthodes et fonctionnalités de notre code, regroupés dans le fichier "tests". Si les rendus informatique et graphique sont cohérents avec nos attentes et celles du cahier des charges, quelques limites subsistent du fait de nos choix de modélisation et de notre code intrinsèque.

# Table des matières

# 1 Introduction

Le projet s'inscrit dans le domaine des écosystèmes, définis comme des systèmes naturels dynamiques évoluants au fil du temps sous l'influence des êtres vivants et de leur habitat. La biocénose, représentée par l'ensemble des organismes vivants, démontre sa capacité évolutive et adaptative en réponse à un contexte écologique et abiotique en perpétuelle évolution. Cette capacité à absorber les impacts sans altérer la structure fondamentale de l'écosystème souligne l'importance de comprendre les interactions complexes au sein de ces systèmes.

L'objectif principal de ce projet est ainsi de créer une simulation d'écosystème, un ensemble vivant de différentes espèces interdépendantes, en relation les unes avec les autres et avec leur environnement à une échelle spatiale déterminée. A notre échelle, cette simulation est extrêmement basique, mais pourrait servir de base de travail à un programmeur désirant comprendre certains enjeux. En effet, la structure de notre projet, et surtout l'orientation objet que nous lui avons donnée, permet, en théorie, de simuler autant d'espèces que nécesaire, ainsi que chacune de leurs interactions. Bien sûr, cela prendrait un temps qui serait exponentiel en fonction du nombre de classes à créer mais ceci permet ainsi véritablement de se rendre compte de la complexité des systèmes qui nous entourent, et, par voie de conséquence, d'en apprécier le caractère essentiel. Si leur modélisation informatique n'est pas simpliste, la nécessité de les protéger afin d'en éviter le délitement devient alors un enjeu sociétal. C'est ainsi que nous nous sommes orientés vers le développement ludique d'un écosystème, tentant ainsi de faire saisir à nos petits frères et soeurs leur importance : chaque être vivant dans l'écosystème est nécessaire à la survie de l'autre, et c'est ce qui en permet la stabilité.

Nous envisagerons donc ici la simulation du fonctionnement d'un écosystème de mare, dont les animaux présents sont des grenouilles, des mouches et un renard, chacun ayant des interactions avec les deux autres. Tout l'enjeu est donc de savoir favoriser les interactions permettant la survie du renard.

# 2 Présentation du groupe et de la méthode de travail

Notre groupe est composé de 3 étudiants de deuxième année, tous ayant choisi la filière SOIA à l'ENSTA Bretagne : Clément PATRIZIO, Tea TOSCAN du PLANTIER et Thibaut VIVIER. Nous avons pris le parti, en plus des heures allouées à cette matière, de nous retrouver une fois supplémentaire par semaine afin de pouvoir développer convenablement le projet. Nous avons également travaillé en utilisant une version collaborative via GitHub, ce qui nous permettait de poursuivre notre développement une fois chacun rentré. L'architecture de notre projet a été définie assez vite, de manière à pouvoir organiser et séquencer notre travail. Nous fonctionnions en écrivant des "scripts", qui permettaient de s'assurer que notre travail avançait bien dans les temps, et de bien le cadrer (nous utilisions Trello). Chacun d'entre nous a été en charge d'une partie spécifique du code, puis du rapport.

Nous nous sommes assez vite orientés vers l'idée de développement ludique (sous forme

de jeu) de l'écosystème. Ainsi, outre les exigences du cahier des charges qui sont rappelées dans la partie suivante, nous avons développé plusieurs fonctionnalités permettant de rendre notre projet plus interactif, en y ajoutant des règles et en y ajoutant notamment un animal (le renard).

## 3 But de notre projet

Notre projet est créé sous forme de jeu. L'utilisateur est un renard (nommé Bruno), dont le but est de manger les grenouilles environnantes. Dans l'écosystème dans lequel évoluent le renard et les grenouilles se trouvent également des mouches. Les mouches, qui apparaissent aléatoirement dans l'écran de jeu, sont de la nourriture pour les grenouilles, qui sont attirées par elles. Cependant, ces mouches sont malheureusement nocives pour le renard : si celui-ci est piqué par une mouche, il meurt et l'utilisateur a donc perdu.

Tout l'enjeu pour l'utilisateur est donc de chercher à manger les grenouilles, sans se faire piquer par les mouches.

Notre projet peut être utilisé à la fois avec l'interface graphique que nous avons conçue, et sans.

## 4 Cahier des charges

Le cahier des charges impose plusieurs variables et méthodes à mettre en place au cours de ce projet dans certaines classes imposées. En effet, notre projet doit inclure trois fichiers : une classe *Fly* (Mouche), une classe *Frog* (Grenouille) et un fichier *Pond* (Mare). La classe *Fly* doit comporter les variables masse et vitesse. Elle doit aussi disposer des constructeurs suivants : un prenant en compte la masse et la vitesse, un prenant seulement la masse et un ne prenant aucun paramètre. Enfin, elle doit comporter des méthodes spécifiques. Elle doit contenir des setters et des getters pour toutes les variables, une méthode *toString* ne prenant aucun paramètre et retournant une chaîne de caractères décrivant la Mouche (si la masse est nulle (ou inférieure à 0, elle est considérée comme morte) ; une méthode *grow* prenant un entier représentant la masse ajoutée, puis augmentant la masse de la mouche. Si la masse est inférieure à 20, la vitesse augmente d'un pour chaque masse gagnée. Si la masse est de 20 ou plus, la vitesse diminue de 0,5 pour chaque masse gagnée. Enfin, la méthode *isDead* doit renvoyer *True* si la masse est nulle (ce qui implique de facto que l'animal est mort).

La classe *Frog* a, elle aussi, des contraintres. Elle devra contenir les variables *name, age, speed, isFroglet*, et *species*. Certaines méthodes sont également imposées : une prenant en compte le nom, l'âge et la vitesse de la langue, une prenant en compte le nom, l'âge en années (représentant l'âge de la grenouille en années sous la forme d'un double) et la vitesse de la langue. Une dernière doit prendre uniquement un nom, avec une grenouille par défaut à 5 mois et une vitesse de langue de 5. Plusieurs méthodes sont également attendues à l'intérieur de cette classe. La méthode *grow* doit prendre un nombre représentant le nombre de mois que la grenouille va vieillir. Il faut veiller à modifier l'âge

en années, à diminuer la vitesse de la langue d'un pour chaque mois de plus à partir de 30 mois, sans jamais descendre en dessous de 5, et à mettre à jour *isFroglet*. L'autre méthode est la méthode *eat* qui prend en paramètre une mouche à tenter d'attraper et de manger. Si la mouche est morte, la méthode est interrompue. La mouche est attrapée si la vitesse de la langue est supérieure à la vitesse de la mouche. La grenouille vieillit alors d'un mois en utilisant la méthode *grow*.

Le dernier fichier imposé est le fichier *Pond.java*. Ce fichier Java est un gestionnaire, ce qui signifie qu'il doit contenir et exécuter des objets *Frog* et *Fly* et piloter leurs valeurs selon un ensemble simulé d'actions. Ce fichier doit contenir différentes méthodes qui doivent par exemple créer 4 grenouilles de la classe *Frog* avec différents noms et différentes variables.

Afin de remplir ce cahier des charges, nous avons d'abord mis en œuvre ces méthodes. Cependant, lors de la programmation de notre jeu, certaines de ces méthodes et variables nous ont semblé moins pertinentes, c'est pourquoi nous avons décidé de les retirer de notre code.

## 5 Structure du projet

Afin de rendre notre projet jouable, et agréable à manier pour l'utilisateur, le but a été de créer une interface graphique. Nous allons donc avoir trois classes importantes : *PondApplication.java* qui régit le déroulé de l'application en elle-même (création des objets utilisés lors du jeu, initialisation de la fenêtre graphique, évolution des objets dans le temps), *GameFrame.java* qui est en réalité l'objet illustrant la fenêtre graphique. Cette fenêtre écoute les touches pressées du clavier pour, notamment, bouger le personnage (qui est un renard dénommé Bruno) contrôlé par l'utilisateur. *GamePanel.java* est la classe représentant la mare, un objet graphique auquel on peut ajouter un fond visuel (présenté Figure **??**) et ajouter des animaux.

Figure 1 – Image initiale du Jeu

Pour modéliser les animaux, nous avons deux classes primordiales. La première est *Animal*. Cette classe modélise l'animal que l'on veut ajouter à la mare. Elle contient les méthodes fondamentales communes à tous les animaux comme *render* pour l'afficher dans la marre, *kill* pour le tuer, etc. La seconde classe utile est *GameButton*. Cette classe est issue de *JButton*, qui sera le bouton utilisé pour représenter physiquement l'animal dans la mare.

Enfin, nous pouvons créer 3 animaux différents : la mouche, la grenouille et le renard. Ces classes sont issues de la classe *Animal* et ajoutent des spécificités telles que *eat*, *move* (puisque les animaux ne mangent pas tous et ne bougent pas tous de la même manière) ou encore la variable *TongueSpeed* pour la grenouille par exemple. Chacune des variables propres à chaque classe est décrite dans le paragraphe suivant.

Vous trouverez le diagramme de nos classes dans l'annexe A.

### 5.1 La classe *Animal*

Cette classe est la classe de base de notre projet.

Elle comprend un constructeur vide (puisque tout est spécifié dans les classes *Fly*, *Frog* et *Fox*). En revanche, elle possède des variables communes à chaque animal (qui seront toutes spécifiées en fonction de l'animal) : un type *type*, un nom *name*, une masse *mass*, une position $x$ (horizontale) et $y$ (verticale), un "cercle de survie" *radius*, un cercle d'attaque (qui, intersecté avec le cercle de survie, détermine si l'animal est tué ou pas) *attackRadius*, une vitesse *speed*, et enfin des éléments nécessaires au développement du jeu (*button*, et *pond*).

Dans cette classe, nous précisons aussi également les éléments imposés par le cahier des charges, tels que *isDead*, *kill*, qui déterminent si l'animal est mort, ou qui le tuent (en mettant une masse inférieure ou égale à 0). Nous avons également ajouté des méthodes *findNearestObject*, *findNearestFly* et *findNearestFrog*, qui permettent de déterminer respectivement les animaux, les mouches et les grenouilles les plus proches de l'animal qui appelle cette méthode. Ces méthodes appellent les listes d'animaux, et trouvent le plus proche au sein de cette liste. *findNearestFly* et *findNearestFrog* précisent la mouche ou la grenouille la plus proche. Enfin, la méthode *render* permet l'affichage dans notre interface graphique, tandis que la méthode *toString* permet l'affichage des éléments dans le terminal.

### 5.2 La classe *Fly*

Cette classe est spécifique à la mouche.

Elle comprend un constructeur spécifiant les différentes variables associées à celle-ci, introduites dans la classe *Animal*. En plus de cela, elle comprend la méthode *move* qui permet de faire bouger la mouche. Celle-ci bouge aléatoirement à une vitesse de 10 pixels lorsqu'elle ne perçoit aucun risque par jour. En revanche, si une grenouille tente de la manger mais qu'elle n'y parvient pas, la mouche perçoit alors le danger et s'enfuit à une vitesse deux fois plus rapide vers le centre de l'écran (méthode *flee*). De plus, la mouche perd une certaine masse lorsqu'elle vole, ainsi si elle ne mange pas, sa masse réduit peu à peu jusqu'à ce qu'elle meure. Enfin, la méthode *sting* permet à la mouche de piquer le renard, s'il n'est pas mort. Le renard est alors tué.

### 5.3 La classe *Frog*

Cette classe est spécifique aux grenouilles.

Elle comprend également un constructeur spécifiant les différentes variables associées à celle-ci. Elle comprend aussi la méthode *move* qui fait bouger les grenouilles grâce à la méthode *goTo* qui les déplace. Les grenouilles se déplacent en fonction des mouches présentes autour d'elles. En effet, s'il reste au moins une mouche vivante, la grenouille se déplace vers la mouche la plus proche, afin de tenter de la manger. Sinon, elle se déplace aléatoirement. De plus, la méthode *eat* permet à une grenouille de manger une mouche. Si la distance entre la mouche et la grenouille est inférieure au rayon d'attaque (*attackRadius* de la grenouille, alors la mouche est mangée si sa vitesse est inférieure à la vitesse de la langue de la grenouille (*tongueSpeed*. Cependant, si la vitesse de la mouche est supérieure, la grenouille échoue à la manger et la mouche s'échappe (méthode *flee* précédemment décrite).

### 5.4 La classe *Fox*

Identiquement aux classes spécifiques aux animaux précédentes, la classe *Fox* contient elle aussi un constructeur explicitant les différentes variables associées au renard, ainsi qu'une méthode *move* pour le faire bouger et une méthode *eat* qui lui permet de manger. Le renard est déplacé par le joueur. Ce dernier s'aide du clavier pour déplacer l'animal :

s'il appuie sur la flèche de droite le renard se déplacera à droite, et de même pour les flèches de gauche, du haut et du bas. Le renard mange les grenouilles, ainsi l'utilisateur déplacera logiquement le renard vers les grenouilles à attraper. Si la distance entre le renard et la grenouille est inférieure au rayon d'attaque du renard, alors la grenouille est mangée. L'utilisateur doit cependant faire attention à ne pas se faire piquer par une mouche, auquel cas il meurt.

### 5.5 Les classes *GameButton, GameFrame, GamePanel*

Ces classes sont celles qui nous permettent d'obtenir une interface graphique, permettant au joueur de contrôler son renard. Chaque animal de l'écosystème est représenté par un bouton (*GameButton*, issue de *JButton*) dans l'interface graphique, qui intègre le bouton en son sein au moyen de la classe *GamePanel*. Le tableau *GamePanel* est inscrit dans la fenêtre *GameFrame* qui écoute les ordres donnés par l'utilisateur via son clavier. Ces ordres sont utilisés pour faire évoluer le renard dans son environnement. Les méthodes propres à ces classes sont détaillées une par une dans notre *javadoc*.

### 5.6 La classe *PondApplication*

Cette classe est celle qui nous permet concrètement de générer notre écosystème. Elle intègre toutes les classes dépendant de Animal et de l'interface graphique (*GameButton*, *GameFrame* et *GamePanel*), afin de rendre vivantes chacune de leurs interactions. C'est véritablement la classe "résumé' de notre jeu, elle intègre les précédentes pour permettre au tout de fonctionner en symbiose. Les méthodes propres à cette classe sont également détaillées dans notre *javadoc*.

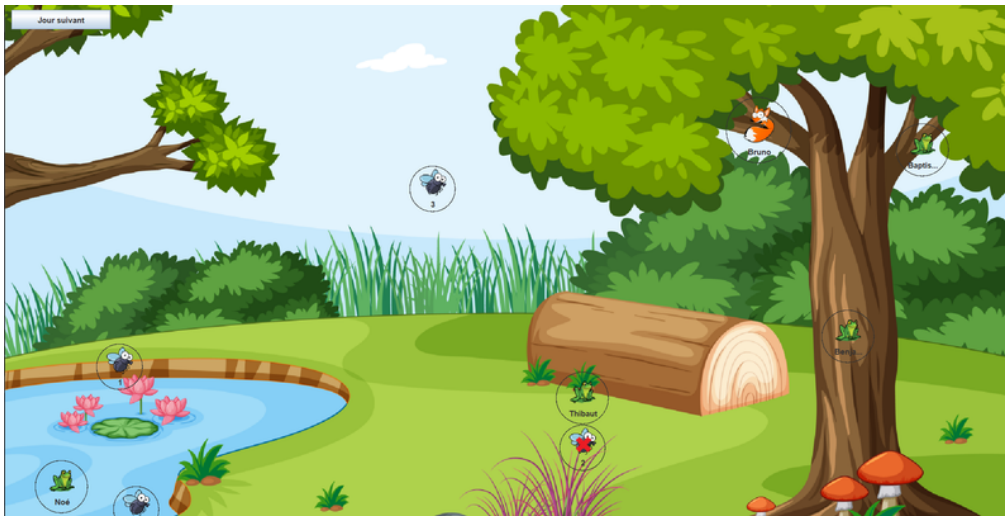


Figure 2 – Capture d'écran du jeu

# 6 Mise en forme du développement de l'application

## 6.1 Tests unitaires

Nous avons pris le parti de tester chacune des méthodes (objet) essentielles au bon fonctionnement de notre écosystème, et permettant à l'utilisateur d'aboutir au résultat attendu. Ainsi à chaque classe (objet) "*Class*" de notre projet correspond un fichier "*ClassTest.java*" qui teste les méthodes essentielles de la classe. Pour cela, nous avons utilisé JUnit 4, le framework de test qui nous a été conseillé. Chacun des tests que nous avons effectués est construit de la même façon : avant chaque test, nous générons une nouvelle instance de notre jeu (ou de notre classe à tester), puis nous testons au moyen d'assertions (*assertEquals* ou *assertTrue*) les méthodes principales de notre classe. Cela évite ainsi les interactions non désirées entre les tests et garantit également que chaque test s'exécute dans un état initial prévisible. Ainsi, en cas d'échec du test, il nous était renvoyé directement une erreur, ce qui nous imposait de devoir la corriger pour assurer une compilation globale de l'ensemble de nos tests.

## 6.2 Documentation

Les classes sont toutes documentées ce qui nous a permis de générer des fichiers HTML avec JavaDoc. Ces fichiers nous permettent de visualiser agréablement la documentation du projet. Nous avons ensuite hébergé ces fichiers sur github et construit un lien HTTPS via l'outil Github Pages. Ainsi vous pouvez retrouver la documentation de notre projet en suivant ce lien : https://clem-pat.github.io/EcosystemProject

# 7 Limites du projet

Les limites de notre projet sont notamment dues à plusieurs facteurs : les choix de modélisation que nous avons faits, ainsi que les limites de notre code.

Nous avons en effet pris le parti de modéliser un écosystème dont l'interaction avec l'utilisateur (notre joueur) devait être la plus naturelle et la moins complexe possible. Ainsi, certaines méthodes telles que *isFroglet* dans la classe *Frog* n'ont pas été implémentées, car "inutiles" à la compréhension de notre jeu, et n'apportant pas au produit final une plus-value déterminante. Cependant, nous avons parallèelement à cela ajouté des méthodes qui nous semblaient nécessaires comme par exemple la mise en place un système d'âge, qui implique qu'un animal de moins de 5 jours ne peut pas attaquer un autre animal. Nous avons estimé qu'étant donné que notre projet est un jeu duquel l'utilisateur est maître, ajouter une "artificialisation" du phénomène (en nommant précisément une grenouille, et en lui imposant d'aller manger une mouche donnée) nuirait à la version que nous avons voulu donner à notre écosystème. De fait, comme précisé dans les descriptions de nos classes, chaque mouche et chaque grenouille agit en fonction de son environnement propre, et est autonome dans sa manière de se comporter.

Notre projet aurait également pu être amélioré en apportant plus d'éléments pour améliorer les règles du jeu, assez simplistes, ainsi qu'en ajoutant par exemple un tableau

permettant de voir en direct l'évolution du score, ou alors d'avoir certaines formes de "bonus" en fonction de leur action.

# 8 Conclusion

Ainsi, ce projet nous a permis d'expérimenter la programmation orientée Objet, à travers la prise en main d'un nouvel outil pour nous : Java. Nous avons pu mettre en oeuvre un certain nombre d'acquis au cours de notre travail, comme par exemple la manipulation des héritages, des constructeurs et dépendances, des diagrammes de classe, et de la documentation *javadoc*. A travers un cahier des charges précis, nous avons essayé d'aller plus loin que ce qui nous était initialement demandé, en ajoutant une interaction entre l'objet et l'utilisateur, à travers notre "jeu", et en ajoutant des méthodes à celles initialement demandées. Ceci nous a semblé être la meilleure façon d'approfondir nos connaissances en Java, tout en réutilisant ce que nous avions appris auparavant, afin de nous rapprocher davantage de ce qu'on pourrait attendre d'un ingénieur : savoir mettre en oeuvre ses compétences, et surtout, savoir les réutiliser à bon escient.

# A Annexe

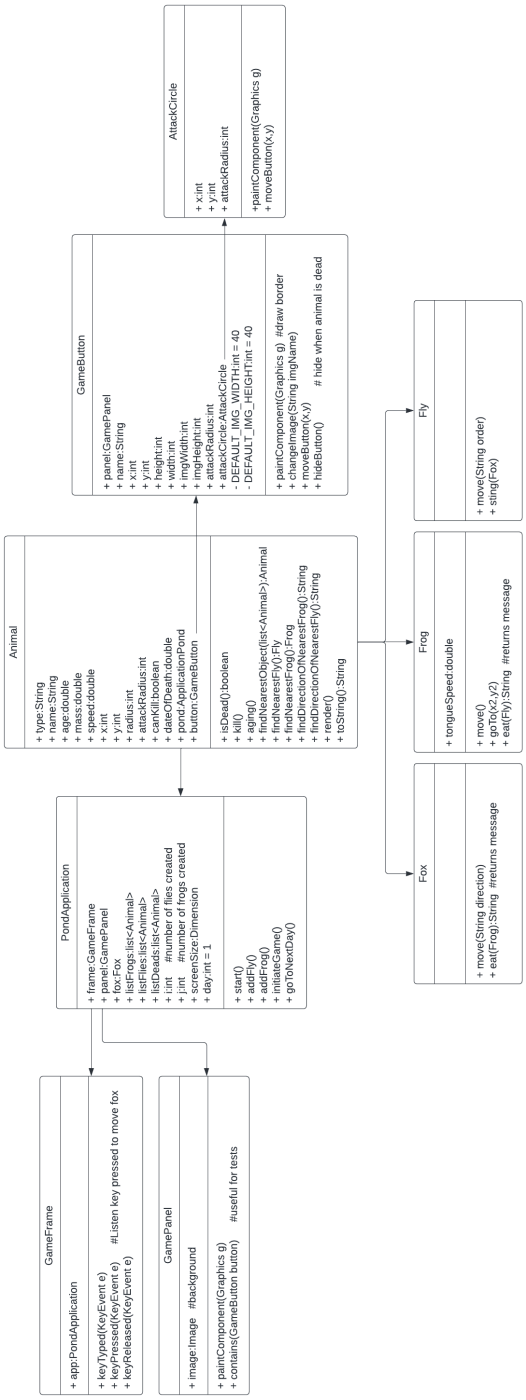


Figure 3 – Diagramme des classes

# Table des figures