

PCII - Projet de groupe

Raccooking

1 - Cahier des charges :

Nous souhaitons réaliser un jeu monojoueur en temps réel qui consiste à gérer le stock de pains d'une boulangerie attaquée par de vilains raton-laveurs.

Pour cela, le boulanger devra faire cuire ses pains et empêcher les raton-laveurs de les atteindre en se déplaçant sur la carte. Il pourra ainsi les vendre et racheter de quoi faire d'autres pains.

Pour gagner, il devra atteindre un objectif de richesse fixé avant la fin du temps imparti.

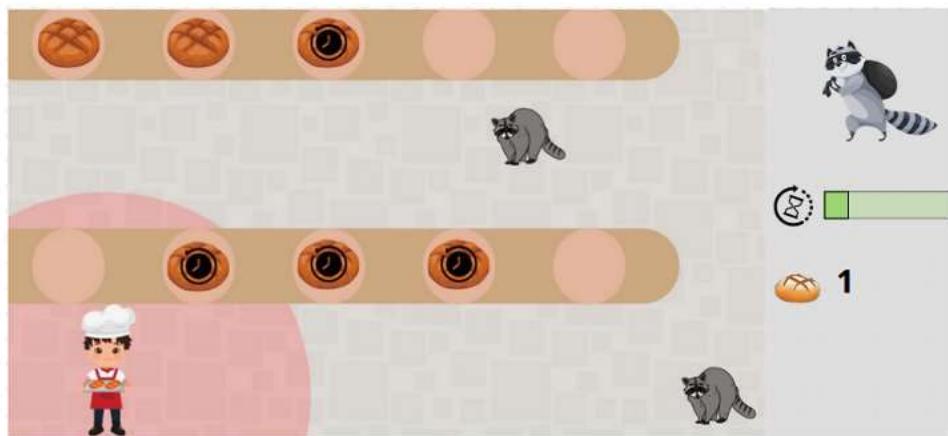
Le joueur peut se déplacer sur la carte pour faire fuir les rats laveurs qui se déplacent aléatoirement. Il peut récupérer les pains qui sont cuits à travers un panneau de contrôle et les vendre pour gagner de l'argent, qui lui servira à acheter des ingrédients et ainsi préparer d'autres pains. En cliquant sur le joueur, son panneau de contrôle apparaît.

En cliquant sur un raton-laveur, on peut voir le temps qu'il reste avant qu'il s'en aille et le nombre de pains qu'il a volé.

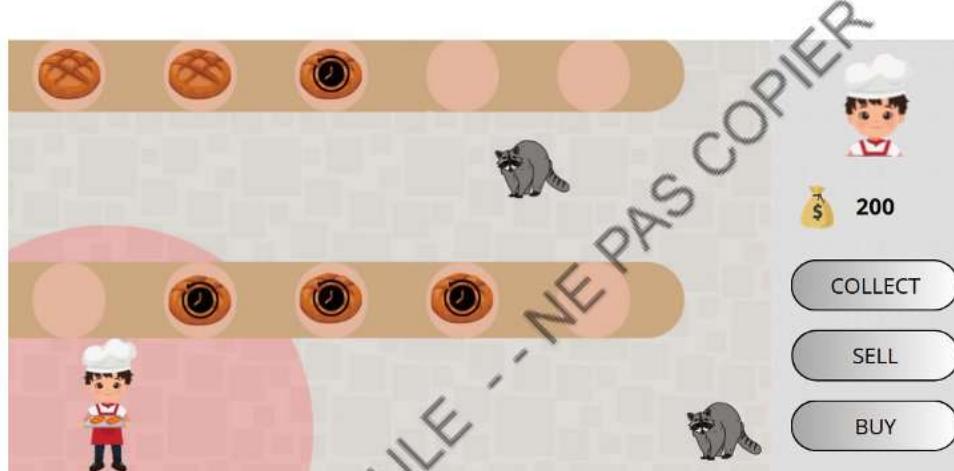
Ci-dessous, un dessin prototype de notre jeu :



Lorsqu'on clique sur un raton-laveur :



Lorsqu'on clique sur le boulanger :



2 - Analyse :

Pour réaliser ce projet, il est nécessaire de définir dans un premier temps les différentes fonctionnalités à implémenter. Pour cela, il va leur être donné un indice de priorité et de difficulté sur 5 (avec 5 le plus haut score). Ainsi, nous nous retrouvons avec la liste de fonctionnalités suivante :

1. Implémenter une carte du jeu
 - Priorité : 5
 - Difficulté : 1

Implémenter la carte du jeu n'est pas vraiment compliqué, et assez rapide à faire. Néanmoins, son indice de priorité est maximal, car elle est nécessaire aux fonctionnalités principales requises de ce projet (le déplacement des unités) ainsi que pour placer nos ressources.

2. Déplacement du joueur
 - Priorité : 5

- Difficulté : 2

Le déplacement du joueur est une fonctionnalité requise du cahier des charges de notre projet. Il n'est pas très difficile à implémenter, du moment qu'on code correctement les directions.

3. Dynamique du pain

- Priorité : 5
- Difficulté : 2

La dynamique de notre ressource - ici, du pain - n'est pas très compliquée à réaliser, mais néanmoins un élément essentiel du jeu, requis dans le cahier des charges.

4. Déplacement des ratons-laveurs

- Priorité : 5
- Difficulté : 2

Nécessaire pour établir la structure de base du jeu. Ce n'est pas très complexe, mais il était important de bien gérer la sélection automatique de la prochaine position d'un raton laveur.

5. Gestion de la vie des ratons-laveurs:

- Priorité : 3
- Difficulté : 1

Permet de rajouter de la fluidité au jeu, avec des ratons laveurs qui disparaissent et apparaissent.

6. Gestion de la boulangerie

- Priorité : 4
- Difficulté : 4

C'est la pièce maîtresse du jeu donc elle est très importante. Cependant, elle rassemble plein de classes et est intimement liée à l'affichage, ce qui rend la tâche difficile.

7. Affichage de la boulangerie et ses composants avec interaction

- Priorité : 5
- Difficulté : 4

Pour pouvoir afficher la boulangerie avec des composantes interactives, il faut gérer plusieurs interactions à l'écran, qui vont nous permettre de le mettre à jour : cela peut être un peu compliqué à implémenter. Par ailleurs, cette fonctionnalité est très importante car elle constitue l'interface du notre jeu.

8. Panneau de contrôle du boulanger

- Priorité : 4
- Difficulté : 2

La bonne organisation des panneaux de contrôle nécessite de gérer des JPanel dans des JPanel ce qui rend la tâche un peu complexe. Par ailleurs, ce panneau de

contrôle est très important car il permet la gestion des ressources du boulanger (vente , achat ...)

9. Panneau de contrôle du raton-laveur

- Priorité : 2
- Difficulté : 1

Le panneau de contrôle du raton-laveur nécessite de gérer la barre de progression qui représente la vie du raton-laveur via un thread. Ce panneau de contrôle n'est pas très important car il n'est présent que pour donner des informations liées au raton-laveur à titre indicatif.

10. Chasser les raccoons

- Priorité : 3
- Difficulté : 3

Ici la difficulté est de trouver une manière élégante de régler ce problème. Comme le joueur contrôle le mouvement du boulanger, on ne peut pas prédire comment celui-ci va bouger, s'il y aura des rats-laveurs sur le chemin ou non. Il faut donc trouver un moyen pas trop compliqué, de faire fuir les rats-laveur si possible, sans déclencher des erreurs avec son mouvement courant.

11. Ecran de démarrage

- Priorité : 3
- Difficulté : 2

Un écran de démarrage permet de rajouter de l'esthétique au jeu et aussi d'apporter justement ce côté "jeu". Il n'est pas très compliqué à faire mais nécessite de gérer plusieurs éléments permettant de lancer le jeu.

12. Plusieurs niveaux

- Priorité : 1
- Difficulté : 2

Il s'agit là d'une extension permettant de donner un aspect plus "jeu" à notre projet, et donc peu prioritaire par rapport au reste. Elle n'est pas non plus très difficile à implémenter.

13. Musique de fond

- Priorité : 1
- Difficulté : 2

Avoir une petite musique de fond permet de rendre la partie plus "amusante" et de donner l'impression de vraiment être dans un jeu. Evidemment, il s'agit là d'une extension tout à fait optionnelle. Néanmoins, il peut être compliqué d'introduire une musique en boucle qui ne se relance pas à chaque nouvelle partie.

14. Fin de jeu

- Priorité : 3

- Difficulté : 2

Enfin, tout bon jeu doit bien se finir. Pour cela, il est nécessaire d'implémenter un objectif au jeu avec une condition et un écran de fin, bien que moins prioritaire que les fonctionnalités principales du jeu. Implémenter une fin de jeu n'est pas très compliqué car il suffit simplement de récupérer quelques éléments de la partie.

3 - Plan de développement :

Une fois le cahier des charges analysé, il faut séparer chaque fonctionnalité en tâches à réaliser. Cela nous donne donc les tâches suivantes :

1. Implémenter une carte du jeu :

- a. Créer des cases
- b. Classe "carte" contenant les composants de la carte

2. Déplacement du joueur :

- a. Classe Baker correspondant au joueur
- b. Gestion des touches du clavier
- c. Déplacement du Baker sur la carte

3. Dynamique du pain :

- a. Classe BakedGoods et sous classes
- b. Gestion de la cuisson

4. Déplacement des ratons-laveurs :

- a. Classe Raccoon
- b. Trouver le pain cuit le plus près
- c. Fonction déplacement aléatoire sur une case voisine
- d. Déplacement du Raccoon sur la carte
- e. Animation de déplacement d'une case à l'autre

5. Gestion de la vie des ratons-laveurs:

- a. Fonction de suppression et remplacement dans le tableau
- b. Thread de gestion du tableau des ratons-laveurs

6. Gestion de la boulangerie

- a. Achat des ingrédients nécessaires à la fabrication des pains
- b. Ajout des pains
- c. Vente des pains

7. Affichage de la boulangerie et ses composants

- a. Dessiner une fenêtre
- b. Dessiner les pains et les fours
- c. Dessiner les raton laveurs
- d. Dessiner le joueur

- e. Implémenter un thread pour redessiner l'affichage entre ses différentes mises à jour

8. Panneau de contrôle du boulanger

- a. Dessiner un JPanel avec les éléments prévus tel que compteur et boutons
- b. Ajouter le panneau de contrôle à l'affichage
- c. Implémenter un thread pour mettre à jour le contenu du JPanel

9. Panneau de contrôle du raton-laveur

- a. Dessiner un JPanel avec les éléments prévus tel que le nombre de pains volés et son temps de vie restant
- b. Implémenter un thread pour mettre à jour le nombre de pains volés et la barre de progression qui représente le temps écoulé.

10. Chasser les raccoons

- a. Etablir un périmètre de cases autour du joueur qui fait fuir les rats-laveurs
- b. Représenter visuellement le périmètre pour le joueur

11. Ecran de démarrage

- a. Ajouter une image de fond, un bouton pour démarrer le jeu et un autre pour gérer le son
- b. Dessiner un JPanel pour choisir le niveau
- c. Implémenter un Thread pour gérer l'animation du choix des niveaux

12. Plusieurs niveaux

- a. Dessiner les cartes dans des fichiers txt
- b. Lire le fichier du niveau pour initialiser la carte

13. Musique de fond

- a. Lancer une musique en boucle au démarrage du jeu
- b. Arrêter la musique à la fin de la partie

14. Fin de jeu

- a. Implémenter un Thread pour gérer l'écoulement du temps et la fin du jeu
- b. JFrame pour l'écran de fin avec possibilité de relancer le jeu
- c. Gérer la victoire ou perte de la partie

Par ailleurs, nous avons également des tâches qui sont utiles et nécessaires dans la conception de notre projet, mais n'ont rien à voir avec le projet en lui-même / l'analyse, que nous allons réaliser tout au long de notre projet :

15. Réaliser une documentation complète du projet : il s'agit de ce document !

- a. Pré-analyse et cahier des charges : cette sous-tâche est séparée du reste, car elle est réalisée en brainstorming au tout début de notre projet avant de se consacrer au reste.

16. Diagramme de Gantt pour suivre le développement du projet au fil des semaines : un extrait est visible un peu plus bas, et le diagramme complet est disponible en pièce jointe.

17. Nettoyage du code : une tâche récurrente tout au long du projet qui prend suffisamment de temps pour mériter sa propre ligne dans le diagramme



Extrait du diagramme de Gantt de ce projet. Pour la version complète, voir le lien ci-dessous :

https://centralesupelec-my.sharepoint.com/:g/personal/mathilde_needham_etu-upsaclay_fr/ETdWC4SFijFFoXN003vmoNYBzz5VwaE7ikBpETIHRIcYq?rtime=M3JIFpF73Ug

4 - Conception :

Dans le cadre de ce projet, nous allons concevoir notre projet avec l'**architecture Modèle-Vue-Contrôleur**. De ce fait, nous allons répartir nos fichiers selon 3 packages : Model, View et Controller.

Pour notre fichier **Main**, nous le séparons du reste de l'architecture MVC.

Le package **Model** contient toutes les données de notre projet qui caractérisent l'état du système : il contient donc :

- *Bakery*, la “carte de notre jeu” dans lequel sont stockées les composantes notre carte (représentée par un tableau 2D de cases) : notre joueur, les rats-laveurs et les fours dans lesquels peuvent cuire nos pains
 - *Tile*, correspondant à une case de la carte
 - *Oven*, qui hérite de *Tile* et correspond à une case non-traversable contenant un pain (un four donc)

- *Entity*, une classe abstraite représentant les entités de notre jeu : elle contient une case (*Tile*) représentant sa position actuelle et une fonction abstraite *move()* permettant de se déplacer
- *Baker*, la classe représentant notre joueur et héritant de *Entity*
- *Raccoon*, qui correspond aux ratons-laveurs et hérite d'*Entity*
- *BakedGoods*, la classe dont héritent nos pains et autres viennoiseries, qui gère leurs 3 états : COOKING, COOKED et BURNT, et qui appelle la recette correspondante au pain dont il est question
- *Bread*, la classe représentant des pains et qui hérite de *BakedGoods*
- *Brioche*, la classe représentant des brioches et qui hérite de *BakedGoods*
- *Croissant*, la classe représentant des croissants et qui hérite de *BakedGoods*
- *Cooking*, le Thread qui gère la cuisson des pains en fonction du temps

Le package **View** contient les fichiers relatifs à l'interface visible du projet. Ainsi, il est composé de :

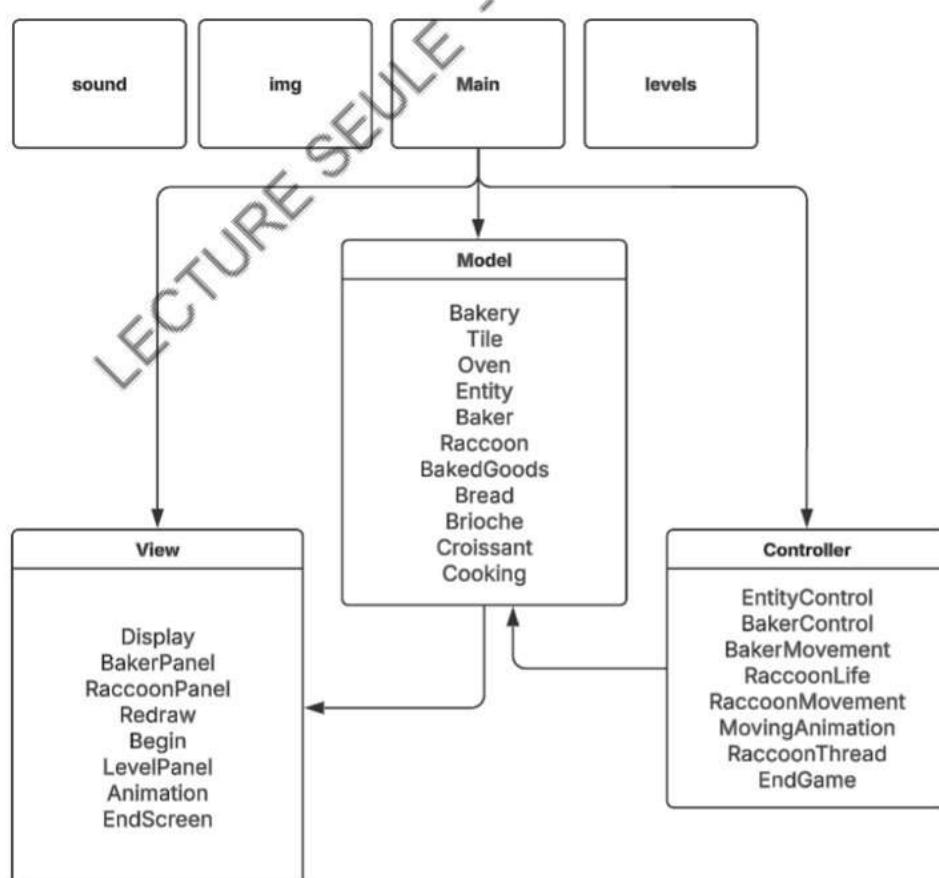
- *Display*, le JFrame correspondant à notre affichage général
- *Redraw*, le thread qui redessine l'interface graphique pour fluidifier les animations
- *BakerPanel*, l'écran de contrôle du joueur qui s'affiche lorsqu'on clique sur le personnage du boulanger
- *RaccoonPanel*, l'écran de contrôle du raton-laveur qui s'affiche lorsqu'on clique sur le personnage du raton-laveur
- *Redraw*, le thread permettant de redessiner notre affichage mis à jour à intervalles réguliers
- *Begin*, le JFrame correspondant à l'écran d'accueil
- *LevelPanel*, le panneau animé de l'écran d'accueil permettant de choisir le niveau de jeu
- *Animation*, le thread qui contrôle l'animation de choix de niveaux
- *EndScreen*, l'écran de fin de partie, qui permet de quitter le jeu ou bien de relancer une nouvelle partie

Enfin, le package **Controller** contient les fichiers relatifs à la capture des événements dans l'interface qui modifient le modèle. Ici, notre package contient un unique fichier :

- *EntityControl*, qui implémente l'interface *MouseListener* et qui appelle la méthode *mouseClicked()* pour afficher le panneau de contrôle de l'entité sur laquelle on clique, c'est-à-dire un raton-laveur ou le boulanger
- *BakerControl*, qui implémente l'interface *ActionListener* et qui appelle la méthode *actionPerformed()*, qui gère les actions lorsqu'on clique sur un bouton du panneau de contrôle du boulanger

- *BakerMovement*, qui implémente l'interface KeyListener et qui appelle la fonction move() de Baker dans la direction indiquée pour se déplacer sur la carte
- *RaccoonLife*, le thread qui regarde si des ratons-laveurs quittent la boulangerie, c'est-à-dire si leur attribut age dépasse la constante MAX_AGE : si c'est le cas, les remplacent par de nouveaux ratons-laveurs
- *RaccoonMovement*, le thread qui gère le mouvements des ratons-laveurs, en appelant move() pour chacun des raton-laveur après avoir trouvé sa prochaine position
- *MovingAnimation*, le thread qui, entre chaque déplacement de raton-laveur, gère un déplacement fluide
- *RaccoonThread* , le thread qui gère la mise à jour de la barre de progression présente dans le panel du raton-laveur : elle représente le temps écoulé avant que celui-ci ne quitte la boulangerie et du nombre de croissant , pain , brioche volés
- *EndGame*, le thread qui gère le temps d'une partie et y met fin lorsque celle-ci se termine

Par ailleurs, les images et icônes sont stockés dans un dossier **img**, les fichiers de niveaux se trouvent dans un dossier **levels** et le fichier audio de la musique de jeu se situe dans un dossier **sound**.



Représentation des différents packages et dossiers de notre projet

A présent, nous allons détailler un peu plus la conception de chaque fonctionnalité, en reprenant celles que nous avons définies dans l'analyse :

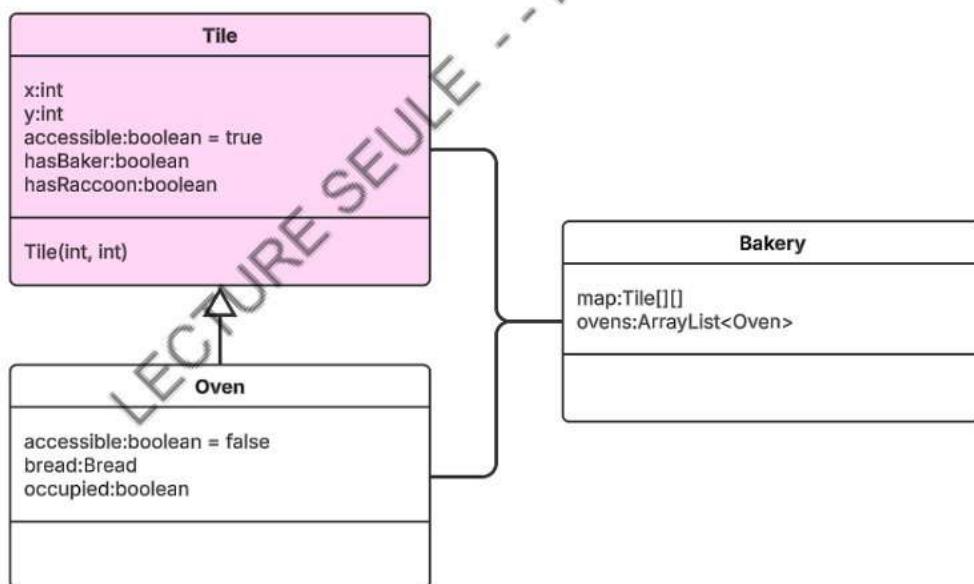
1. Implémenter une carte du jeu

La classe *Bakery* représente notre carte du jeu, elle en rassemble toutes les composantes. Dans son constructeur on initialise le boulanger, les rats-laveurs et les fours.

Notre carte est définie par un tableau de tableau de cases, représentées par la classe *Tile*. Celle-ci a une sous-classe *Oven*, pour implémenter les fours qui peuvent se trouver sur la carte, créant ainsi des cases non-traversables pour les entités du jeu.

Ce tableau est initialisé dans le constructeur de *Bakery* et montre l'état courant du jeu : où sont les rats-laveurs, où se situe le boulanger et quels fours sont occupés par des pains en train de cuire.

Pour plus facilement retrouver les fours par la suite, les cases *Oven* sont aussi stockées dans un tableau séparé.



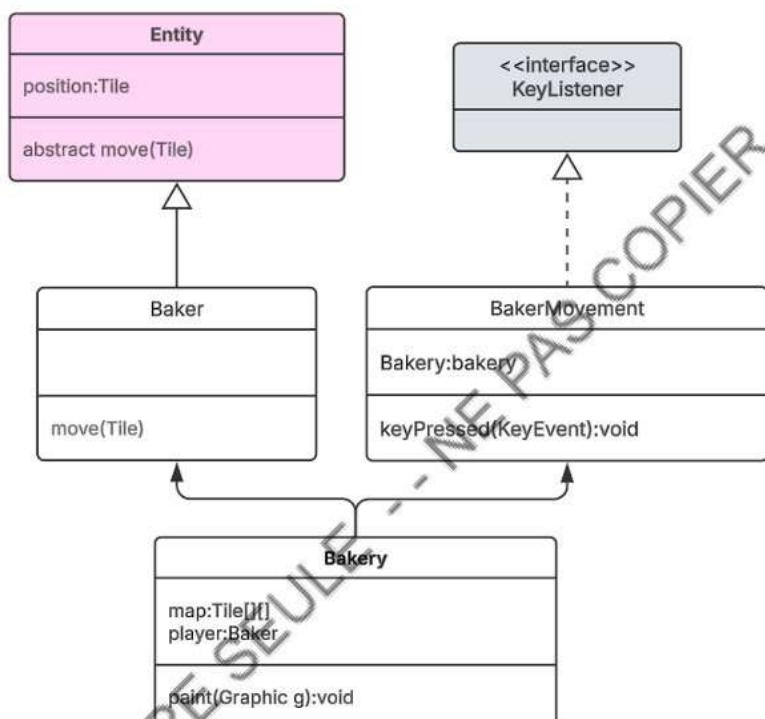
2. Déplacement du joueur

Pour implémenter le déplacement d'une unité jouable, nous créons dans un premier temps une classe *Baker*, qui représente notre joueur. Nous allons la faire hériter d'une classe mère *Entité*, qui sera utile plus tard pour nos rats-laveurs.

Pour le déplacement, une classe *BakerMovement* implémentant l'interface *KeyListener* permet de se déplacer en fonction de touches pressées au clavier :

- en haut lorsqu'on clique sur Z ou flèche du haut
- en bas lorsqu'on clique sur S ou flèche du bas
- à gauche lorsqu'on clique sur Q ou flèche gauche
- à droite lorsqu'on clique sur D ou flèche droite

Une fois la touche récupérée (grâce aux keycodes Java), on vérifie que la case cible est bien dans les bornes de la carte puis on appelle la fonction *move()* de *Baker*. La fonction *move()* vérifie que la case est traversable et, si c'est le cas, met à jour la position du joueur.



Ci-dessous, l'algorithme du déplacement du joueur :

BakerMovement.keyPressed() :

Entrée : KeyEvent

Sortie : rien

Tile pos <- position du joueur

Si flèche haut ou Z et qu'on n'est pas tout en haut :

Baker.move(case du haut)

Si flèche bas ou S et qu'on n'est pas tout en bas :

Baker.move(case du bas)

Si flèche gauche ou Q et qu'on n'est pas tout à gauche :

Baker.move(case à gauche)

Si flèche droite ou D et qu'on n'est pas tout à droite :

Baker.move(case à droite)

Baker.move() :

Entrée : Tile

Sortie : rien

On vérifie si la case cible est accessible. Si c'est le cas :

On retire le boulanger de sa case actuelle (récupérée avec position)

Baker.position <- case cible

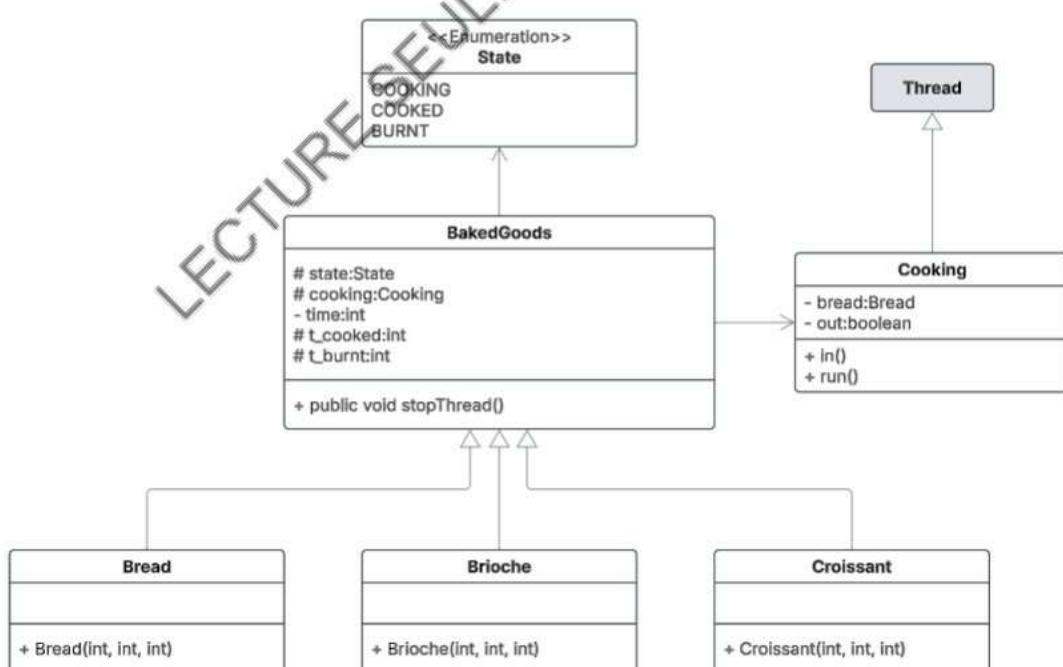
On place le boulanger sur la case cible

3. Dynamique du pain

Pour gérer la dynamique du pain, nous devons d'abord implémenter une classe qui reprend les attributs communs à tous les types de ressources. Nous l'appellerons *BakedGoods*. Chaque type de ressource héritera de cette classe. Nous avons décidé de mettre du pain, des croissants et des brioches. Chacun a un prix de vente différent, une recette différente et des températures de cuisson différentes.

Une ressource a un état représenté par le type énuméré *State*, qui est une classe interne à la classe *BakedGoods*. Nous avons ainsi trois états : 'COOKING', 'COOKED' et 'BURNT'.

La classe *BakedGoods* contient également deux attributs 't_cooked' et 't_burnt' qui représentent respectivement le temps au bout duquel une ressource est cuite ou brûlée. La durée de vie de chaque ressource est représentée par l'attribut 'time' de *BakedGoods*.



Lorsqu'on crée une ressource, on appelle d'abord le constructeur de la classe *BakedGoods* qui prend en argument le prix, le temps de cuisson et le temps au bout

duquel la ressource brûle. Le constructeur initialise l'attribut time, le state et va également créer et lancer un Thread de la classe *Cooking*.

Il possède un attribut 'out', initialisé à false, qui sera changé à true lorsque le pain sera récupéré afin de sortir de la boucle du thread. Celui-ci prendra en argument le pain lui-même. Sa fonction run() permettra d'incrémenter l'attribut 'time' du pain de la ressource ainsi que de changer son *State* au moment opportun selon l'algorithme suivant :

Cooking.run() :

Entrée, Sortie : rien

Tant que time<= T_BURNT et out == false :

Si l'attribut time du pain est égal à la constante T_COOKED de pain :

on set l'attribut state à COOKED

Sinon, si l'attribut time du pain est égal à la constante T_BURNT de pain :

on set l'attribut state à BURNT

On incrémente l'attribut time

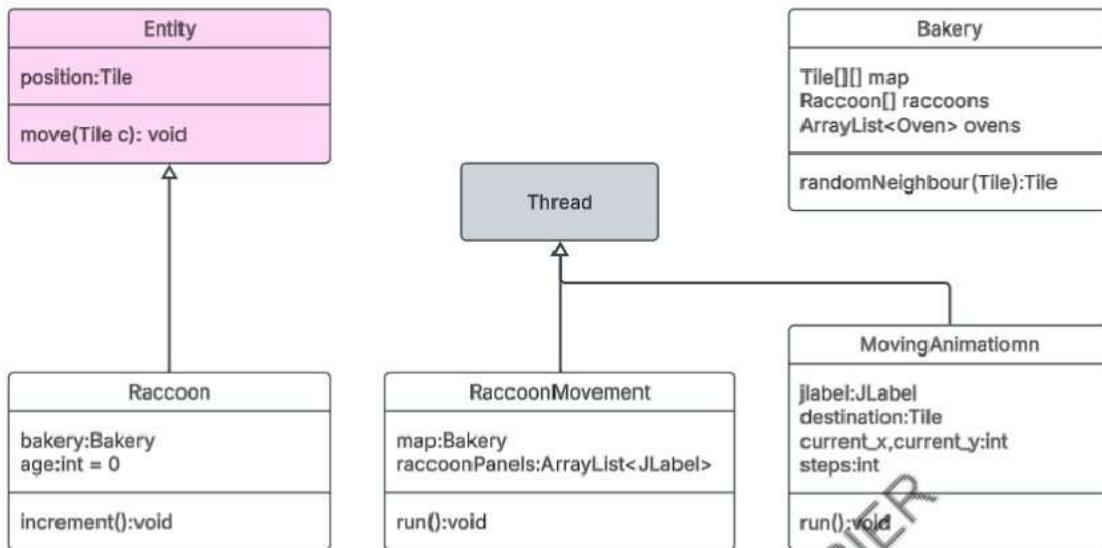
4. Déplacement des rats-laveurs

Comme pour le joueur, on crée une classe pour les rats-laveurs *Raccoon* qui hérite de *Entity*, et qui assure qu'ils ont une position et une implémentation de la fonction move(). Dans la fonction nextMove(), le raton-laveur choisit sa prochaine position en fonction de l'état courant du jeu, qu'il accède grâce à son attribut 'bakery'.

Par ailleurs, un thread *RaccoonMovement*, traverse le tableau de rats-laveurs actuellement dans notre boulangerie, et, pour chacun d'entre eux, appelle la fonction move() prenant en argument la case renvoyée par nextMove(). Il va aussi lancer un thread *MovingAnimation* pour chaque mouvement d'une case à une autre qui permet de donner un mouvement plus fluide au raccoon.

Dans ce thread, on calcule de combien le *JLabel* doit avancer pour arriver à destination en 'steps' étapes; dans le run() on a une boucle qui va à chaque fois ajouter un peu à ses coordonnées 'current_x' et 'current_y', et déplacer le *JLabel* grâce à la fonction setBounds().

On a accès dans *RaccoonMovements* au tableau de *JLabel* grâce à l'attribut 'raccoonLabels' de *Display*, qui est donné en argument lors sa création.



Ci-dessous l'algorithme implémentant le déplacement des rats-laveurs:

Raccoon.nextMove():

Entrée: rien

Sortie: Une case (classe *Tile*)

Si il est en fuite

 Trouver une case voisine qui ne soit pas à côté du boulanger

 Si aucune on renvoie position courante

Sinon si aucun pain n'est cuit:

 On choisit un case voisine et libre aléatoirement et on se déplace

Sinon, on trouve le pain cuit le plus proche:

 Si on est devant le four avec le pain cuit:

 Raton laveur mange le pain

 Sinon :

 On regarde dans quelle direction il faut aller, et on se déplace

RaccoonMovement.run():

Entrée, Sortie : rien

On effectue en continu :

 On récupère le tableau de rats-laveurs de la carte

 Pour chaque raton-laveur :

 Si son âge est < AGE_MAX :

 On incrémente son âge de 1

 On lance un nouveau Moving Animation thread

 On récupère son nextMove() et on se déplace sur la case cible

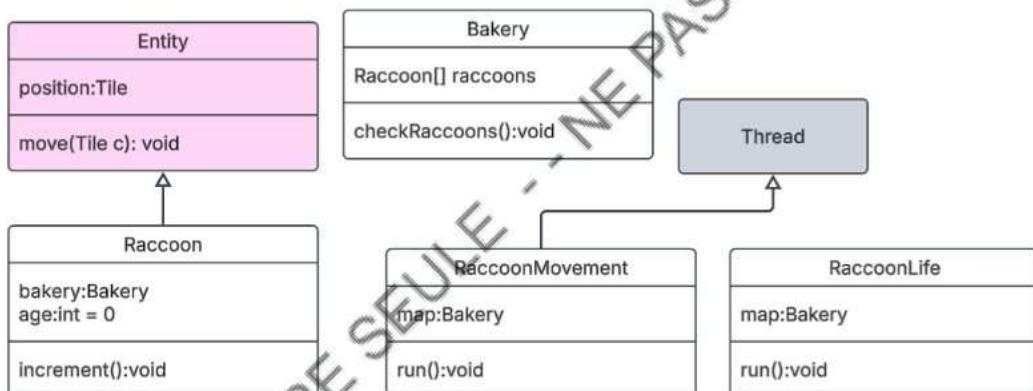
5. Gestion de la vie des ratons-laveurs

Dans notre classe *Raccoon*, on ajoute un attribut ‘age’ pour chaque instance de la classe, qui va être initialisé à zéro dans le constructeur, et une constante ‘MAX_AGE’ qui permet de définir à partir de quand le raton-laveur quitte la boulangerie.

A chaque fois qu’un raton-laveur bouge, le Thread *RaccoonMovement* incrémentera son âge de 1.

Ensuite, il nous faut gérer quand les rats laveurs quittent la boulangerie. Pour cela, on a un nouveau thread *RaccoonLife*. Ce thread va simplement appeler la fonction *checkRaccoons()* de *Bakery* qui va passer à travers notre tableau de rats laveurs dans la boulangerie : si l’un d’entre eux dépasse l’âge limite, il est supprimé et remplacé par un nouveau raton laveur qui apparaît en bas à droite de la boulangerie.

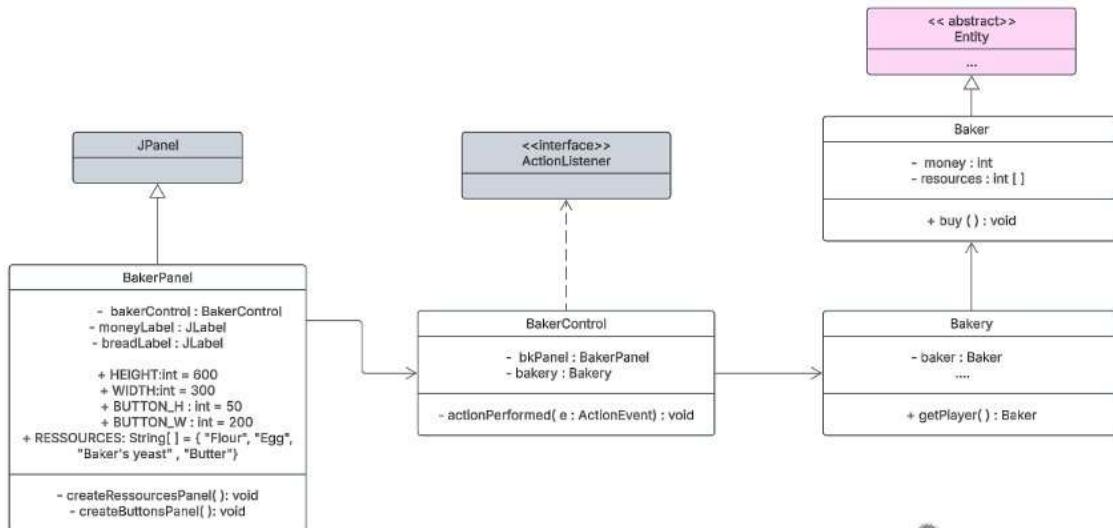
Enfin, on a un troisième thread *RaccoonThread* qui vérifie que la barre de progression qui apparaît lorsqu’on clique sur un raton laveurs soit bien à jour.



6. Gestion de la boulangerie

a. Achat des ingrédients nécessaires à la fabrication des pains

Pour acheter de nouveaux ingrédients , il faut cliquer sur le bouton “Buy” du panneau de contrôle du boulanger. Une boîte dialogue s'affiche alors, avec les différents ingrédients qu'on peut acheter : farine, œufs, levure, beurre. Il y a la possibilité d'en choisir plusieurs en même temps. L'achat de ses ingrédients dépend également de la richesse du boulanger, cette contrainte est gérée par la méthode *buy* de la classe *Baker*. Les prix unitaires des ingrédients sont également affichés dans la boîte de dialogue.



Ci-dessous, l'algorithme correspondant à l'achat d'une ou plusieurs ressources :

Cas “Buy” de BakerControl.actionPerformed() :

Entrée : ActionEvent.getActionCommand()

Sortie : rien

Un nouveau JPanel apparaît avec des checkbox pour chaque ingrédient

Si on clique sur le bouton OK :

On vérifie les checkbox cochées

On appelle Baker.buy() sur chaque ingrédient sélectionné

Si on clique sur Cancel :

Ferme le JPanel

Baker.buy :

Entrée : String s

Sortie : void

Si on a suffisamment d'argent donc > 2€ :

Switch case sur s :

on ajoute 1 à l'indice correspondant à s dans le tableau des ressources

on décrémente Baker.money de 2

b. Ajout des ressources

Pour ajouter une ressource, il faut cliquer sur l'icône correspondante dans le panel du joueur. Ceci déclenche l'appel de la méthode bake de la classe *BakerControl* avec en paramètre “Bread”, “Brioche” ou “Croissant” en fonction du bouton cliqué.

BakerControl.bake:

Entrée: String type

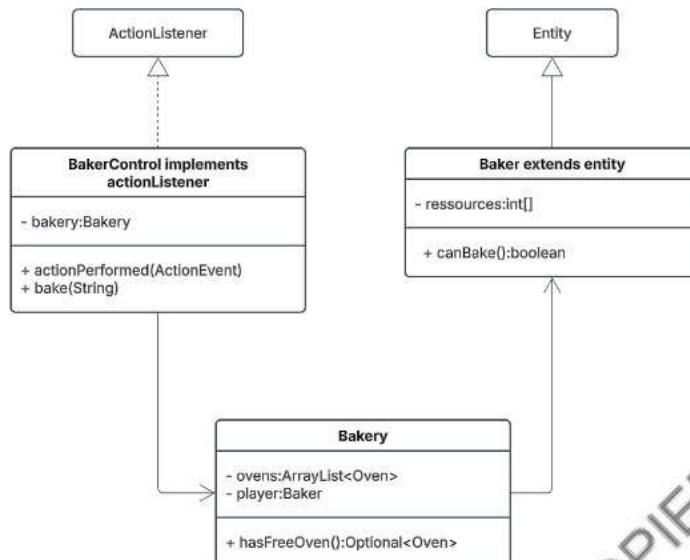
Sortie: rien

On récupère la recette du type passé en argument

Si le boulanger a assez de ressources pour la recette :

si la boulangerie contient un Oven vide :

**on y ajoute un nouveau pain
on retire les ressources nécessaires au boulanger**



Pour déterminer s'il y a un *Oven* libre, nous utilisons la méthode `hasFreeOven()` de la classe *Bakery* qui suit l'algorithme suivant :

Bakery.hasFreeOven:

Entrée: rien

Sortie: `Optional<Oven>`

Pour chaque oven contenu dans l'ArrayList ovens de Bakery :

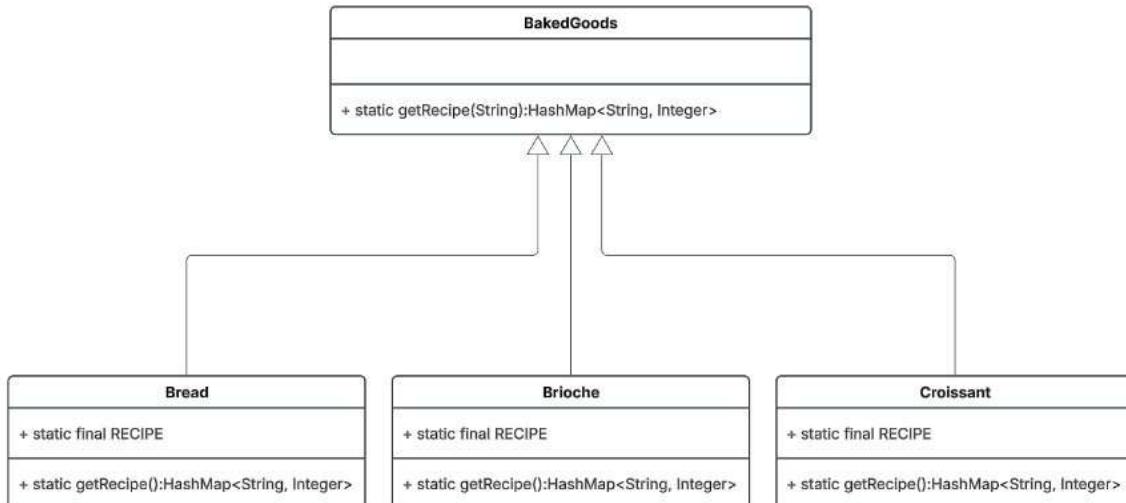
Si l'oven n'est pas occupé :

Renvoie un Optional de l'oven

Renvoie empty

Cette méthode permet donc à la fois de déterminer s'il y a un *Oven* libre et de le récupérer.

Les recettes sont stockées sous forme de table de hachage qui associe un *String* à un *int*. Pour récupérer la recette, nous utilisons la méthode `getRecipe()` de *BakedGoods*. Celle-ci récupère la constante 'RECIPE' de *Bread*, *Croissant* ou *Brioche* en fonction de la chaîne de caractères donnée en argument. Les ingrédients requis pour chaque type de pain sont affichés à côté de leurs boutons respectifs.



c. Vente des pains

On peut récupérer un pain si :

- Il est cuit et se trouve directement en dessous, au-dessus, à gauche ou à droite du joueur.
- Il est brûlé

Pour le récupérer, il faut cliquer sur le bouton "Collect & Sell" du panel du boulanger. Celui-ci déclenche l'appel de la méthode `collectBread()` de `Bakery` qui utilise l'algorithme suivant :

Bakery.collectBread():

Entrée: rien

Sortie: rien

Pour chaque oven de de l'ArrayList Ovens :

 si l'oven est occupé :

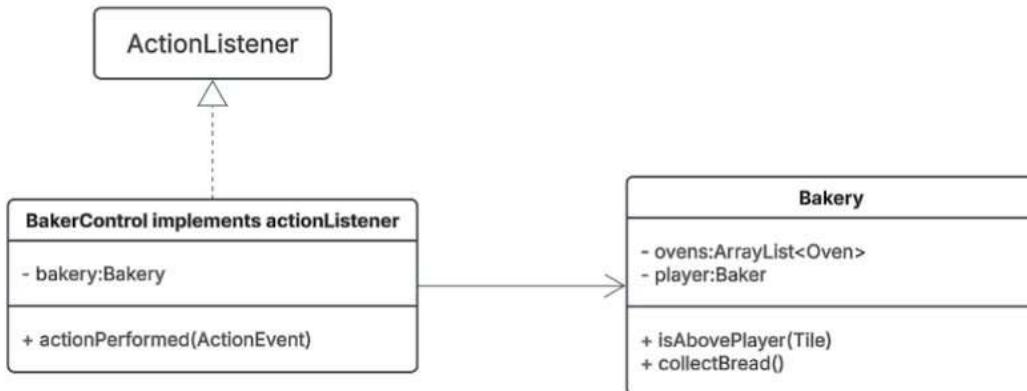
 si son pain est cuit et que l'oven se trouve juste au dessus du joueur :

 on vend le pain

 on retire le pain

 sinon, si son pain est brûlé :

 on retire le pain



Pour vérifier qu'un *Oven* se trouve au-dessus du joueur, on utilise la méthode `isAbovePlayer()` de *Bakery*. Celle-ci vérifie simplement que la *Tile* passée en argument ait le même *x* et un *y* de moins que celle du joueur.

7. Affichage de la boulangerie et de ses composantes

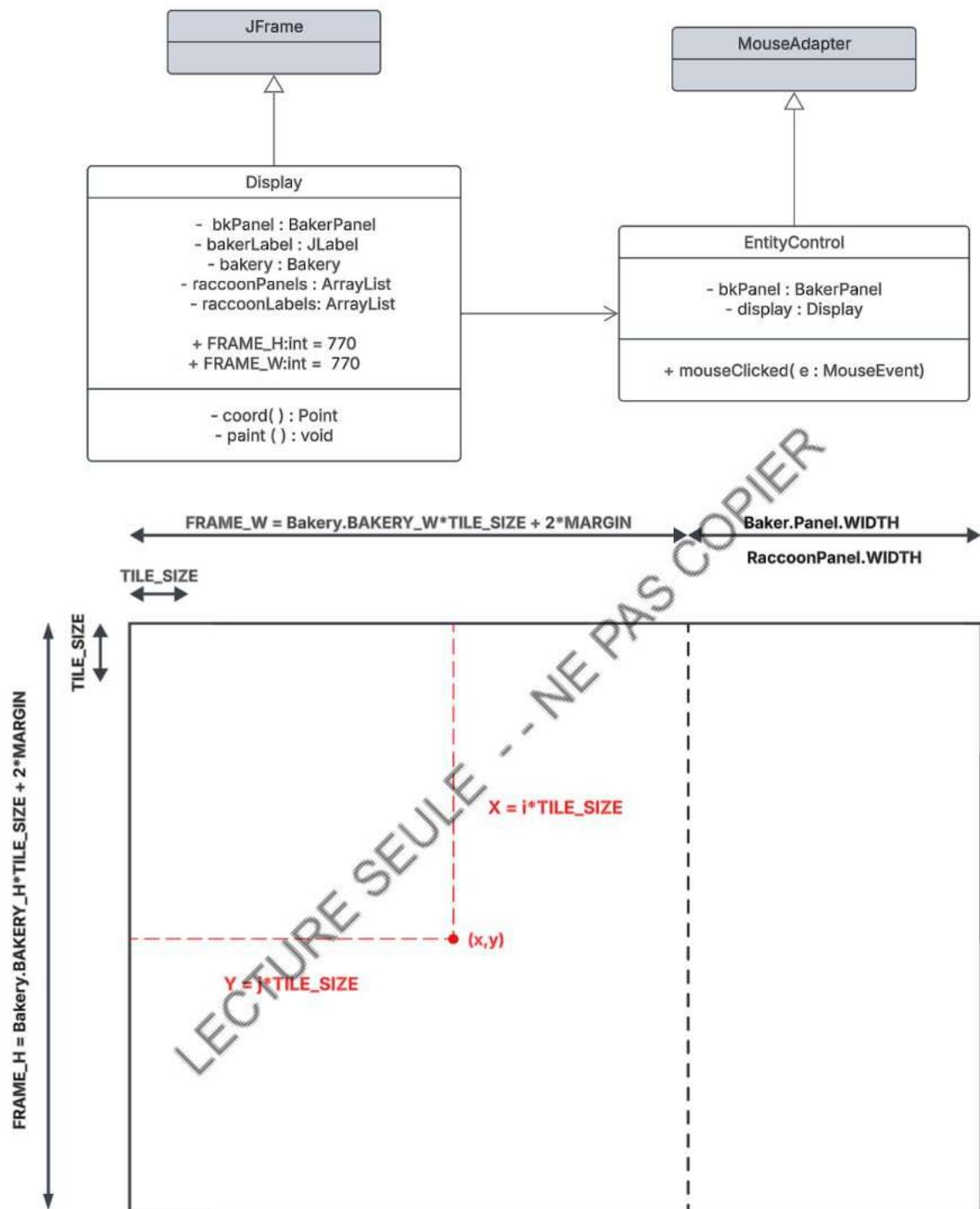
a. Dessiner une fenêtre

Pour afficher notre boulangerie nous utilisons l'**API Swing** et la classe *JFrame*.

Pour chaque composante on récupère sa position du modèle et on la met à l'échelle de la vue pour l'afficher. Pour pouvoir modifier rapidement les dimensions de la boulangerie ainsi que la taille des cases, nous avons définies des constantes :

- 'FRAME_H' qui correspond à la hauteur de la fenêtre, et dépend de la taille de la boulangerie 'BAKERY_H'
- 'FRAME_W' qui correspond à la largeur de la fenêtre
- 'TILE_SIZE = 75' qui correspond à la taille en pixels d'une case

Ci-dessous, le diagramme de classe et le plan de notre vue du jeu :



Plan de notre vue du jeu

b. Dessiner les pains et les fours

On commence par créer des *JLabels* dont le fond représente le four et l'icône représente le pain. Pour cela, dans le constructeur de *Display*, on appelle la fonction *initOvens()* qui utilise l'algorithme suivant :

Display.initOvens :

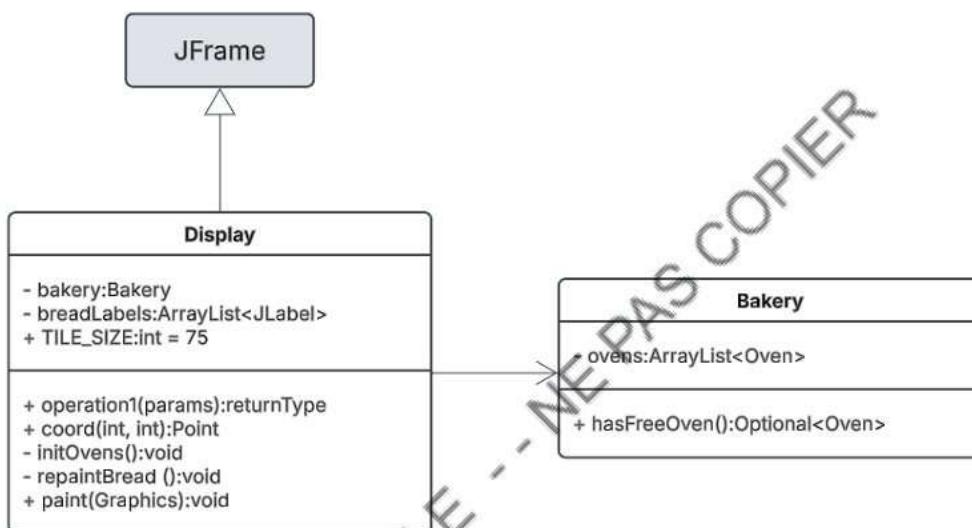
Entrée: rien

Sortie: rien

Pour chaque **Oven** dans l'**ArrayList ovens** de **Bakery** :

- on crée un **JLabel**;
- on multiplie les coordonnées de l'**oven** par la taille d'une case
- on set la taille du Label
- on ajoute le label à l'attribut **breadLabel** de **Display**
- on set différents attributs du label pour avoir un fond gris et une bordure noire

Pour multiplier les coordonnées, on utilisera la fonction **coord()** car c'est une opération qu'on répétera souvent.



Puis, pour mettre à jour l'apparence des pains, on appelle la fonction **repaintBread()** dans la fonction **paint()** de **Display**.

Display.repaintBread :

Entrée, Sortie: rien

pour i allant de 0 au nombre de Ovens :

si le **ovens[i]** est occupé :

- on stocke le nom de fichier correspondant à l'état du pain
- on récupère le **breadLabel[i]**
- on change son icône en fonction du nom du fichier

sinon :

- on met l'icône à null

c. Dessiner les entités (boulanger et raton laveurs)

Pour dessiner les rats laveurs et le boulanger, nous avons créé un **JLabel** pour chaque entité contenant l'image de cette dernière. Pour cela on récupère la 'position' de l'entité dans le modèle puis on ajoute le **JLabel** à l'affichage en adaptant les coordonnées récupérées à la vue en fonction de la taille de la case.

d. Mise à jour de l'affichage

Pour gérer la mise à jour de l'affichage des différentes entités dans la boulangerie ont utilisé un thread qui va appeler la méthode repaint() qui affiche tous les éléments de la boulangerie en récupérant leur 'position' dans le modèle. Ce thread nous sert également à mettre à jour le contenu du panneau de contrôle du boulanger.

8. Affichage du panneau de contrôle du boulanger

Pour créer le panneau de contrôle du boulanger nous utilisons la classe *JPanel*. Concernant la structure du *JPanel* principal, nous avons choisi de configurer sa structure avec un *BoxLayout*. Celui-ci nous a permis d'avoir une structure simple avec une superposition de *JPanel* les uns à la suite des autres verticalement.

D'autre part, pour avoir une organisation plus flexible du *JPanel* contenant les 2 boutons "Collect & Sell" et "Buy", nous avons opté pour le *GridBagLayout*. Ce layout nous a été très utile pour placer les 2 boutons de manière à pouvoir personnaliser l'espace entre eux. Nous avons également choisi de créer des *JPanel* pour chaque éléments à cuire (pain , croissant , brioche). Ces *JPanel* contiennent le boutons pour démarrer la cuisson et la recette qui lui correspond. Nous avons également opté pour le *GridBagLayout*. De plus, nous avons voulu afficher la somme d'argent à atteindre pour gagner et le temps de la partie qui s'écoule. Pour cela, nous avons créé un *JPanel* organisé avec un *BoxLayout*. Dans ce dernier, nous avons ajouté un *JLabel* contenant l'objectif à atteindre et un *JPanel* contenant le temps restant représenté par une *JProgressBar*.

Enfin, pour pouvoir modifier rapidement les dimensions du Panel et des boutons ainsi que les différentes ressources que le boulanger peut acheter, nous avons définies des constantes :

- 'HEIGHT' = 600 correspond à la hauteur du *JPanel* principal
- 'WIDTH' = 300 correspond à la largeur du *JPanel* principal
- 'BUTTON_H' = 50 correspond à la hauteur des boutons
- 'BUTTON_W' = 200 correspond à la largeur des boutons
- 'RESSOURCES' = { "flour" , "egg" , "yeast" , "butter"}

Notons que ce panneau de contrôle ne s'affiche que si on clique sur le boulanger en suivant l'algorithme ci-dessous :

Entrées : MouseEvent e, bakerLabel, bakerPanel

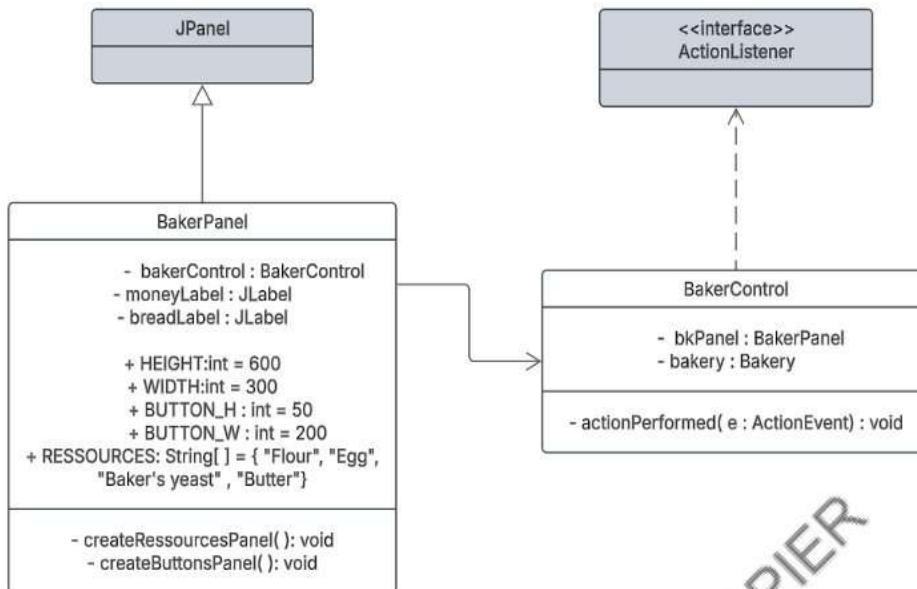
Sortie : rien

Si e.source == bakerLabel alors :

On cache tous les panneaux raccoons avec SetAllRaccoonPanelNonVisible()

On ajoute le bakerPanel à l'affichage

On rend visible le bakerLabel



9. Affichage du panneau de contrôle du raton-laveur

Pour créer le panneau de contrôle des rats-laveurs, nous avons également utilisé un *JPanel*.

Concernant la structure du *JPanel* principal, nous avons choisis de configurer sa structure avec un *BorderLayout*. Celui-ci nous a permis d'avoir une structure simple avec les positions Nord, Sud, Est et Ouest pour placer les autres *JPanel*.

En outre, nous avons choisi de créer un *JPanel* qui va contenir toutes les informations liées au raton laveur. Ce dernier est organisé grâce à un *GridBagLayout* pour pouvoir placer correctement les 2 autres *JPanel*. En effet , nous avons créé un *JPanel* qui contient une image du temps et une barre de progression qui représente le temps écoulé avant que le raton-laveur ne sorte de la boulangerie.

Nous avons également créé un *JPanel* qui contient le nombre de pain , croissant et brioche volés.

Enfin, pour pouvoir modifier rapidement les dimensions du Panel qui sont les mêmes que celle du panel du boulanger, nous avons défini des constantes tel que :

- 'HEIGHT' = `BakerPanel.HEIGHT` correspond à la hauteur du panel principal
- 'WIDTH' = `BakerPanel.WIDTH` correspond à la largeur du panel principal

Notons que ce panneau de contrôle ne s'affiche que si on clique sur un raton-laveur en suivant l'algorithme ci-dessous :

Entrées : MouseEvent e, raccoonLabel_list, raccoonPanel_list

Sortie : rien

Pour i allant de 0 au nombre de label de raton-laveur faire :

Si e.source == raccoonLabel_list[i] alors:

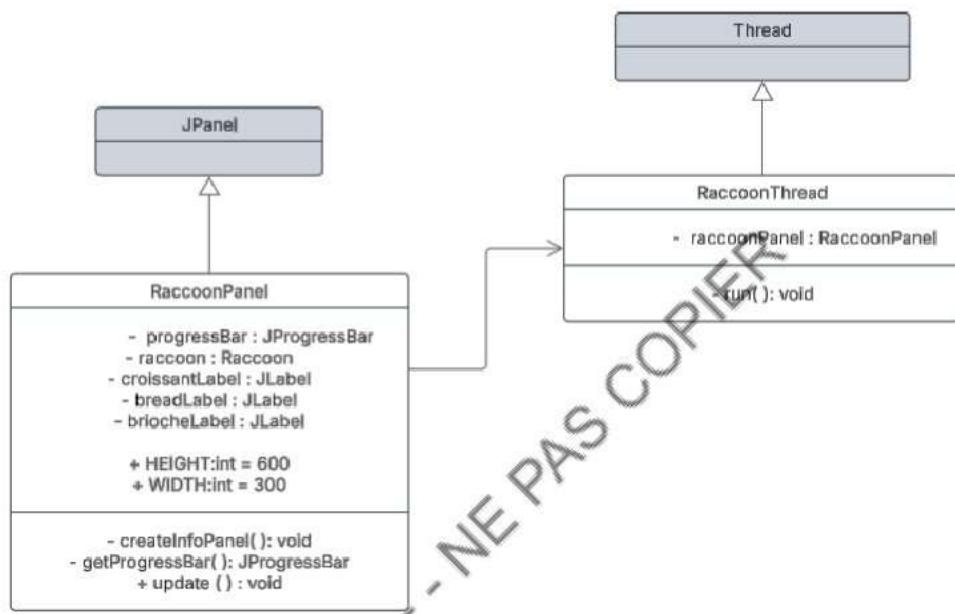
On cache le panneau du boulanger avec SetBakerPanelNonVisible()

On ajoute le raccoonPanel_list[i] à l'affichage

On rend visible le raccoonPanel_list[i]

Sinon faire :

On cache le panneau du raton_laveur

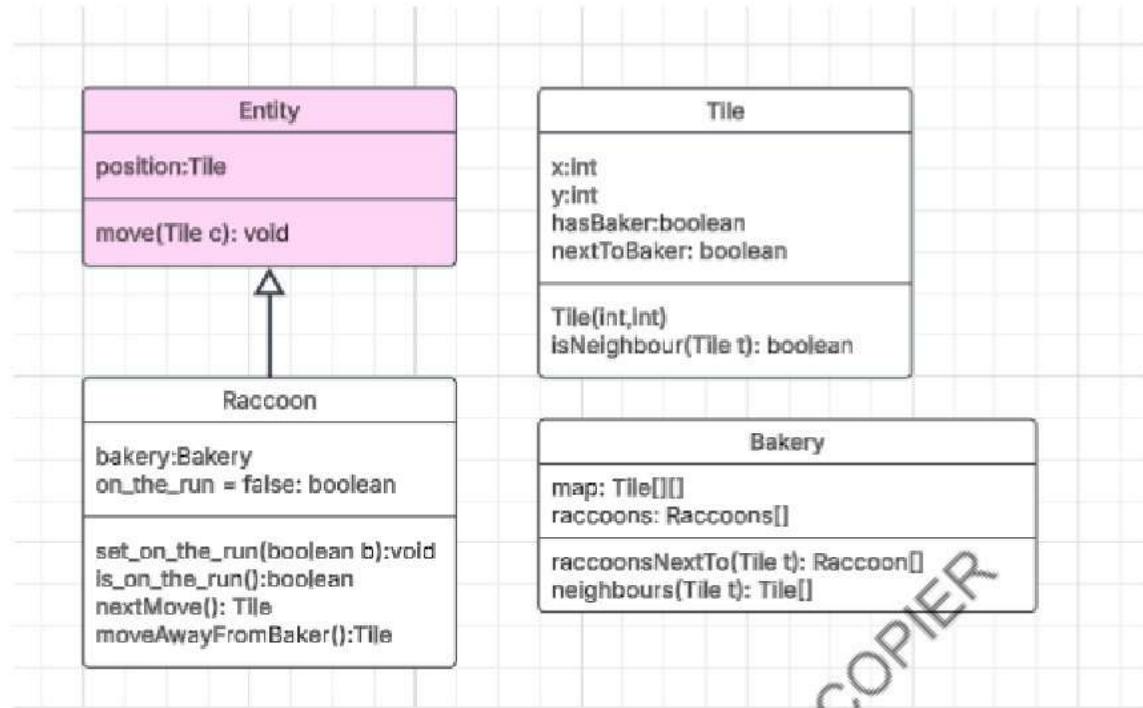


10. Chasser les raccoons

Ici pour faciliter l'implémentation de cette fonctionnalité, nous avons ajouté un attribut ‘nextToBaker’ à la classe *Tile*. Comme la plupart des autres classes ont accès à la classe *Bakery*, cela permet d'éviter le dédoublement d'informations et d'éviter des recalculs inutiles en simplifiant la logique. Dans *Raccoon* lors de la fonction `nextMove()`, on regarde s'il doit s'enfuir et si oui on cherche une case voisine qui ne soit pas à côté du boulanger, s'il n'y en a pas il reste coincé ou il est. Cette façon de faire permet d'assurer que quelque soit le mouvement courant du raton-laveur, il le finira avant de partir de fuite, évitant ainsi de faire planter le jeu.

Pour mettre à jour l'attribut ‘nextToBaker’, lorsque le boulanger bouge, dans la fonction `move()` de *Baker*, on met à faux toutes les cases voisines de la position précédente puis son `update` la position du boulanger et on met à vrai toute ses cases voisines. De plus l'attribut de fuite du raton-laveur est mis à vrai dans la fonction `move()` du boulanger.

Lors de l'affichage de la boulangerie, on utilise tout simplement la position du boulanger pour dessiner une zone autour de celui-ci représentant la zone interdite.



11. Ecran de démarrage

a- Ajout d'une image de fond, bouton play et son :

Pour donner plus de vie à notre jeu, nous avons créé une fenêtre de taille 800x600 pixels, avec une image de fond générée par IA. Afin d'ajouter correctement les boutons par-dessus l'image, nous avons définie cette dernière comme étant l'arrière-plan en utilisant un *JLayeredPane*. Ensuite nous avons ajouté le bouton 'Play' de début de jeu et le bouton d'activation/désactivation du son.

b- Création JPanel pour le choix des niveaux :

Pour plus de diversité, nous avons mis en place un système de choix de niveaux géré par un *JPanel* de taille 300x100 définie dans les constantes 'LEVEL_W' et 'LEVEL_H' et un thread pour l'animation détaillé dans la sous-section suivante. Nous avons choisi d'organiser ce *JPanel* manuellement pour qu'on puisse personnaliser la position de deux boutons de navigation et un *JLabel* pour le niveau .

c- Implémenter un Thread pour gérer l'animation du choix des niveaux

Comme mentionnée précédemment , nous avons implémenté un thread d'animation pour le choix de niveaux.

En effet, lorsqu'on clique sur le bouton de droite ou de gauche on a un système de défilement des niveaux pour réaliser ce choix. Notons que ce défilement est circulaire c'est-à-dire quand on atteint le dernier niveau si on clique sur le bouton de droite on obtient le premier ceci est réalisé en suivant l'algorithme ci-dessous.

De plus,derrière chaque choix de level il y a un fichier texte qui le représente détaillé dans la section suivante de notre documentation.

Algorithme si on clique sur un bouton de navigation :

Entrées : String direction , levelPanel , LEVEL_W , MOVE

Sortie : rien

On récupère le levelLabel a partir du levelPanel

Tant que c'est vrai faire :

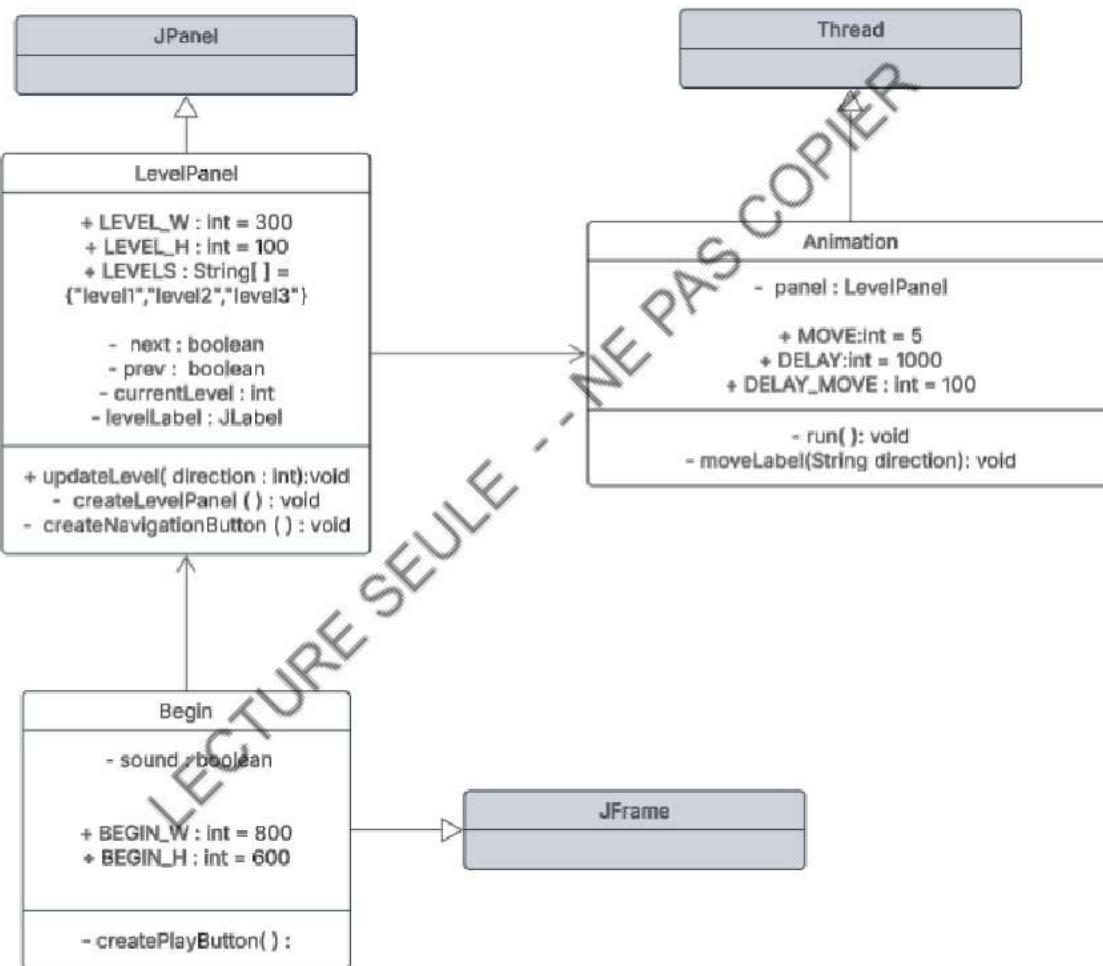
 On récupère la position x du label

 Si direction == "right" et x < LEVEL_W alors :

 On décale vers la droite le label avec un pas égal à MOVE

 Si direction == "left" et x > 0 alors :

 On décale vers la gauche le label avec un pas égal à MOVE



12. Plusieurs niveaux

Pour rallonger la durée de notre jeu et nous permettre d'y ajouter plus de contenu, nous avons décidé de mettre un système de niveaux. Pour cela, nous créons manuellement des cartes dans des **fichiers .txt** (stockés dans le dossier “*levels*”) dont les symboles correspondent aux éléments de la boulangerie : ‘_’ pour une case normale, ‘O’ pour un four, ‘B’ pour le joueur, ‘R’ pour un raton laveur et ‘S’ pour le spawn des rats-laveurs lorsqu'ils réapparaissent. De plus, la taille en cases de la map est indiquée au début du fichier suivie par l'objectif monétaire à atteindre pour gagner, du nombre de rats-laveurs et de la durée du niveau.

Pour le moment, ceux-ci sont sélectionnables sur l'écran de démarrage, avec un panel dédié. Lorsque l'on sélectionne un niveau, le constructeur de *Bakery* prend le fichier en argument pour fixer ses constantes de tailles et initialiser l'array des raccoons. Puis en traversant ligne par ligne le fichier, le constructeur initialise les cases selon le symbole présent.

13. Musique de fond

Concernant la musique de fond , celle-ci démarre uniquement lorsqu'on clique sur le bouton Play qui est dans l'écran d'accueil *Begin* et que le son est activé. En cliquant sur le bouton de son, on change l'état booléen du flag ‘sound’ (qui est à true par défaut). Si ‘sound’ est à true au début du jeu, alors la musique joue; sinon, elle ne joue pas.

Lorsque le jeu se termine, nous avons fait le choix de couper le son exactement à la création de l'écran de fin en remettant le flag à faux.

Lorsque l'on clique sur le bouton de son :

Entrée : ActionListener

Sortie : rien

Si ‘sound’ est à vrai :

On met à faux

On change l'icône du bouton pour montrer que le son est OFF

Sinon :

On met à vrai

On change l'icône du bouton pour montrer que le son est ON

Ci-dessous, la méthode appelée si on a démarré le jeu avec du son :

Begin.PlaySound :

Entrée, Sortie : rien

On crée un File avec le chemin de la musique

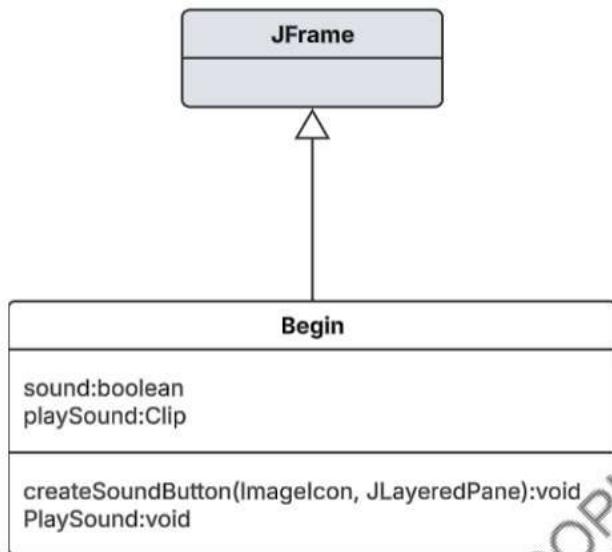
On récupère l'AudioStream du fichier

On récupère le Clip audio

On ouvre l'AudioStream

On loop le Clip de façon à ce qu'il joue en boucle pendant le jeu

On démarre la musique



14. Fin de jeu

Enfin, pour la fin de jeu, le thread *EndGame* se charge de mettre fin à la partie lorsque le temps est écoulé, en instanciant une fenêtre de fin de jeu *EndScreen* de 800x600 pixels. Sur l'écran de fin de jeu se trouvent un image de fond et un message de fin (qui changent selon que le joueur ait gagné ou non), avec le but à atteindre et la somme d'argent possédée par le *Baker*. **Si le joueur atteint ou dépasse l'objectif, la partie est remportée; sinon, elle est perdue.** En bas de l'écran, un bouton 'Replay' permet de lancer une nouvelle partie, et un bouton 'Quit' ferme le programme.

Dans *EndGame*, la variable 'TIME' correspond à la durée en secondes d'un niveau. L'attribut 'timeLeft' est initialisé à 'TIME' lors de la création du thread, et décrémente secondes par secondes. Lorsque le temps est écoulé, un *EndScreen* est créé.

Ci-dessous les algorithmes utilisés :

EndGame.run :

Entrée, Sortie : rien

Tant que timeLeft > 0 :

On attend 1 seconde

On décrémente timeLeft

On affiche "Time's up !" dans le terminal

On crée un écran de fin *EndScreen* en récupérant 'display'

On rend visible cet écran de fin

Constructeur de *EndScreen* :

Entrée : Display

Si le son joue :

On arrête le son

On récupère **Bakery** et **Baker**

'GOAL' <- le 'GOAL' de **Bakery**

On cache **Display**

On crée la fenêtre de fin :

On appelle **Baker.gameEnded()**

On affiche le 'GOAL' et l'argent du joueur

Si le joueur gagne :

On met l'image de fond gagnante

On affiche "YOU WIN !"

Sinon :

On met l'image de fond perdante

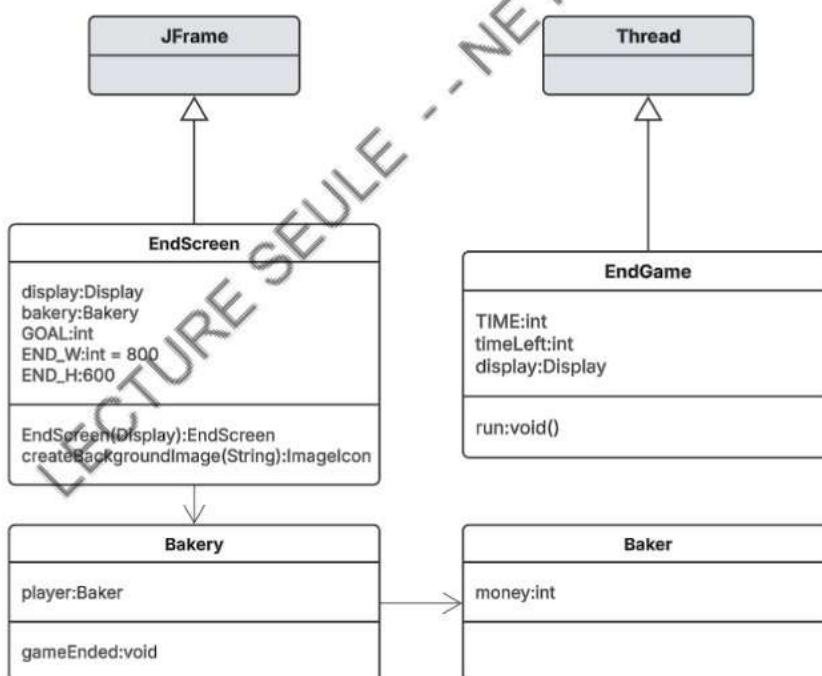
On met un message de perte de partie

On crée un bouton 'Replay' :

Lorsqu'on clique dessus -> ferme l'écran et crée un écran **Begin**

On crée un bouton 'Quit' :

Lorsqu'on clique dessus -> ferme le jeu



5 - Documentation Utilisateur :

Ci-dessous figurent les indications permettant de lancer le programme et d'y jouer.

Prérequis :

Pour lancer et faire tourner ce programme, il est nécessaire d'avoir un ordinateur (le code ne tournant actuellement pas sur smartphone) sur lequel est installé le langage de programmation Java. Il faut également avoir un environnement de développement (IDE) permettant de faire tourner des programmes Java, préféablement IntelliJ. Aucun système d'exploitation particulier n'est requis.

Si Java n'est pas déjà installé sur votre ordinateur, il est possible de le télécharger à l'adresse suivante : <https://www.java.com/en/>

Si vous n'avez pas d'IDE permettant de faire tourner des programmes Java, IntelliJ Community Edition est disponible à l'adresse suivante : <https://youtrack.jetbrains.com/articles/IDEA-A-21/IDEA-Latest-Builds-And-Release-Notes>

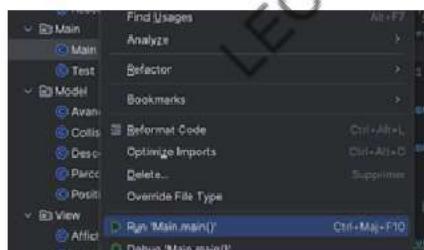
Etapes de lancement du programme :

1. Ouvrir le code dans l'IDE choisi. Dans l'exemple suivant, nous supposons être sur l'IDE IntelliJ
2. Ouvrir le dossier du programme depuis l'éditeur d'IntelliJ :

Main Menu (4 barres verticales) > File > Open > naviguer dans l'explorateur de fichier jusqu'au dossier en question



3. Clic droit sur le fichier Main.java dans l'arborescence du projet > Run 'Main.main()'



4. Pour toutes les fois d'après : cliquer sur le triangle vert en haut à droite pour immédiatement lancer le programme



Une fois en jeu :

Le joueur peut se déplacer à l'aide des flèches directionnelles et de ZQSD. Pour acheter des ingrédients, cuire des pains ou vendre des pains, il faut cliquer sur le sprite du joueur pour faire apparaître le panneau de contrôle.

Les rats-laveurs se déplaceront constamment en direction des pains pour tenter de les voler : il est possible de cliquer sur un raton-laveur pour voir le nombre de pains voler, ainsi que leur barre de vie : lorsque celle-ci tombe à 0, il disparaît et est immédiatement remplacé par un nouveau raton laveur, qui apparaît en bas à droite de la carte.

6 - Documentation Développeur :

Le fichier Main.java et sa fonction main() permettent de lancer le jeu : c'est ici que sont instanciés la boulangerie, l'affichage et les threads RaccoonMovement, RaccoonLife et Redraw. L'affichage du jeu, Display.java récupère les attributs de Bakery et instancie les panneaux de contrôles, BakerMovement et EntityControl.

Les constantes sont toutes écrites en majuscules. Certaines d'entre elles ont un impact direct sur la taille de l'écran de jeu, comme TILE_SIZE, BAKERY_H et BAKERY_W, dont dépendent les dimensions de l'écran de jeu.

Le fichier Bakery.java rassemble les données principales du jeu : le joueur, la carte de cases, les rats-laveurs et les fours contenant des pains.

Chaque pain possède son propre thread de cuisson, qui est tué si le pain est vendu ou brûlé. De la même manière, chaque raton-laveur possède son propre thread de vie, instancié à son apparition sur le jeu.

Pour ajouter des niveaux de jeux, il faut créer une carte sur un fichier txt en s'inspirant de ceux présents dans le dossier "levels". Cependant, notre fenêtre de jeu ayant une taille fixe, la limite pour une carte est de 13 sur 13 cases. Toute carte ayant une taille supérieure dépassera de l'écran et ne sera pas visible.

7 - Résultats, conclusions et perspectives :

1. Ce qui a été réalisé :

Nous avons réussi à implémenter l'intégralité des fonctionnalités définies dans l'analyse. Voici quelques captures d'écrans de notre jeu, à l'état actuel :

Tout d'abord, l'écran de démarrage :

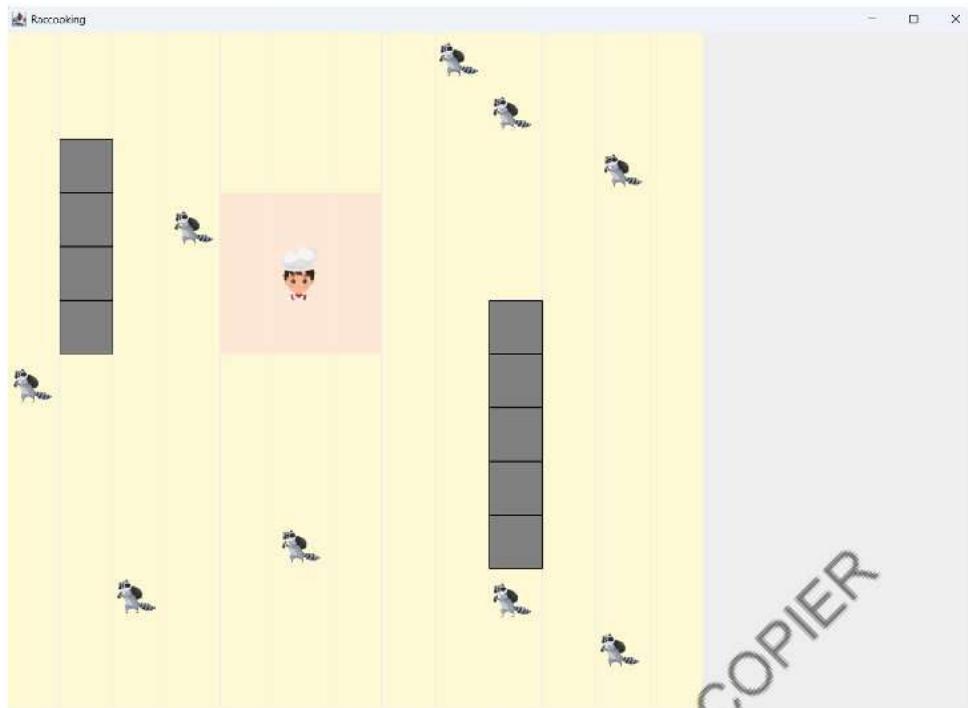


Changement d'apparence du bouton de son

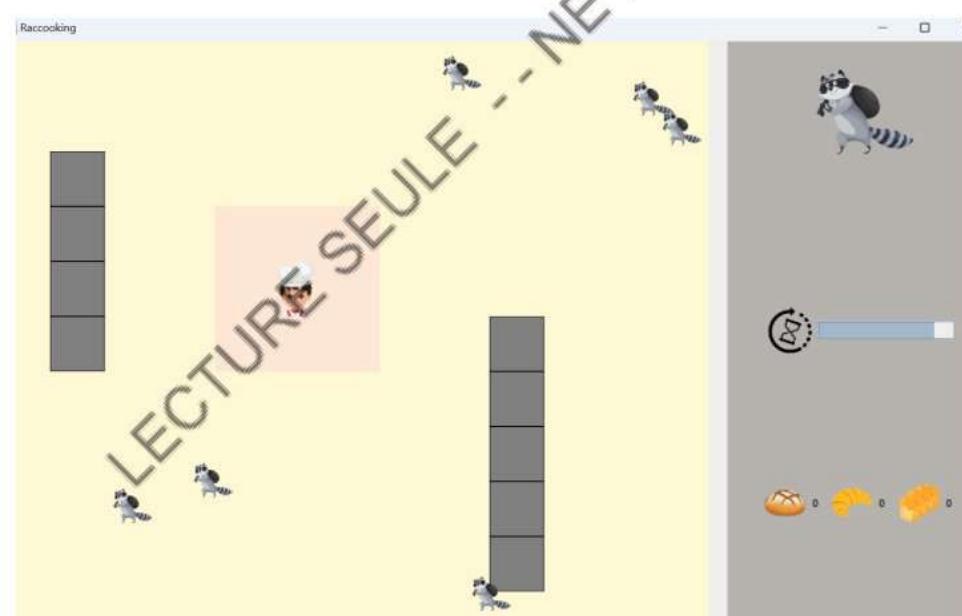


Animation de sélection de niveau

Maintenant, l'écran au démarrage du jeu et les panneaux de contrôle. Il s'agit ici du niveau 1 :



Écran d'affichage au démarrage du jeu. La zone rouge autour du joueur sert de périmètre de sécurité faisant fuire les rats-laveurs



Panneau de contrôle lorsque l'on clique sur un raton-laveur. Affiche la barre de vie du raton-laveur, ainsi que le nombre de pains volés.



Panneau de contrôle du joueur, lorsque l'on clique sur le boulanger.

Indique la somme d'argent et les ingrédients possédés, et affiche boutons permettant respectivement de récupérer et vendre ses pains, d'acheter des ingrédients et d'ajouter un pain, brioche ou croissant à la cuisson (avec les ingrédients nécessaires indiqués)

En bas se trouvent l'objectif monétaire du jeu et le temps restant



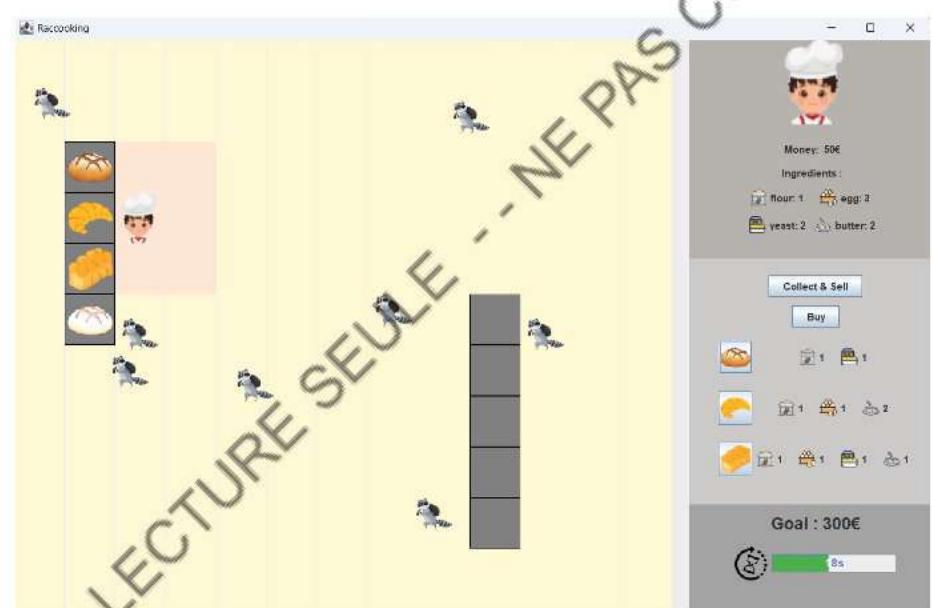
Achat d'ingrédients lorsque l'on clique sur le bouton "Buy" du panneau de contrôle. Cocher une case permet de débloquer le sélecteur de quantité ou d'écrire directement dedans.

Les prix à l'unité sont indiqués à côté de chaque ingrédient

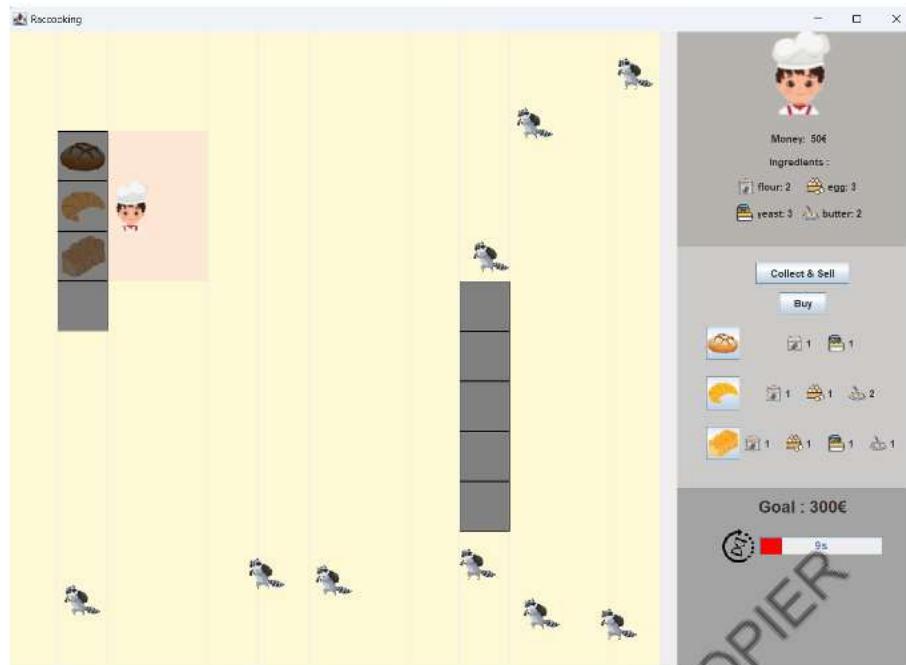
Ci-dessous, les affichages des pains en fonction de leurs degrés de cuisson :



Affichage des pains au début de la cuisson



Ici, un pain, une brioche et un croissant sont cuits : ils sont prêts à être collectés et vendus

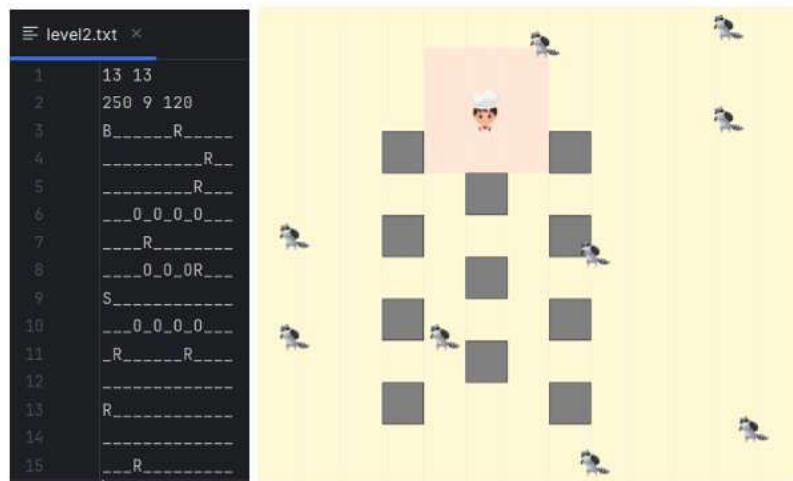


Ici, 3 pains brûlés. Ils peuvent être collectés pour libérer des fours, mais ne rapportent pas d'argent.

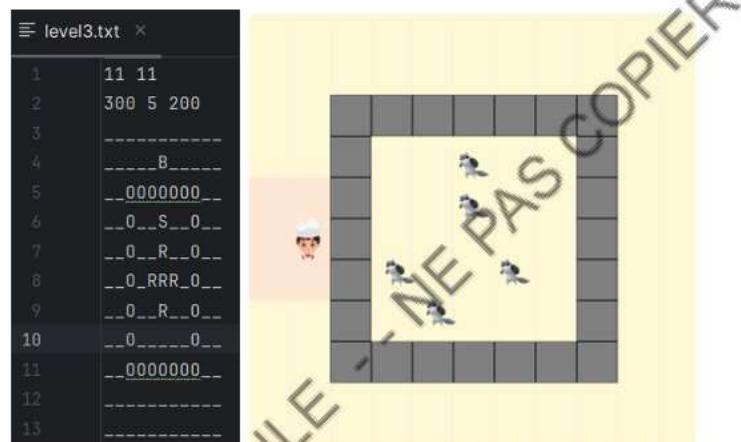
Comparaisons des niveaux et de leurs fichiers .txt correspondants :



Niveau 1, du fichier level1.txt



Niveau 2, du fichier level2.txt



Niveau 3, du fichier level3.txt

Enfin, voici les 2 écrans de fin possibles :



Partie perdue



Partie remportée

2. Ce que nous aurions aimé faire ou améliorer

- **Raccourcis clavier** : dans le *BakerPanel* de nombreux clics sont nécessaires, par exemple pour acheter des ingrédients. Il aurait été intéressant d'ajouter des raccourcis clavier pour diminuer le nombre de manipulations et s'adapter aux préférences de chaque utilisateur.
- **Taille de la map** : même si les maps peuvent avoir n'importe quelle taille, la taille des cases reste la même ce qui veut dire qu'on peut se retrouver avec une fenêtre immense qui dépasse de l'écran ou une fenêtre minuscule. Pour gérer ce problème, nous aurions pu mettre une taille de fenêtre fixe et calculer la taille des cases en fonction de la taille de la map.
- **Lier les Labels aux Ovens** : il aurait fallu lier les *Label* aux *Oven* pour éviter de changer l'image, et donc de les redessiner à chaque fois.
- **Animation ou indicateur visuel de la disparition d'un raton-laveur** : actuellement, on ne remarque pas forcément la mort d'un raton-laveur. Si nous avions eu plus de temps, il aurait pu être intéressant d'ajouter un indicateur visuel lors de la disparition d'un raton-laveur de la carte, et peut-être aussi un pour l'apparition d'un nouveau.

3. Conclusion :

Nous avons toutes beaucoup apprécié ce projet d'informatique. Non seulement il nous a permis d'apprendre à faire de la programmation concurrente avec des Threads, mais il nous a aussi permis de consolider nos acquis en programmation objet, patron MVC et interfaces interactives. Cette UE nous a également donné l'occasion de travailler en plus grands groupes qu'auparavant, et nous a fait acquérir

Groupe 1D - Nour **Ettayeb**, Sidonie **Minodier**, Victoria **Myot** et Mathilde **Needham**

des méthodes de collaboration que nous ne connaissions pas encore. Enfin, nous avons beaucoup apprécié les libertés laissées dans le sujet du projet, qui nous ont permis d'implémenter toutes les idées que nous voulions.

LECTURE SEULE -- NE PAS COPIER