

# **Rapport Sujet 1**

Simulation de pigeons dans un espace public

8INF957

Hamid Mcheick

10 November 2021

KLEQ07060000

VARA11029703

# Structure du Projet

Afin de répondre au mieux à la question, nous avons du considérer le projet et penser une architecture qui permettrait de résoudre au mieux la problématique posée en incluant de la programmation orientée objet, du multithreading et de la gestion d'erreurs.

De ce fait on peut distinguer plusieurs classes :

- La classe Board qui représente le plateau sur lequel les pigeons vont se déplacer.
- La classe Case : le board est une matrice composée d'objet Case.  
Ainsi une case est définie par sa position et les nourritures qu'elle contient ou non.
- La classe Constants : les différentes constantes qui permettent de définir la simulation.
- Les classes GameFrame et GamePanel : classes qui permettent de construire l'interface graphique et de représenter le board, les pigeons et leur comportement quand ils vont itérer sur l'espace de jeu.
- La classe Nourriture définit une nourriture qui sera placée sur le Board avec un Timer et un état de "fraîcheur".
- La classe Pigeon : définit un pigeon en tant que Runnable afin de traiter les différentes actions que celui-ci va effectuer.
- La classe Main : lance le jeu.

*Nb : Les captures d'écrans présente dans la suite de cette documentation peuvent être non représentative de la version finale du code soumis.*

# Board.java

La classe Board permet de définir notre espace des jeu. Un board est caractérisé par sa taille. De ce fait quand on crée une instance de cette classe un matrice de dimension 2 est créée de taille Size x Size. Chaque élément de cette matrice est un objet de type Case que l'on explicitera plus tard dans cette documentation.

## **Les méthodes de la classe Board:**

Les différentes surcharges de la méthode AddNourriture :

```
public void AddNourriture(int n) {
    // rien n'empêche d'avoir deux nourritures sur la même case
    for (int k = 0; k <n ; k++) {
        AddNourriture();
    }
}

public void AddNourriture(){
    Random randomGenerator = new Random();
    int i = randomGenerator.nextInt(Size);
    int j = randomGenerator.nextInt(Size);
    AddNourriture(i,j);
}

public void AddNourriture(int i, int j){
    if(Board.stream().flatMap(Collection::stream).anyMatch(c-> c.X==i && c.Y==j))
        try {
            GetCase(i,j).AddNourriture();
        }
        catch (Exception e) {
            System.out.println(e);
        }
}
```

La méthode AddNourriture permet d'ajouter de la nourriture sur le Board. La méthode de "base" permet de vérifier que la case d'indexes i et j

existe bien et d'ajouter de la nourriture sur cette case. Par la suite il nous a semblé judicieux de surcharger cette méthode pour ajouter une nourriture sur une case choisie aléatoirement et par la suite de surcharger encore cette méthode pour permettre l'ajout d'un nombre N de nourritures (cette méthode est notamment utilisé pour l'initialisation de notre grille de simulation).

La methode DeleteUnfreshedNourriture:

```
public void DeleteUnfreshedNourriture()
{
    Locker.lock();
    for (List<Case> row: Board) {
        for (Case aCase:row)
        {
            try {
                aCase.DeleteNotFreshedNourriture();
            }
            catch (Exception exception)
            {
                System.out.println(exception);
            }
        }
    }
    Locker.unlock();
}
```

Comme précisé dans le sujet, les pigeons ne mangent pas une nourriture qui n'est pas fraîche, de ce fait il semblait nécessaire de proposer à l'utilisateur la possibilité de pouvoir nettoyer la grille de simulation en élevant

les nourritures qui sont périmées. C'est le rôle que la méthode DeleteUnfreshedNourriture remplit. Ainsi on parcourt l'ensemble des cases de la matrice caractérisant le board afin de trouver celles comportant une ou plusieurs nourriture(s) non fraîche. On peut noter que l'on utilise un locker ici, en effet on veut accéder à une ressource critique pour écriture (notamment la suppression d'un ou plusieurs élément dans la collection de nourritures des instances de la classe Case) et on veut se prémunir de tout problème lié à l'accès de cette ressource par plusieurs Threads.

La méthode GetCase:

```
public Case GetCase(int i, int j)
{
    return Board
        .stream() Stream<List<Case>>
        .flatMap(Collection::stream) Stream<Case>
        .filter(c -> c.X==i && c.Y==j)
        .findFirst() Optional<Case>
        .orElse( other: null);
}
```

Permet l'accès de la case d'indexes i,j de la collection de cases Board.  
Si jamais aucun match n'est trouvé dans la collection on retourne un objet null.

## Autres méthodes:

```
public void DisplayFood()
{
    for(int i=0;i<Size;i++)
        for(int j=0;j<Size;j++)
            if(!GetCase(i,j).Nourritures.isEmpty()) {
                for ( Nourriture n: GetCase(i,j).Nourritures)
                {
                    System.out.printf("X = %s , Y = %s \n",i,j);
                    System.out.println(n);
                }
            }
}

public boolean IsEmpty()
{
    boolean isEmpty = true;
    for(List<Case> row : Board)
        for(Case element:row)
            if (element.HasNourriture()) {
                isEmpty = false;
                break;
            }
    return isEmpty;
}

public void DisplayBoard()
{
    for(List<Case> partition : Board) {
        for (Case c : partition)
            System.out.print(" | " + c.toString() + " | ");
        System.out.println("\n");
    }
}

public void DisplayBoard(String Nourriture)
{
    for(List<Case> partition : Board) {
        for (Case c : partition)
            System.out.print(" | " + c.toStringNourriture() + " | ");
        System.out.println("\n");
    }
}
```

Cette section contient les méthodes implémentées durant le processus de développement, utiles afin de comprendre le comportement de notre solution, mais qui n'ont plus été utilisées après l'implémentation de notre interface graphique :

- DisplayFood() : permet d'afficher dans la console les différents objets de type Nourriture présent dans notre matrice de Cases.
- IsEmpty() : permet de savoir si aucune nourriture n'est présente dans notre Board.
- DisplayBoard et sa surcharge : permet d'afficher le board avec deux affichages des cases possible.

# Case.java

## Les méthodes de la classe Case:

```
public int X;
public int Y;
public List<Nourriture> Nourritures = new ArrayList<>();
```

Un objet Nourriture est caractérisé de la manière suivante. Sa position X,Y correspondant à sa position dans la matrice Board et d'une liste d'instances de la classe Nourriture pouvant être vide.

## Les méthodes de suppression de nourritures:

```
public void DeleteNotFreshedNourriture()
{
    for (int i = 0; i < Nourritures.size(); i++) {
        if (!Nourritures.get(i).Fraiche)
        {
            Locker.lock();
            try {
                Nourritures.remove(i);
            }
            catch (Exception exception) {
                System.out.println(exception);
            }
            Locker.unlock();
        }
    }
}

public void DeleteNourriture(Nourriture n )
{
    try {
        Locker.lock();
        if (Objects.isNull(n)) {
            return;
        }
        Nourritures.remove(n);
        Locker.unlock();
    }
    catch (Exception e) {
        System.out.println(e);
    }
}
```

Les deux méthodes ci dessus permettent de supprimer des objets présent dans la collection Nourritures. Par ces méthodes, des Threads peuvent accéder à l'écriture de ressources critiques, ainsi il semblait primordial de gérer l'accès à la collection de Nourritures pour éviter des problèmes de concurrence :

- DeleteNotFresnedNourriture() : permet la suppression des nourritures périmées dans la collection de Nourritures d'une case. Cette méthode est notamment utilisé pour le nettoyage du Board proposé à l'utilisateur.
- DeleteNourriture(): Essaye de supprimer une Nourriture si l'instance en paramètres est non nulle. Sinon retourne une exception.

#### La methode AddNourriture:

```
public void AddNourriture(){  
    Locker.Lock();  
    Nourritures.add(new Nourriture(Constants.ValidityTime));  
    Locker.unlock();  
}
```

Permet l'ajout d'une nourriture dans la collection de nourriture.

#### La methode HasNourriture:

```
public boolean HasNourriture() { return !Nourritures.isEmpty(); }
```

Permet de savoir si la collection de nourritures de la case est vide.

## La méthode GetFresherNourriture:

```
public Nourriture GetFresherNourriture()
{
    if(Nourritures.isEmpty()) {
        return null;
    }
    return Nourritures.stream().min(Comparator.comparingInt(n -> n.PeremptionTimer)).orElse( other: null) ;
}
```

Retourne le nourriture la plus fraiche de la collection de nourriture triée, en utilisant un stream.

## Nourriture.java

```
class Nourriture {
    public int ValidityTime;
    public boolean Fraiche = true;
    private final Lock Locker = new ReentrantLock();
    public int PeremptionTimer = 0;

    public Nourriture(int ValidityTime) {
        this.ValidityTime = ValidityTime;
        Timer timer = new Timer();
        //Ici on incrémente le timer de peremption de 1 pour plus tard trier les nourritures par fraicheur
        //Si on dépasse la date de validité la nourriture est périmée.
        timer.scheduleAtFixedRate(() -> {

            try {
                Locker.lock();
                if (PeremptionTimer > ValidityTime) {
                    Fraiche = false;
                }
                PeremptionTimer++;
                Locker.unlock();
            }
            catch (Exception e) {
                System.out.println(e);
            }
        }, delay: 1, period: 1);
    }
}
```

Classe qui permet de définir une nourriture qui peut être consommée par un pigeon. À la création d'une instance de cette classe, on passe le paramètre ValidityTime qui est le temps avant qu'une nourriture soit périmée. Ainsi un timer est déclenché qui va incrémenté la variable PeremptionTimer qui symbolise la durée depuis la création de l'instance. Si cet attribut dépasse ValidityTime, la nourriture devient non fraîche via la redéfinition de la variable Fraiche.

## Pigeon.java

```
public class Pigeon implements Runnable {  
    public int Id;  
    public int X;  
    public int Y;  
    public Board Board;  
    private final Lock Locker = new ReentrantLock();  
    private final RandomGenerator generator;  
    public int FoodCount = 0;
```

Un pigeon est caractérisé par sa position en X et sa position en Y, un Id unique, une instance de Board sur lequel il va itérer et un compteur de nourriture mangées.

De plus chaque pigeon implémente l'interface Runnable, en effet on voulait que chaque pigeon soit représenté par un Thread et soit autonome.

### La méthode SearchFood:

```
public Case SearchFood()  
{  
    List<Case> cases = Board.Board  
        .stream() Stream<List<Case>>  
        .flatMap(Collection::stream) Stream<Case>  
        .filter(c->c.Nourritures.size()>0)  
        .toList();  
  
    Map<Case, Integer> temp = new HashMap<>();  
    cases.forEach(aCase -> {  
        if(!Objects.isNull(aCase.GetFresherNourriture())){  
            if (aCase.GetFresherNourriture().Fraiche)  
            {  
                temp.put(aCase, aCase.GetFresherNourriture().PeremptionTimer);  
            }  
        }  
    });  
  
    return (temp.size()>0) ? temp.entrySet().stream().min(Map.Entry.comparingByValue()).get().getKey() : null ;  
}
```

La préoccupation d'un pigeon est de se nourrir de nourritures fraîches.

Ainsi nous avons implémenté cette méthode afin d'effectuer une recherche de nourriture.

On commence tout d'abord par récupérer les cases qui comportent de la nourriture en utilisant un stream. Par la suite on se sert de cette collection nouvellement créée en intégrant dessus et en récupérant la nourriture la plus fraîche de chaque case. De ce fait on forme une HashMap ayant pour clé une case et pour valeur le compteur de péremption associé à la nourriture la plus fraîche de cette même case. Ainsi on peut alors retourner la case cible vers laquelle un pigeon va se déplacer (case comportant la nourriture la plus fraîche). Si jamais aucun nourriture fraîche n'est trouvé la case cible (target) sera une objet de type null.

### La methode Move:

```
public synchronized void Move()
{
    double moveProbability = BigDecimal.valueOf(generator.nextDouble())
        .setScale( newScale: 1, RoundingMode.HALF_UP)
        .doubleValue();
    if(moveProbability ==1) {
        Locker.lock();
        try {
            X = generator.nextInt(Constants.BoardSize);
            Y = generator.nextInt(Constants.BoardSize);
            System.out.printf("I moved to %s,%s because I've been afraid \n",X,Y);
        }
        catch (Exception ex) {
            System.out.println(ex);
        }
        Locker.unlock();
    }
}
```

Cette méthode permet de simuler le comportement aléatoire de dispersion d'un pigeon si il est effrayé avec un probabilité variable.

Cas 1

```
@Override
public void run() {
    while (true) {
        Case target = SearchFood();
        if (target == null) {
            try {
                System.out.printf("Waiting for %s milliseconds \n", Constants.PigeonWaitingTime);
                Move();
                Thread.sleep(Constants.PigeonWaitingTime);
            } catch (Exception e) {
                System.out.println("Thread interrupted");
            }
        }
        else {
            System.out.printf(Id+, I'm on case %s, %s \n, X, Y);
            System.out.println("La case " + target + " est celle contenant la nourriture la plus fraiche");
            boolean moved = false;
            try {
                if (target.X != X) {
                    X = (X - target.X) > 0 ? X - 1 : X + 1;
                    moved = true;
                    System.out.printf(Id+, Moving to %s,%s \n, X, Y);
                    Thread.sleep(Constants.PigeonSpeed);
                }
                if (target.Y != Y && !moved) {
                    Y = (Y - target.Y) > 0 ? Y - 1 : Y + 1;
                    System.out.printf(Id+, Moving to %s,%s \n, X, Y);
                    Thread.sleep(Constants.PigeonSpeed);
                }
            }
            catch (Exception exception) {
                System.out.println(exception);
            }
            if (target.X == X && target.Y == Y)// on est sur la bonne case
            {
                if (target.HasNourriture()) {
                    try {
                        Locker.lock();
                        if (!target.Nourritures.isEmpty()) {
                            if(target.Nourritures.get(0).Fraiche) {
                                target.DeleteNourriture(target.GetFresherNourriture());
                                FoodCount++;
                                System.out.printf("Id %s mangé la nourriture en %s %s \n", Id, X, Y);
                            }
                            else {
                                System.out.printf("Id %s : Je ne mange pas la nourriture pas fraiche en %s %s \n", Id, X, Y);
                            }
                        }
                        Locker.unlock();
                    }
                    catch (Exception exception) {
                        System.out.println("Nourriture déjà mangée");
                    }
                }
            }
        }
    }
}
```

Cas 2

Cas 3

Cette méthode est le cœur de la logique métier d'un pigeon (qui est un Thread rappelons-le). Un pigeon a deux comportements spécifiques : un comportement à l'arrêt si aucune nourriture fraîche est présente sur le board et un autre comportement si il détecte une nourriture fraîche sur le board (on peut même distinguer deux sous-comportements ici -> le pigeon n'est pas sur la case cible et le pigeon est sur la case cible). Ainsi pour l'explication de cette méthode nous nous pencherons sur chacun de ces 3 comportements.

Tout d'abord on fait appel à la méthode SearchFood, si la case cible est nulle (donc on a pas trouvé case contenant de la nourriture fraîche) le pigeon va attendre.

Le Thread se met en pause pendant un temps d'attente prédéfinis. On fait appel à la méthode Move pour simuler le changement de position si jamais un pigeon est effrayé.

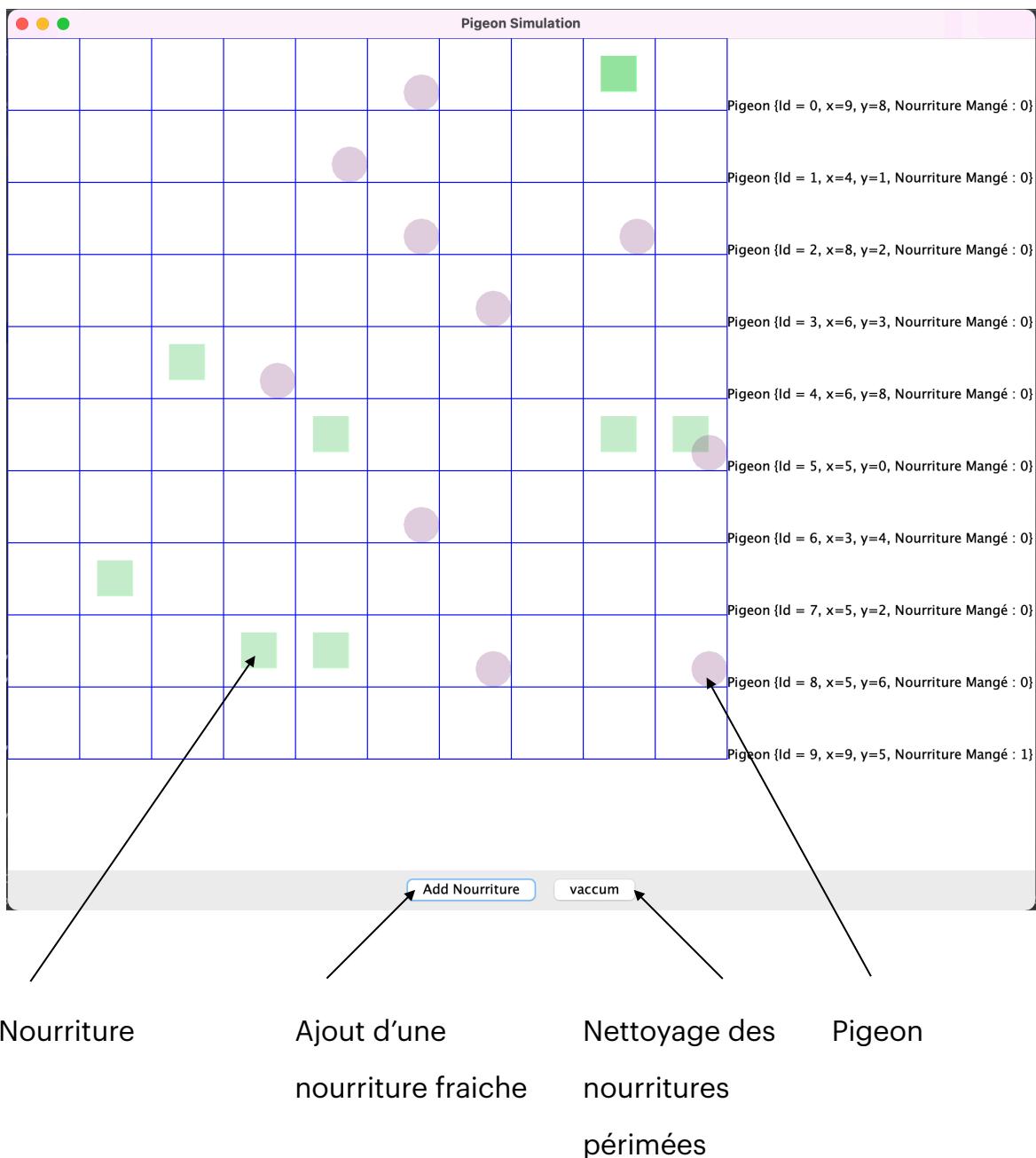
Si une case cible est trouvée, on a deux possibilités : le pigeon se trouve déjà sur la case cible ou le pigeon n'est pas sur la case cible.

Le pigeon n'est pas sur la case cible : Il déplace vers la case cible suivant le code si dessus (partie verte cas 2). Il se déplace en X d'une unité suivant si la case cible est situé à sa gauche ou à sa droite. Il se déplace en Y d'une unité suivant si la case cible est située en haut ou en bas de sa position actuelle. Si jamais il s'est déjà déplacé horizontalement, il ne va pas pouvoir se déplacer verticalement (on évite les déplacement diagonaux). Après cette opération le Thread se met en pause afin de simuler la vitesse de déplacement d'un pigeon défini préalablement.

Le pigeon est sur la case cible : Il n'a pas besoin de se déplacer (cf code ci dessus partie verte cas 3). Le Thread vérifie que la case contient toujours une nourriture -> un autre Thread a très bien pu la consommer juste avant lui. Le pigeon va vérifier que cette nourriture est toujours fraîche (peut être qu'à une milliseconde près elle s'est périmentée), et va la consommer en accédant à la case cible, en supprimant la nourriture la plus fraîche et en incrémentant son compteur de nourritures mangées. Ici il est indispensable de verrouiller l'accès à la case cible et de ce fait à la nourriture que notre pigeon veut consommer afin que deux thread ne puissent pas consommer deux fois la même nourriture.

## GameFrame.java/GamePanel.java

Ici on a les deux classes qui permettent de générer l'interface graphique suivante :



La classe GameFrame permet de définir les propriétés de la fenêtre graphique tel que le titre, la position, etc....

```
public class GameFrame extends JFrame {  
  
    public GameFrame() throws InterruptedException {  
  
        this.add(new GamePanel());  
        this.setTitle("Pigeon Simulation");  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        this.setResizable(true);  
        this.pack();  
        this.setVisible(true);  
        this.setLocationRelativeTo(null);  
    }  
}
```

Cependant c'est la classe GamePanel qui va permettre de dessiner notre plateau de jeu, nos boutons et les informations que l'on veut afficher à l'écran. Pour ce faire cette classe hérite de la classe JPanel permettant de construire une interface graphique. Voici les attributs que l'on a défini pour y arriver :

```
public class GamePanel extends JPanel implements ActionListener {  
  
    private Board Board;  
    private List<Pigeon> Pigeons;  
    private static int PanelSize = Constants.DisplaySize-300;  
    private static final int GAME_UNITS = Constants.BoardSize;  
    private static final int UNIT_SIZE = PanelSize/GAME_UNITS ;  
    private static final int DELAY = 10;  
    private static boolean Running = false;  
    private Timer Timer ;  
    private PausableExecutor executor;
```

Ainsi le GamePanel se charge de représenter le board et les pigeons qui vont le parcourir, la taille de la grille en fonction d'une constante définie et le nombre de case à dessiner qui en découle. Le DELAY correspond au temps de rafraîchissement de la grille graphique.

Le constructeur de cette classe permet de créer les différentes objets, la taille du Panel, les couleurs qui y sont reliées, les boutons et les actions que vont générer l'interaction avec l'utilisateur.

```
GamePanel() throws InterruptedException {

    this.Board = CreateInitializeBoard();
    this.Pigeons=CreatePigeon(Board);
    System.out.println(Pigeons.size());
    JPanel panel = new JPanel();
    this.setPreferredSize(new Dimension(Constants.DisplaySize,Constants.DisplaySize));
    this.setBackground(Color.WHITE);
    this.setForeground(Color.BLUE);
    JButton add_nourriture = new JButton( text: "Add Nourriture");
    JButton vaccum = new JButton( text: "vaccum");
    add_nourriture.addActionListener(e->Board.AddNourriture());
    vaccum.addActionListener(e-> Board.DeleteUnrefreshedNourriture());
    panel.add(add_nourriture);
    panel.add(vaccum);
    setLayout(new BorderLayout());
    add(panel, BorderLayout.SOUTH);
    StartGame();
}
```

On remarque qu'on a deux fonctions d'initialisation des objets qui sont appelées dans le constructeur. Les voici ci-dessous :

```
private static List<Pigeon> CreatePigeon(Board board)
{
    List<Pigeon> res = new ArrayList<>();
    for(int i=0;i<Constants.NombrePigeon;i++)
    {
        Random generator = new Random();
        int k = generator.nextInt(Constants.BoardSize);
        int v = generator.nextInt(Constants.BoardSize);
        res.add(new Pigeon(i,k,v,board));
    }
    return res;
}

private static Board CreateInitializeBoard() {
    Board board = new Board(Constants.BoardSize);
    board.AddNourriture(Constants.InitialNourriture);
    return board;
}
```

CreateInitializeBoard(): Permet de créer un objet de type Board et de placer aléatoirement le nombre de nourritures définis en constante.

CreationPigeon(): Comme nous avons déjà créer notre Board on peut maintenant créer les pigeons avec cette méthode. On génère un nombre N de pigeons et on leur donne des positions aléatoire.

Lancement de la simulation:

```
private void StartGame()
{
    Running = true;
    Timer = new Timer(DELAY, listener: this);
    Timer.start();
    executor = new PausableExecutor();
    Pigeons.forEach(executor::execute);
}
```

On utilise un ThreadPool personnalisé qui va permettre aux Thread Pigeon de s'exécuter. L'intérêt de cet exécuteur est de pouvoir mettre les thread en pause lors du dessin de l'interface graphique.

Dessin de la grille :

```
private void draw(Graphics g)
{
    executor.pause();

    for(int i=0; i<=GAME_UNITS;i++) {
        g.drawLine( x1: i * UNIT_SIZE, y1: 0, x2: i * UNIT_SIZE,PanelSize );
        g.drawLine( x1: 0, y1: i * UNIT_SIZE,PanelSize, y2: i * UNIT_SIZE );
    }

    DrawNourriture(g);

    g.setColor(new Color( r: 100, g: 0, b: 100, a: 50));
    Pigeons.forEach(p->g.fillRect(
        x: (p.X)*(PanelSize/GAME_UNITS)+UNIT_SIZE/2,
        y: (p.Y)*(PanelSize/GAME_UNITS)+UNIT_SIZE/2, width: UNIT_SIZE/2, height: UNIT_SIZE/2));

    g.setColor(Color.BLACK);
    int count = PanelSize/Pigeons.size();
    for(Pigeon pigeon : Pigeons)
    {
        g.drawString( pigeon.toString(),PanelSize,count);
        count+= PanelSize/Pigeons.size();
    }
    executor.resume();
}
```

```

private void DrawNourriture(Graphics g)
{
    for(List<Case> row : Board.Board){
        for(Case c : row)
            for(Nourriture n : c.Nourritures)
            {
                if (n.Fraiche) {
                    g.setColor(new Color( r: 0, g: 255, b: 0, a: 50));
                } else {
                    g.setColor(new Color( r: 255, g: 0, b: 0, a: 50));
                }
                g.fill3DRect(
                    x: (c.X)*(PanelSize/GAME_UNITS)+UNIT_SIZE/4,
                    y: (c.Y)*(PanelSize/GAME_UNITS)+UNIT_SIZE/4, width: UNIT_SIZE/2, height: UNIT_SIZE/2, raised: false);
            }
    }
}

```

Ici on voit bien notre exécuteur personnalisé nous permet de mettre nos Pigeons en pause durant le processus de dessin.

Le principe de dessin de la grille est relativement le même pour toutes nos entités. On adapte les positions de nos éléments relativement à la grille et on les dessine avec un certaine dimension.

- Dessin de la grille
- Dessin des pigeons
- Dessin de la partie droite de l'interface graphique ( informations sur les pigeons)

Le méthode DrawNourriture permet de différencier la couleur en fonction que la nourriture soit périmée ou non.

## Autres Classes

La classe Constants :

```
public class Constants {  
    static final int ValidityTime = 2000;  
    static final int NombrePigeon = 10;  
    static final int BoardSize = 10;  
    static final int InitialNourriture = 10;  
    static final int PigeonSpeed = 1000; //Millisecond  
    static final int PigeonWaitingTime = 1000;  
    static final int DisplayerSize = 1000;  
  
}
```

C'est ici qu'on sont définis les différents constantes essentielles à la simulation : DisplayerSize -> taille de la fenêtre graphique générée  
ValidityTime -> Nombre de millisecondes avant péremption d'une nourriture

La classe Main :

Classe à définir pour la compilation et l'exécution de la solution.

```
public class Main {  
  
    public static void main(String[] args) throws InterruptedException{  
        new GameFrame();  
    }  
}
```

## La classe PausableExecutor :

```
class PausableExecutor extends ThreadPoolExecutor {
    //custom executor pour mettre en pause les thread lorsque l'on dessine l'interface graphique
    private boolean isPaused = false;
    private Lock pauseLock = new ReentrantLock();
    private Condition unpause = pauseLock.newCondition();

    public PausableExecutor() {
        super(Constants.NombrePigeon, Constants.NombrePigeon, keepAliveTime: 1, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<>());
    }

    public void pause() {
        //mettre en pause les thread
        pauseLock.lock();
        try {
            isPaused = true;
        } finally {
            pauseLock.unlock();
        }
    }

    public void resume() {
        //relancer les thread
        pauseLock.lock();
        try {
            isPaused = false;
            unpause.signal();
        } finally {
            pauseLock.unlock();
        }
    }
}
```

Hérite de la classe ThreadPoolExecutor. Permet de créer un exécuteur personnalisé afin de pouvoir mettre en pause les thread rattachés à ce ThreadPool (dans notre cas ce sont des instances de la classe Pigeon)