

# **Rapport de projet**

Mahdi RANJBAR

10 janvier 2022

# Table des matières

---

Introduction	3
Analyse globale	4
Plan de développement	5
Conception générale	8
Conception détaillée	9
Résultat	17
Documentation utilisateur	20
Documentation développeur	21
Conclusion et perspectives	22

# Introduction

---

Ce mini-projet a pour but de mener à bien le développement d'un jeu étant inspiré par le jeu "Flappy Bird" dans lequel un ovale se déplace le long d'une ligne brisée. L'ovale descend sans cesse et clique sur l'interface, ce dernier saute vers le haut. Le but du jeu c'est de maintenir l'ovale en cliquant sur l'écran de façon que la ligne soit à l'intérieur de ce dernier. Voici un extrait de l'interface graphique de ce jeu:

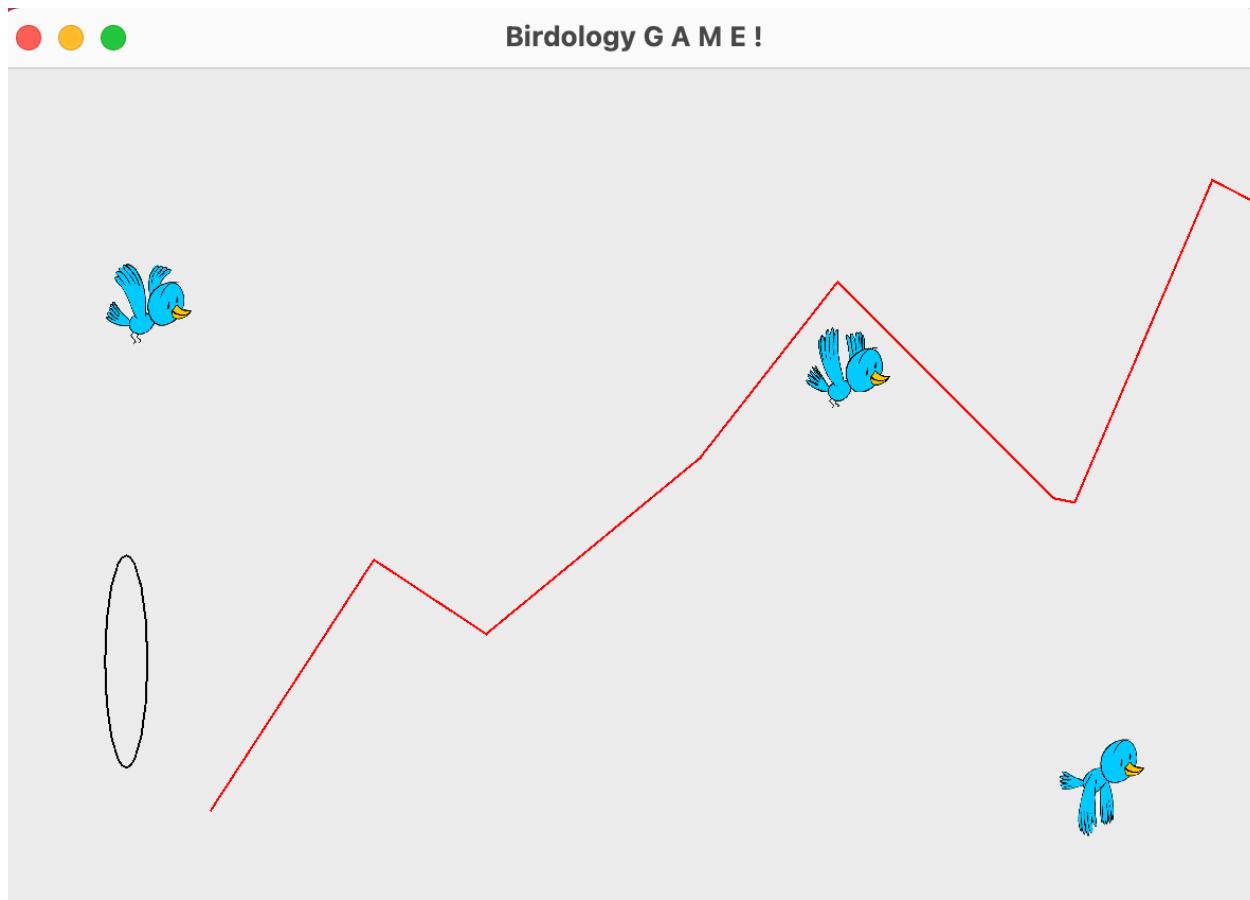


FIG. 1: Extrait de GUI du jeu

# Analyse globale

## I. La réaction de l'ovale aux clics de l'utilisateur

Sous-fonctionalité	Difficulté	Priorité
Création d'une fenêtre dans laquelle l'ovale est dessiné	Simple	Prioritaire
Déplacement de l'ovale vers le haut lorsqu'on clique dans l'interface	Simple	Prioritaire

## II. Le défilement automatique de la ligne brisée

Sous-fonctionalité	Difficulté	Priorité
Descente en permanence de l'oval	Simple	Prioritaire
Création de la ligne brisée	Simple	Prioritaire
Avancement en permanence de la ligne brisée	Modéré	Prioritaire

## III. L'interface graphique avec l'ovale et la ligne brisée

Sous-fonctionalité	Difficulté	Priorité
Détection des collisions entre l'ovale et la ligne brisée	Modéré	Prioritaire
Ajout d'éléments de décor	Modéré	Prioritaire

# Plan de développement

## I. La réaction de l'ovale aux clics de l'utilisateur

Liste des tâches	Temps estimé
Analyse du problème	30 min
Conception, développement et test d'un fenêtre avec un ovale	1h
Conception, développement et test du mécanisme de déplacement de l'ovale	2h
Acquisition de compétences en Swing	3h
Documentation du projet	30 min

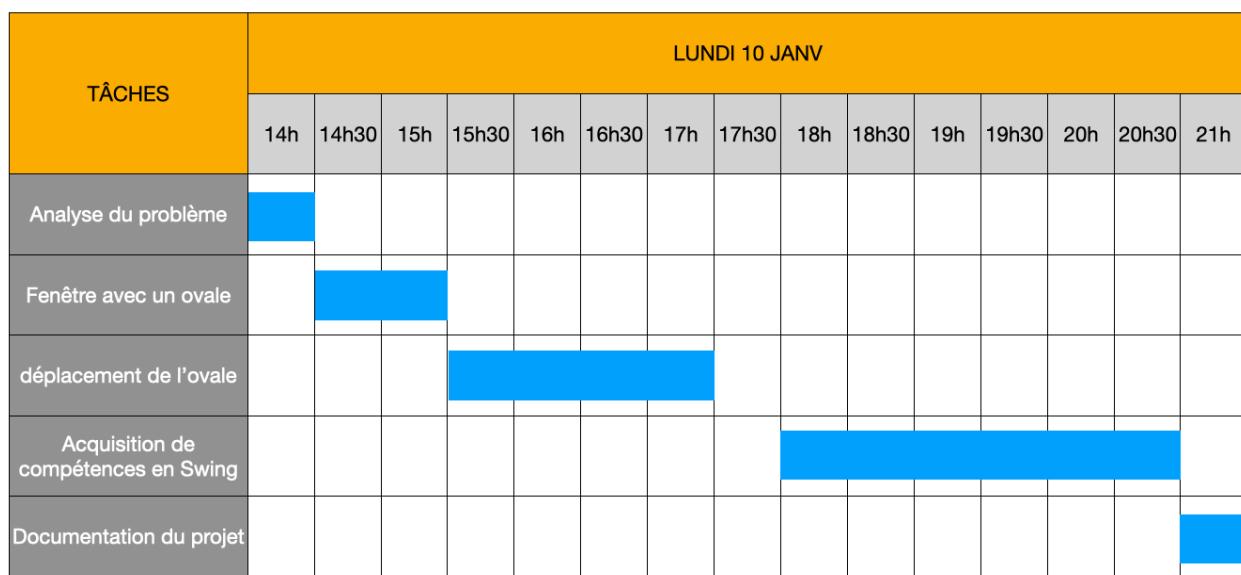


FIG. 2: Diagramme de Gantt 1ière séance

## II. Le défilement automatique de la ligne brisée

Liste des tâches	Temps estimé
Analyse du problème	30 min
Conception, développement et test du mécanisme de la descente automatique de l'oval	1h 30 min
Conception, développement et test de la création de la ligne brisée	1h 30 min
Conception, développement et test du mécanisme d'avancement automatique de la ligne brisée	2h
Conception, développement et test de la ligne brisée infinie aléatoirement	2h 30 min
Acquisition de compétences en Swing	1h 30 min
Documentation du projet	30 min

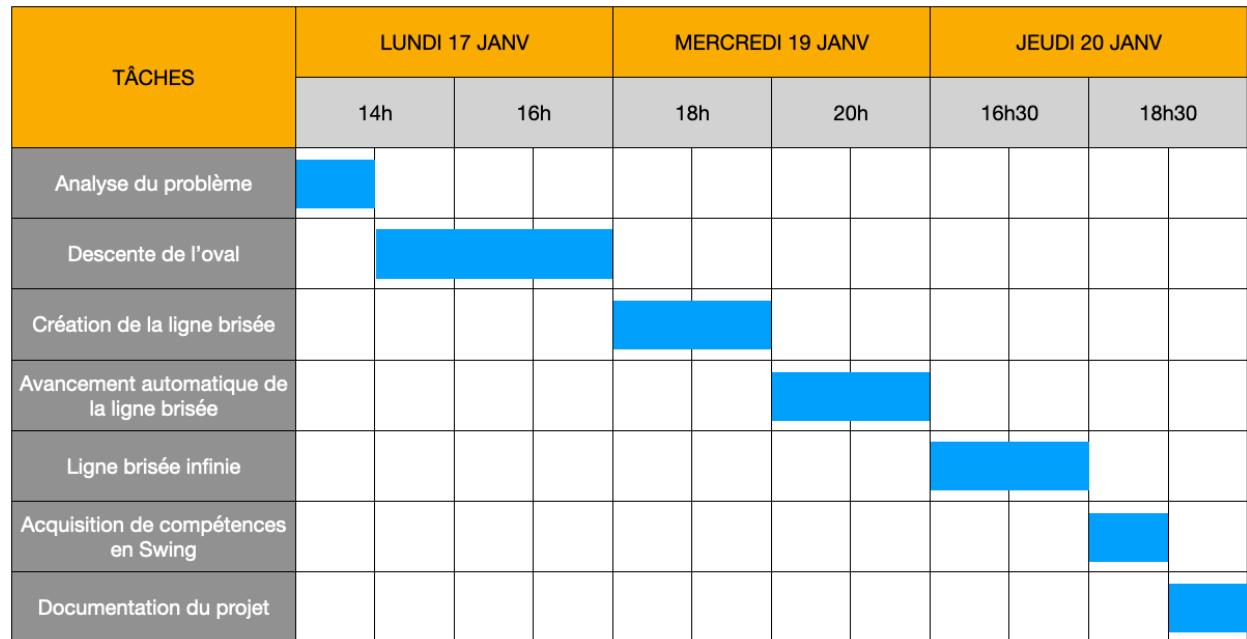


FIG. 3: Diagramme de Gantt 2ième séance

### III. L'interface graphique avec l'ovale et la ligne brisée

Liste des tâches	Temps estimé
Analyse du problème	30 min
Conception, développement et test du mécanisme de détection des collisions	2h 30 min
Conception, développement et test de la création des oiseaux	4h
Acquisition de compétences en Swing	2h
Documentation du projet	1h 30 min

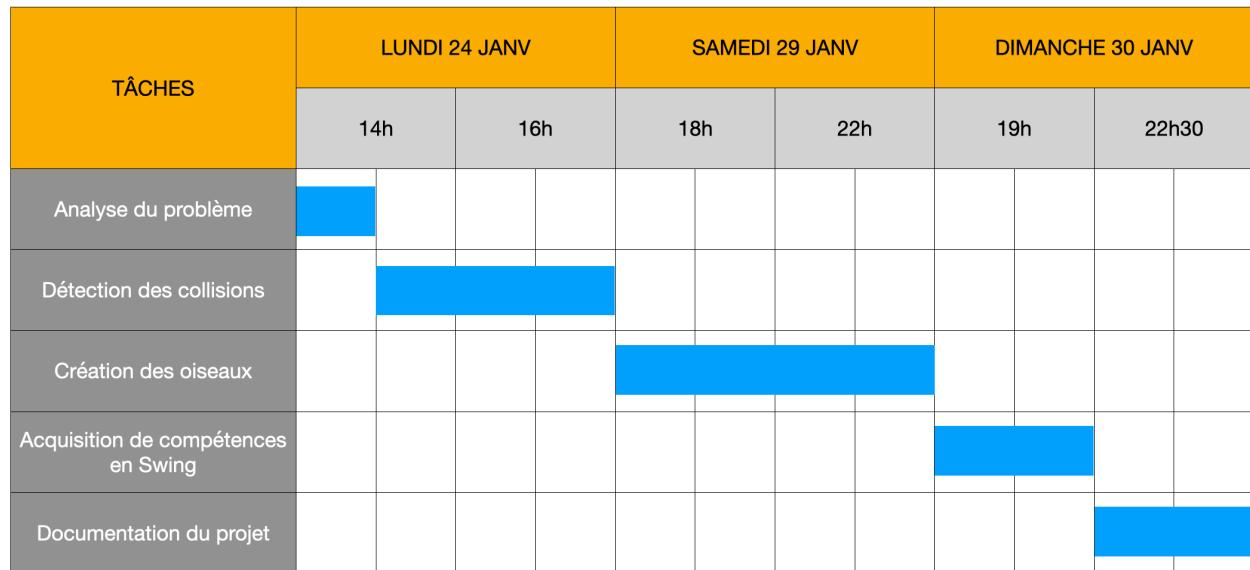
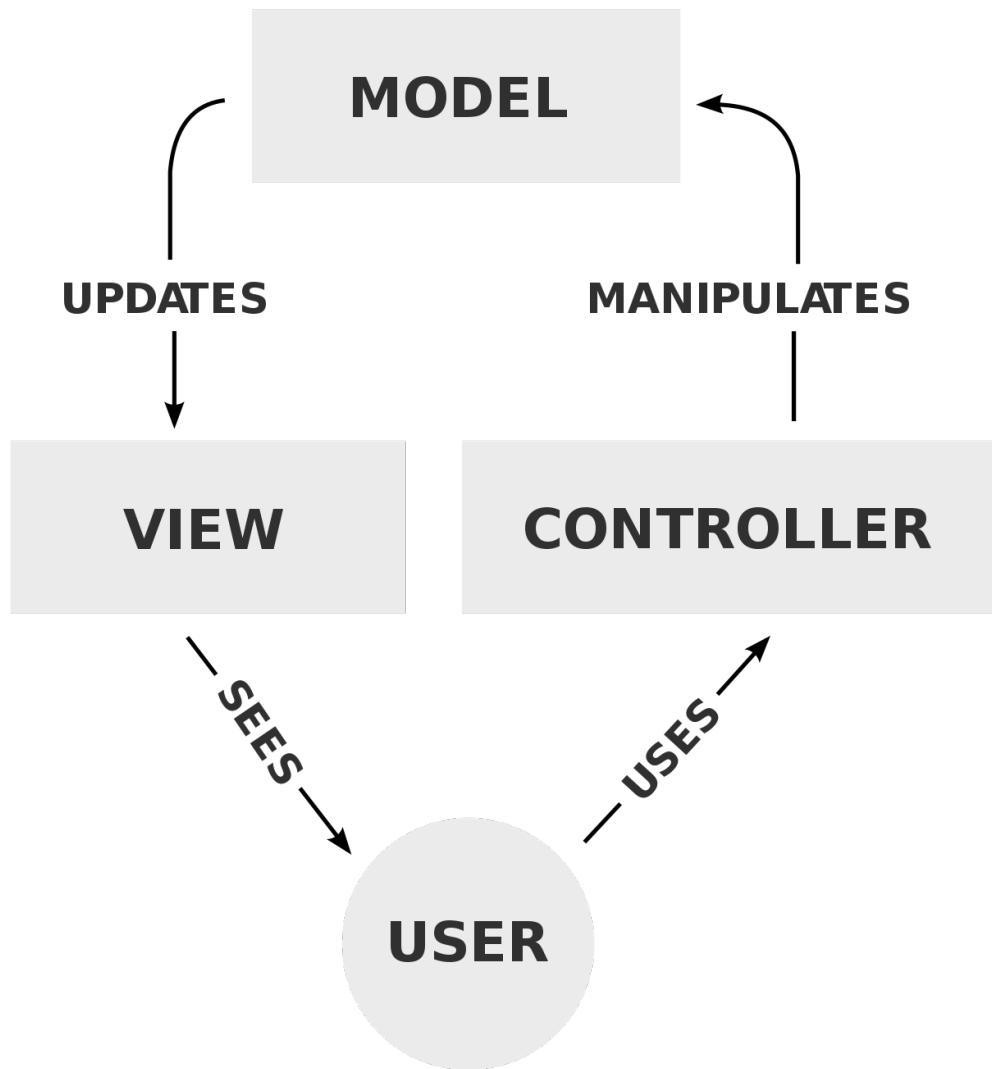


FIG. 4: Diagramme de Gantt 3ième séance

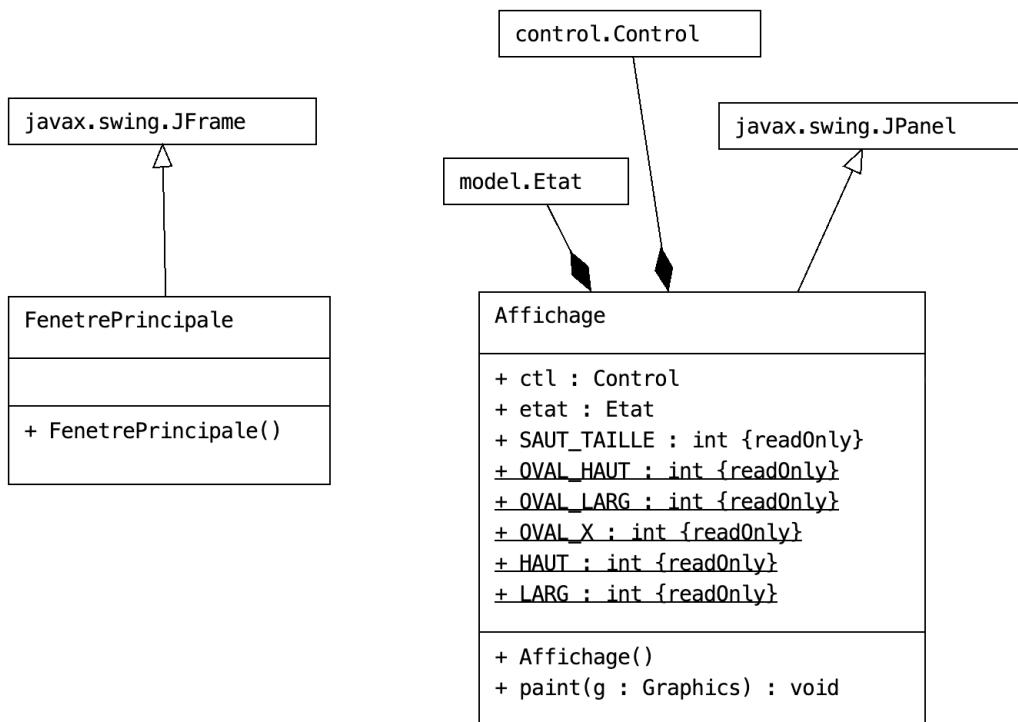
## Conception générale

Nous avons adopté le motif MVC pour le développement de notre interface graphique. L'affichage de l'ovale, de la ligne brisée ainsi que des oiseaux sont associés au bloc View. Pour le déplacement de l'ovale vers le haut lorsqu'on clique dans l'interface, sachant qu'il s'agit de gestion de l'événement clique souris, est donc associé au bloc Controller. Enfin, le jeu de données qui caractérise l'état de l'interface et que la modification de ces données effectue un changement de l'affichage dans l'interface graphique, est associé au bloc Model.

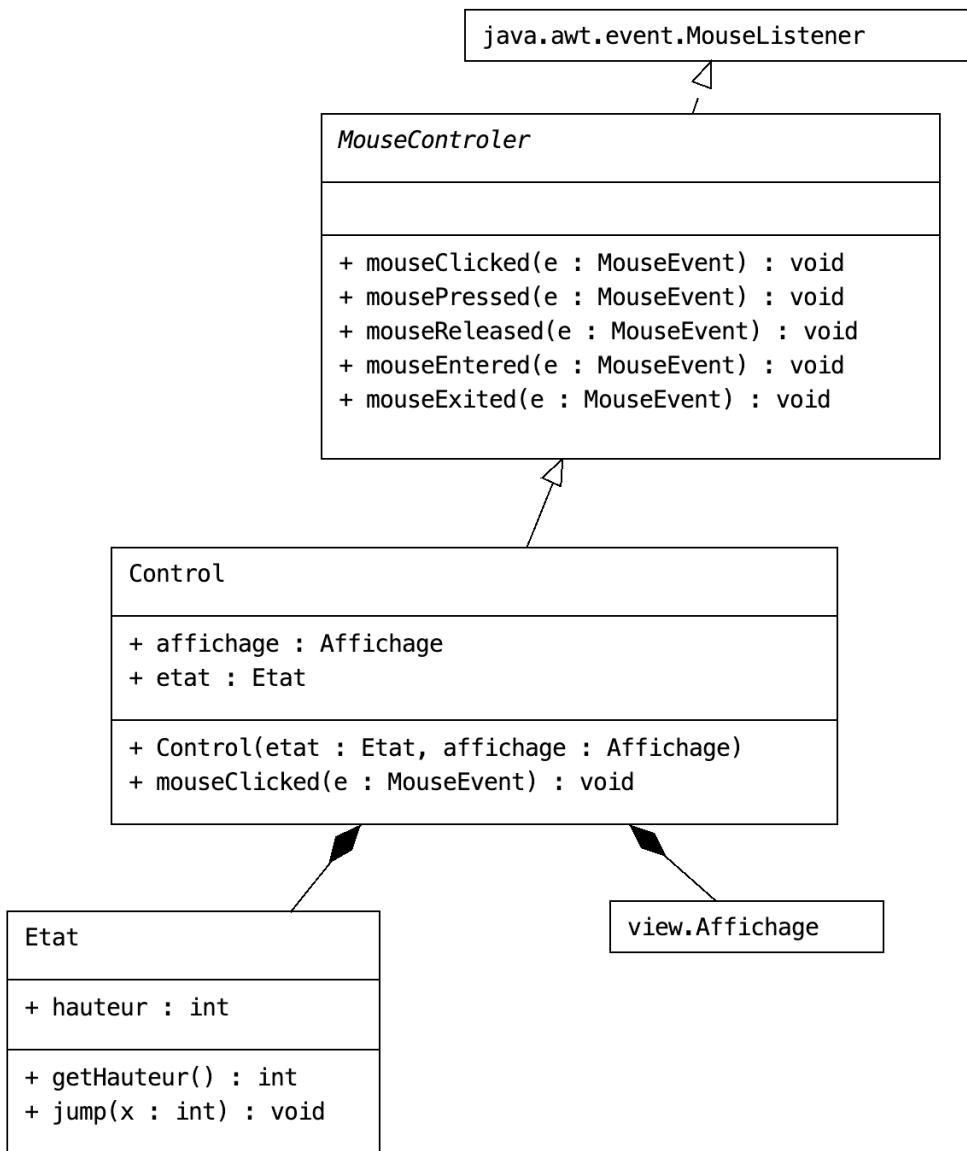


# Conception détaillée

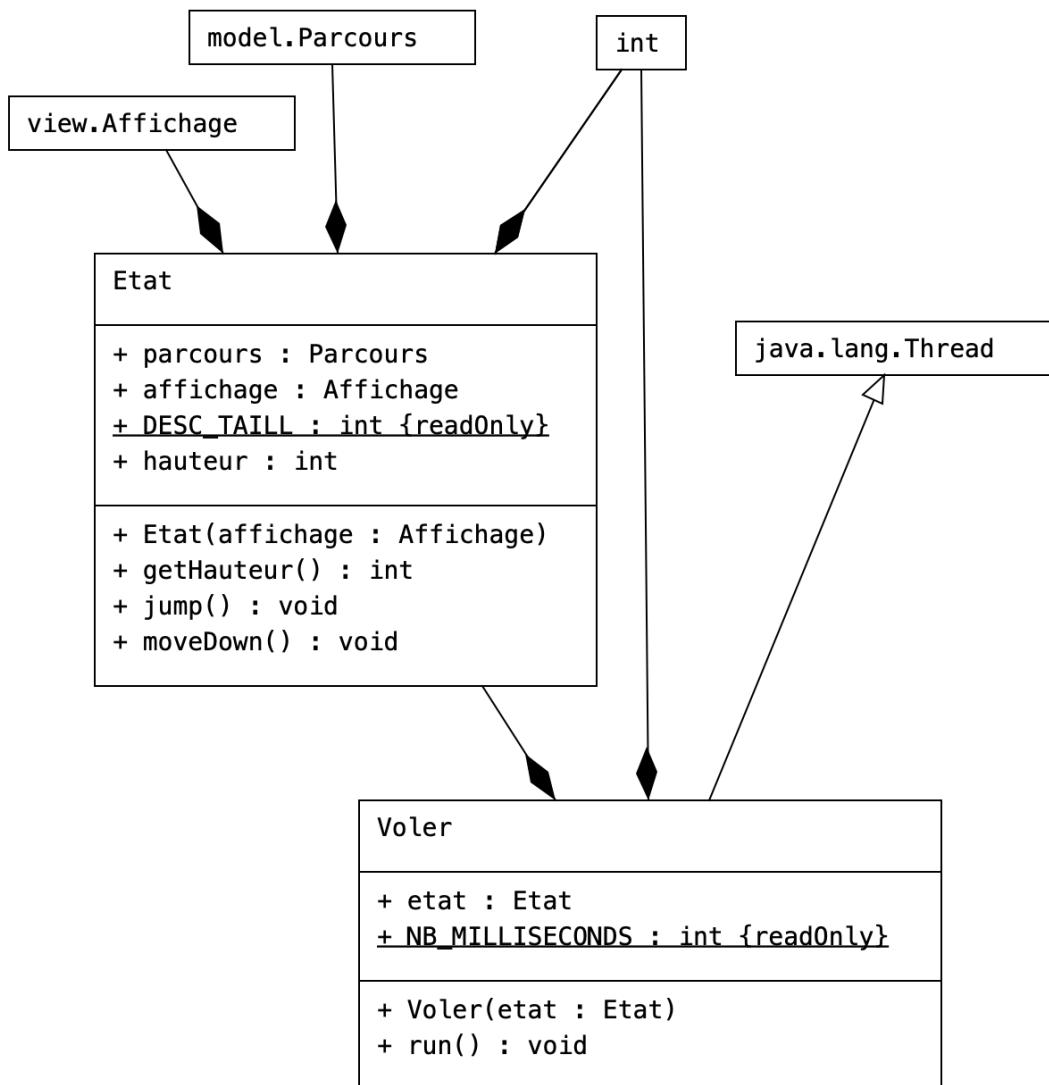
Tout d'abord on crée la classe FenetrePrincipale qui hérite de la classe JFrame et va nous servir à avoir notre fenêtre principale en appelant son constructeur avec le paramètre titre de la fenêtre. Ensuite on crée la classe Affichage qui hérite la classe JPanel et qui va nous servir à représentation de GUI du jeu. Ainsi on définit les constantes pour les dimensions de notre panel et ovale et la taille du saut.



On crée la classe Etat qui a comme l'attribut hauteur et les méthodes getHauteur pour retourner la hauteur et jump pour faire sauter l'ovale tout en vérifiant la borne du panel. Ainsi, on définit la classe Control qui hérite de la classe MouseController implémentant l'interface MouseListener pour faire sauter l'ovale lorsqu'on clique dans l'interface. Ceci est réalisé à travers de la méthode mouseClicked, en appelant la méthode jump.



On modifie la classe état en ajoutant une méthode moveDown qui permet de modifier la valeur de la hauteur de quelques pixels vers le bas, sans sortir de la zone de dessin. Ensuite on crée la classe Voler héritant de Thread qui fait redescendre progressivement la valeur de la hauteur, avec une pause de quelques millisecondes entre chaque chute. Ceci est réalisé depuis une boucle infinie pour appel à la méthode moveDown dans la méthode Run de ce thread.



Par la suite, on crée la classe Parcours ayant pour l'attribut principal une liste de point représentant la ligne brisée du jeu. On initialise cette liste à l'aide de valeurs aléatoires d'abscisse croissante de telle sorte que le dernier point est au delà de la bordure de l'écran. On définit le premier point de la ligne brisée de façon qu'il soit vers la bordure de droite du panel dans la zone visible pour donner un peu de temps à joueur pour qu'il s'adapte au jeu (faisant une analogie du jeu flappy bird). Ensuite on définit la méthode addNewPoint servant à la génération des points. L'algorithme de cette méthode est le suivant:

Approche 1: (Ceci est en commentaire dans le code)

Pour l'abscisse de chaque point, on considère la somme de position correspondant au score du joueur avec la constante POINT\_X\_MAX correspondant à une valeur au de-là de la bordure du panel, en l'occurrence 700. Le fait d'additionner POINT\_X\_MAX avec position nous permet de garantir l'ordre croissant d'abscisse parce que la valeur de position augmente progressivement. Pour l'ordonnée de chaque point, on tire une valeur aléatoire entre les limites de la hauteur du panel. Puis, afin de s'assurer que la vitesse de chute n'est pas inférieure à la valeur de pente de la ligne, on vérifie la pente de chaque ligne et si cette dernière dépasse la limite on la régularise. On calcule donc la pente de la ligne entre le dernier point existant dans la liste et les coordonnées prévu d'un point potentiel avec la formule suivante:  $a = (y_2 - y_1) / (x_2 - x_1)$  dans laquelle 'a' est la pente,  $(x_1, y_1)$  coordonnées du dernier point et  $(x_2, y_2)$  coordonnées prévues d'un point potentiel. Une fois la pente calculée, on vérifie si ça dépasse la limite des pentes définies à travers des constantes SLOPE\_MAX et SLOPE\_MIN. Si oui, en modifiant l'ordonnée ( $y_2$ ) de point potentiel on fait en sorte que la pente soit diminuée et enfin on crée le point ayant les coordonnées  $(x_2, y_2)$  et on l'ajoute à la liste des points. Éventuellement, en appelant à la méthode addNewPoint on initialise le deuxième point de liste.

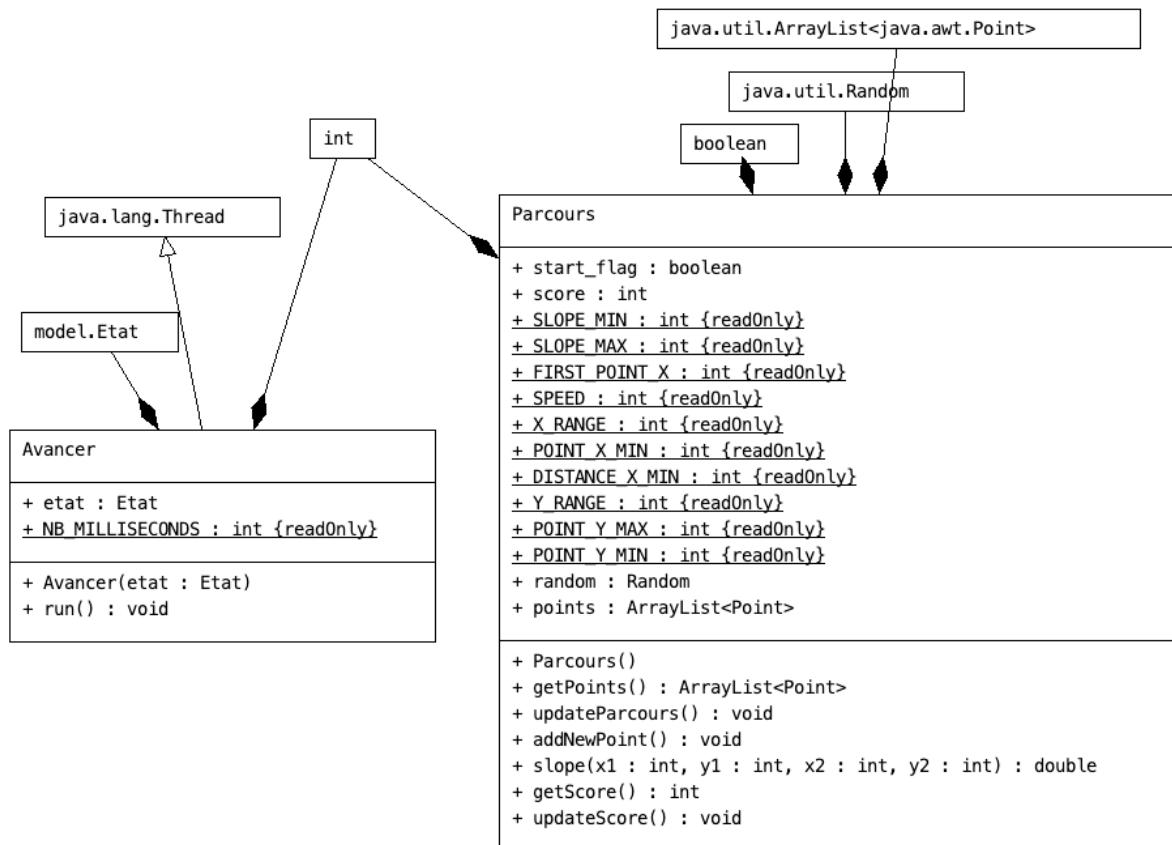
On modifie la méthode getParcours pour qu'elle ne renvoie que les points visibles, dont les coordonnées ont été calculées en retirant la valeur de position à leur valeur d'abscisse. L'algorithme est le suivant: dans un premier temps on crée une liste temporaire depuis la liste de point en prenant en compte la soustraction de la valeur de position à des valeurs de l'abscisse. Puis, on génère un point en appelant la méthode addNewPoint lorsque le dernier point rentre dans la fenêtre visible pour que la ligne brisée ne s'interrompe pas. Et lorsque le deuxième point sort de la zone visible, on retire le premier point de la liste, car il ne sera plus utilisé.

Approche 2: on pioche tout en respectant les limites, un nombre aléatoire pour l'abscisse et un autre pour l'ordonnée du point. Ensuite on vérifie la pente de la ligne engendrée par ce point et le dernier point existant dans la liste des points. Si la pente est jouable, on ajoute le point à la liste des points sinon on repioche deux autres nombres et continue cette boucle jusqu'à ce qu'on trouve le point avec la bonne pente.

Par la suite, on définit la méthode updateParcours qui a comme fonctionnalité de mettre à jour la liste de points en diminuant l'abscisse de chaque point de quelque pixel (montant constant SPEED). Ainsi, si le dernier point entre dans la zone visible, il ajoute un nouveau point à la liste et si le deuxième point sort de la zone visible, il supprime le premier point de la liste.

Enfin, on définit la méthode getScore pour récupérer la valeur courante du score du joueur et updateScore pour augmenter le score. On crée la classe Avancer héritant de la classe Thread permettant d'avancer la ligne brisée vers la gauche constamment et augmenter le score. Ceci est réalisé depuis une boucle pour appel à la méthode updateScore et updateParcours dans la méthode Run de ce thread.

On ajoute également la méthode drawLines pour l'affichage de la ligne brisée à la classe Affichage.



Dans cette dernière partie, on ajoute la méthode testPerdu à la classe Etat afin d'assurer la détection des collisions d'ovale avec la ligne brisée. L'implémentation de cette méthode est la suivante :

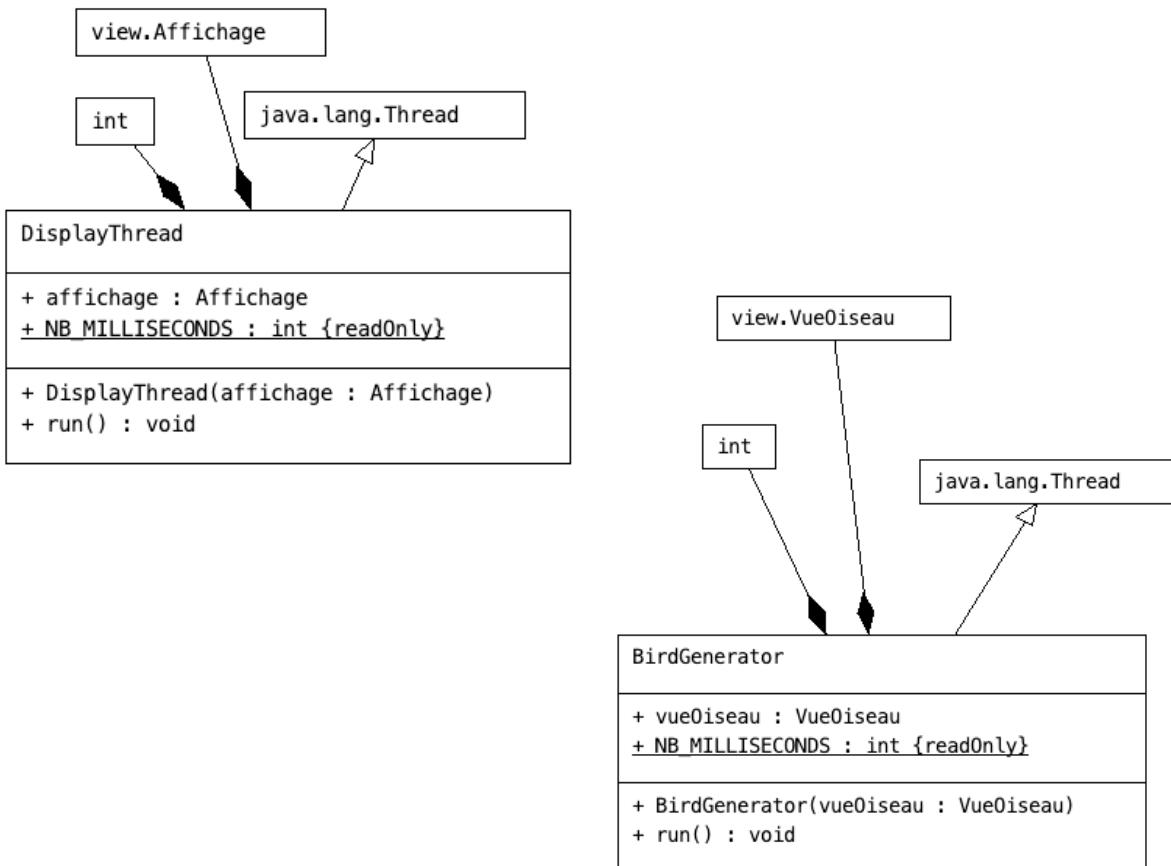
On prends le premier point de la liste et sachant qu'on laisse un peu de temps au joueur pour qu'il s'adapte à la vitesse du jeu, on renvoie faux (disant que le joueur n'est pas perdu) jusqu'à ce que le premier point arrive à l'ovale. Une fois le premier point arrivé à l'ovale, c'est en ce moment qu'on imagine une ligne droite verticale passant au milieu de l'ovale et on choisit deux points de la liste qui sont positionnés de façon que le premier soit avant et le deuxième et après la droite. Ceci est réalisé avec une boucle do while qui prend deux points consécutifs dès le début de la liste à chaque fois et vérifie s'ils satisfont la condition, si non il prend les deux points suivant jusqu'à ce qu'il trouve les deux bons points. Une fois, les deux points trouvés, on calcule la pente de la droite engendrée par ces deux points grâce à la formule  $a = (y_2 - y_1) / (x_2 - x_1)$  dans laquelle  $(x_1, y_1)$  c'est les coordonnées du premier point et  $(x_2, y_2)$  c'est les coordonnées du deuxième point. Une fois la pente trouvée, on met les coordonnées du premier point dans l'équation  $y_1 = ax_1 + b$  afin de trouver  $b$  qui est l'ordonnée à l'origine. Ensuite lorsqu'on a la pente et l'ordonnée à l'origine ( $b$ ), on peut écrire l'équation de la droite engendrée par les deux points. Finalement, en mettant dans cette équation la valeur d'abscisse d'ovale, on obtient la valeur  $y$  et on vérifie si cette valeur est dans l'ovale on renvoie faux sinon on renvoie vrai et le joueur est perdu.

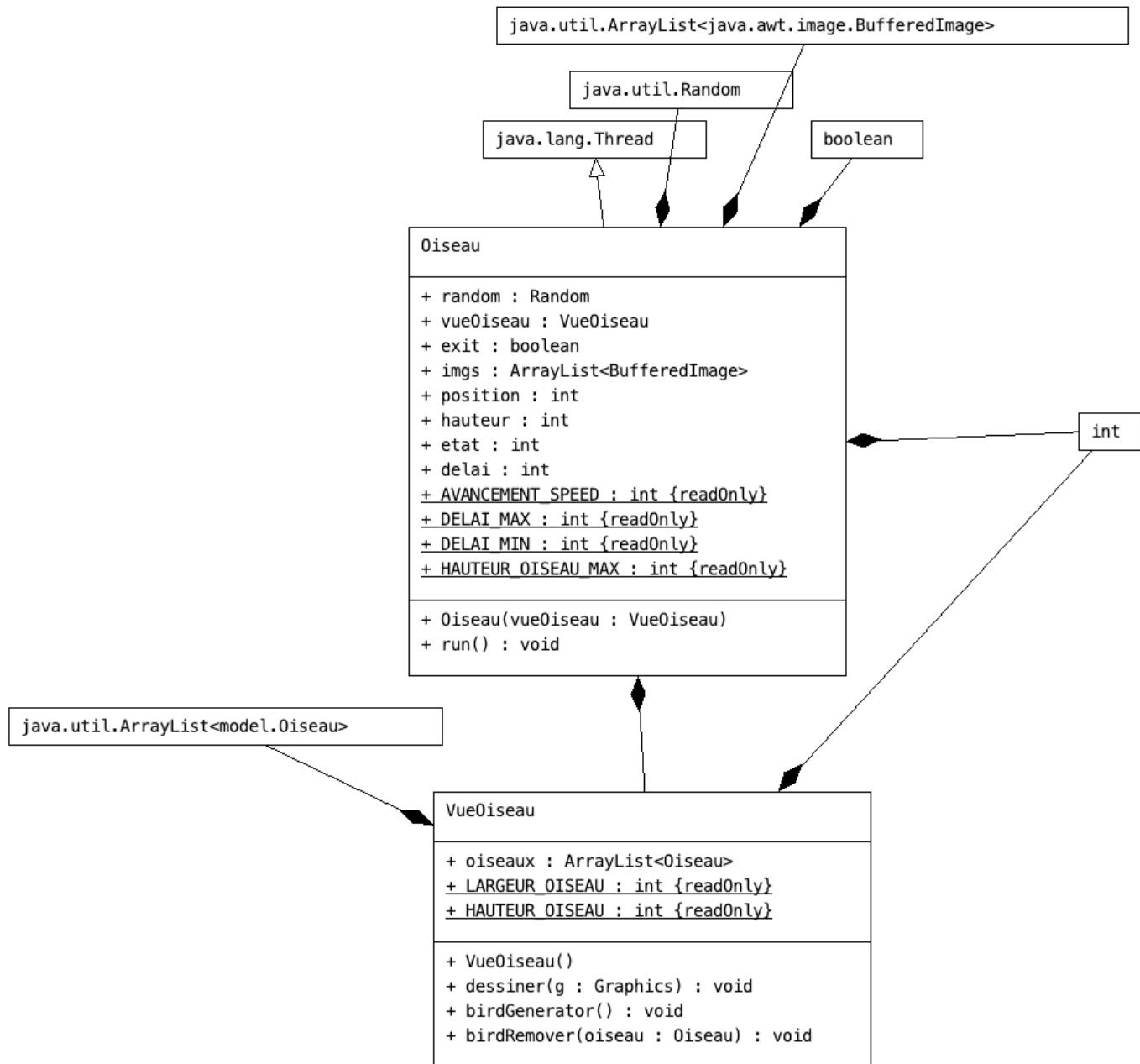
Par la suite, pour augmenter l'impression de défilement, on ajoute des oiseaux qui se déplacent en battant des ailes dans le décors. Pour cela, on crée la classe Oiseau héritant de la classe Thread qui représente un oiseau dans le jeu. Ce dernier munie de quatre attributs, délai qui indique le temps entre chaque mise à jour de l'affichage pour l'oiseau, etat qui permet de savoir dans quelle position est l'oiseau, hauteur qui définit la hauteur de l'oiseau dans la fenêtre graphique et position qui définit l'abscisse. Le constructeur de la classe choisit une valeur aléatoire pour le délai et la hauteur. La position est fixée de manière à ce que l'oiseau soit complètement à droite, au delà de la fenêtre visible. La méthode Run met à jour l'état et la position de l'oiseau toutes les délais millisecondes et elle s'arrête lorsque l'oiseau est complètement sorti de la zone visible.

Ensuite, on crée la classe VueOiseau qui possède un attribut de liste des oiseaux et qui possède une méthode dessiner(g) pour gérer l'affichage des oiseaux. Cette méthode met dans  $g$  l'image correspondant à l'état de l'oiseau, placée à la position courante, pour chaque élément de sa liste. Puis, on ajoute la méthode birdGenerator afin d'ajouter des nouveaux oiseaux au jeu. Ainsi on écrit la méthode birdRemover pour la suppression d'un oiseau. On modifie la classe Oiseau de façon que lorsqu'un oiseau sort de la zone visible, sera supprimé de la liste des oiseaux en appelant la méthode birdRemover.

Afin d'ajouter constamment les oiseaux au jeu, on crée la classe BirdGenerator héritant de la classe Thread. Ceci est réalisé depuis une boucle qui appelle constamment la méthode birdGenerator avec une pause de quelques millisecondes tout en assurant qu'il n'y a pas trop d'oiseaux en même temps dans le jeu.

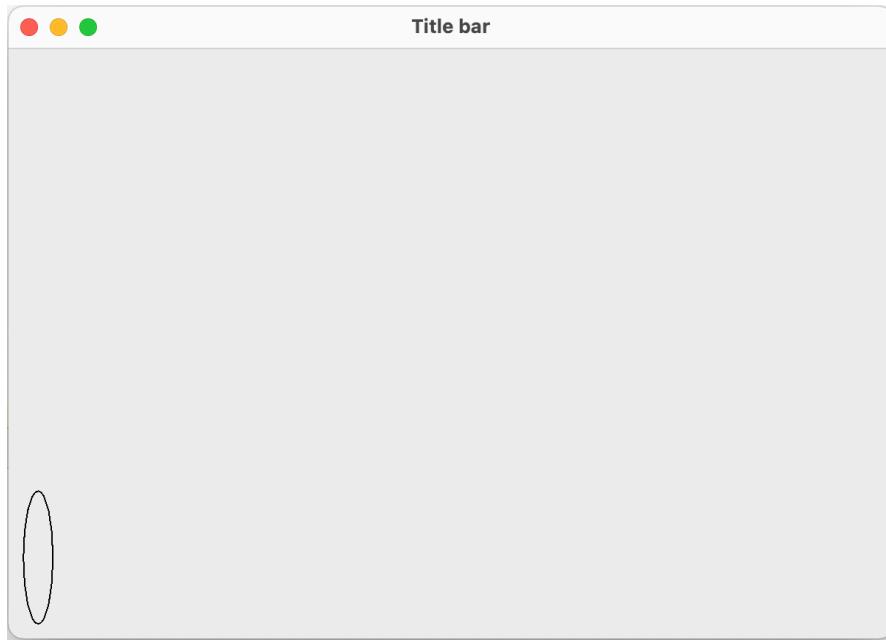
Finalement, on crée la classe `DisplayThread` qui est un thread permettant de mettre à jour l'affichage de l'interface graphique du jeu toutes les 40 millisecondes afin d'optimiser la performance et éviter que les autres threads appellent la méthode `repaint` lorsqu'ils effectuent des modifications dans le model. (Cela conduit naturellement à une surcharge au niveau de l'affichage, donc à des ralentissements, et donc à un visuel assez peu satisfaisant). Ainsi on modifie les autres threads de façon qu'ils devront simplement prévenir ce thread d'affichage (`DisplayThread`), et non plus directement la vue. La méthode `Run` de thread `DisplayThread` est réalisé depuis une boucle qui appelle la méthode `revalidate` et `repaint`.



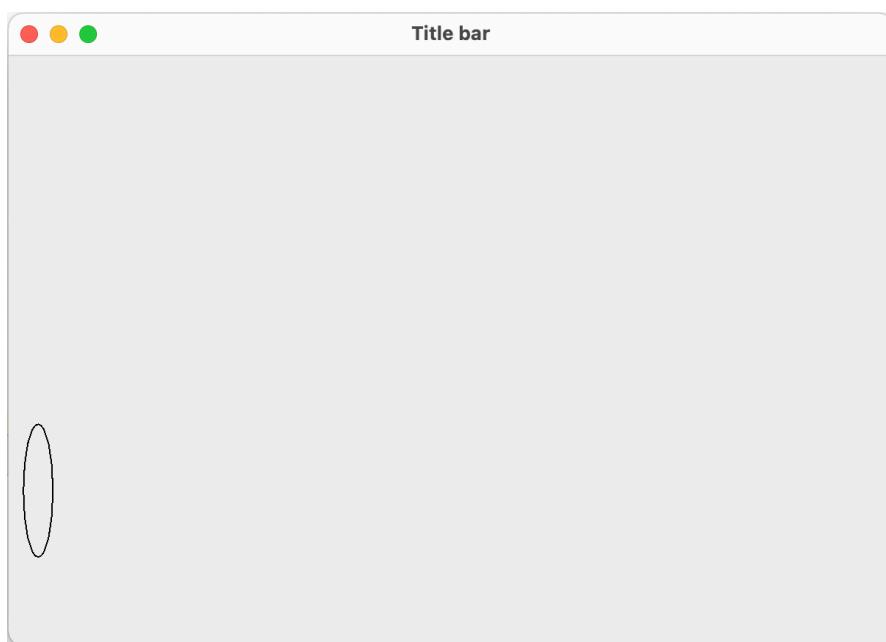


# Résultat

---



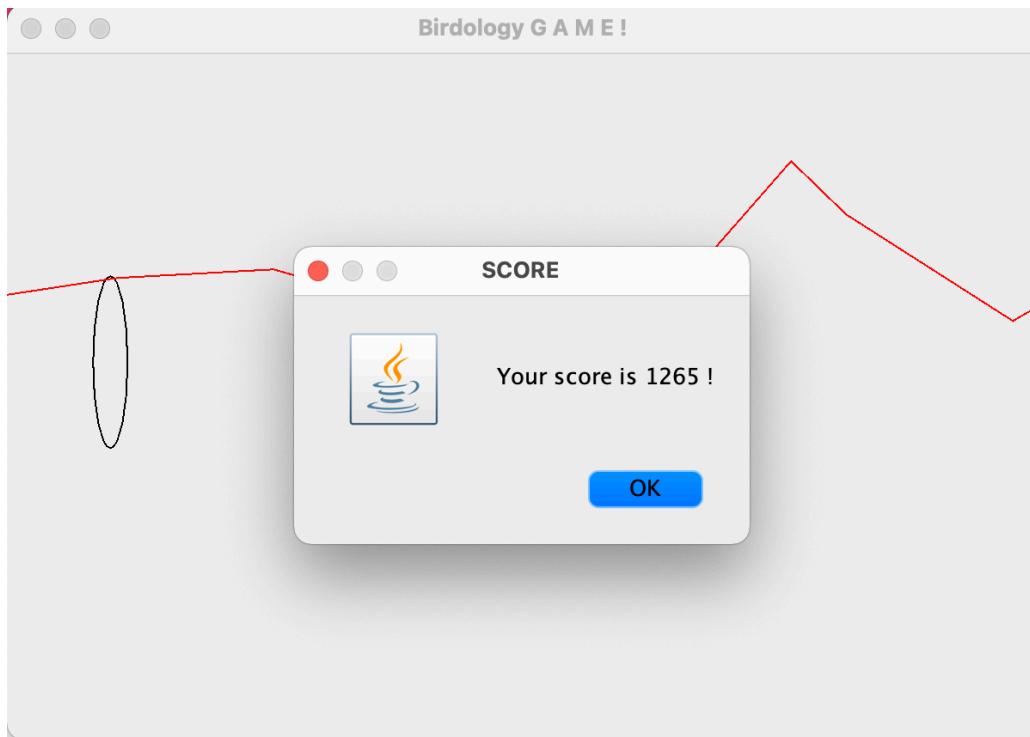
Réaction de l'évènement clique souris



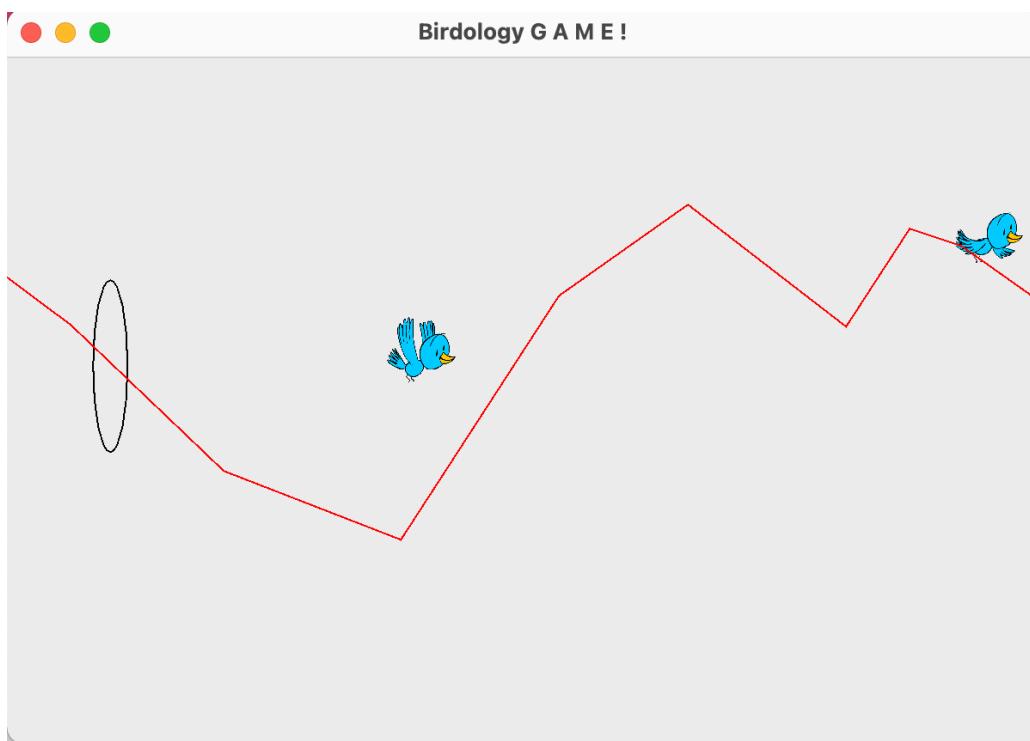
Déroulement de la ligne brisée et descente automatique de l'ovale :



Mécanisme de détection de collision :



Oiseaux de décoration:



# Documentation utilisateur

---

- **Prérequis :** Java avec un IDE (ou Java tout seul si vous avez fait un export en .jar exécutable)
- **Mode d'emploi (cas IDE) :** Importez le projet dans votre IDE, sélectionnez la classe Main à la racine du projet puis « Run as Java Application ». Cliquez sur la fenêtre pour faire monter l'ovale.
- **Mode d'emploi (cas .jar exécutable) :** double-cliquez sur l'icône du fichier .jar. Cliquez sur la fenêtre pour faire monter l'ovale.

## Documentation développeur

---

Les classes les plus importantes à regarder en premier sont la classe `Etat` qui définit l'ensemble des données caractérisant l'état de l'interface et la classe `Affichage` qui représente l'interface graphique du jeu. Plus, c'est la classe `Main` qui contient la méthode `main` et exécute le jeu.

Ainsi, les fonctionnalités qui ne sont pas encore implémentées sont dans un premier temps le calcul de bruit de Perlin en deux dimensions pour dessiner des ovales représentant des nuages avec une structure en utilisant les méthodes de la classe `Shape`. Par la suite, pour donner une meilleure impression du jeu au joueur, on rendra plus beau la présentation de l'ovale et la ligne brisée en utilisant la classe `Graphics2D`. Finalement, on créera le fichier exécutable `.jar` du jeu afin d'éviter de demander à l'utilisateur de lancer un IDE pour démarrer le jeu.

## Conclusion et perspectives

---

Jusqu'à présent, on a réussi à créer une fenêtre contenant un ovale qui descend constamment vers le bas et une ligne brisée infinie qui avance tout au long de la fenêtre de droit à gauche. La fenêtre du jeu réagit à l'événement clique souris et fait sauter l'ovale vers le haut lorsque le joueur clique sur cette dernière. Par ailleurs, on a implémenté un mécanisme de détection de collision lorsque l'ovale sort de la ligne brisée. Une fois l'ovale est sorti de la ligne brisée, par conséquent le jeu se termine en affichant le score du joueur à l'écran. Également, on a ajouté des oiseaux qui se déplacent en battant des ailes dans le jeu afin de donner mieux l'impression de défilement.

Les difficultés principales rencontrées consistaient dans un premier temps de l'utilisation de l'interface graphique de Java ensuite la manipulation de plusieurs threads. Ceci a été résolu d'abord grâce à énorme aide de professeur Nguyen et ainsi en lisant des ressources variées en particulier la documentation du Java sur le site d'Oracle et l'utilisation des différents forums.

Tout au long de cette période de réalisation du projet, j'ai eu l'opportunité d'acquérir beaucoup de nouvelle connaissance sur la programmation concurrente et aussi sur l'aspect graphique d'un programme. De plus, j'ai approfondie mes connaissances en programmation Java.

Pour finir, j'envisage de continuer à évoluer ce projet petit à petit tout en apprenant de nouvelles choses, et le présenter comme le premier jeu que j'ai développé dans ma carrière professionnelle et aussi l'utiliser en tant qu'un exemple sur ma chaîne YouTube pour enseigner la programmation aux gens.