Alexandre MOUTOUH
Isis ALIE-SANDEVOIR

M. MAVROMATIS

PROJET RABBITWORLD

Document de conception

Polytech Marseille
13/01/2018

Alexandre MOUTOUH
Isis ALIE-SANDEVOIR
INFO4A

POLYTECH MARSEILLE
Aix-Marseille Université

Table des matières

Lien Github : https://github.com/IsisAlieSandevoir/RabbitWorld

1. Introduction

Le projet est un jeu de la vie, programmé en langage objet, suivant les règles de la modélisation objet. La principale problématique du projet est donc de réaliser un jeu suivant une modélisation objet bien conçue. Le jeu devra suivre certains critères comme l'utilisation du polymorphisme, de "Design Patterns" et d'interactions entre objets.

L'équipe a fait le choix de réaliser un jeu de la vie concernant un champ avec des lapins et des carottes.

Du point de vu organisationnel, le projet doit aussi être conçu selon la méthode agile soit suivant de courtes périodes appelées sprint avec un rapport au client à la fin de ces périodes.

2. Présentation du projet

2.1 Outils utilisés

Le projet est programmé en JAVA. Nous avons fait ce choix car l'une des contraintes du client est l'utilisation d'un langage orienté objet ainsi que l'utilisation de notions orientées objet telles que l'héritage, le polymorphismes et l'interaction entre objet.

Les schémas présentés respectent les conventions UML.

Le jeu doit être capable d'être exécuté sur n'importe quel ordinateur sous Unix et Windows avec Java 1.8 d'installé.

2.2 Présentation du concept du jeu

Les principaux services attendus par l'utilisateur sont de pouvoir jouer à un jeu de la vie (Conway game), et observer l'évolution des cellules du jeu. La thématique du jeu est la vie des lapins. Ainsi, dans le cadre de ce projet, les cellules deviennent des lapins.

Les principaux traitements sont la naissance des lapins, la vie des lapins, la reproduction des lapins et la mort des lapins. De manière plus secondaire, est observable l'évolution du cycle de vie des carottes. En l'occurrence, dans notre cas, la pousse d'une carotte dans le champ, le vieillissement des carottes (qui deviennent alors empoisonnées), l'ingestion des carottes par les lapins (qui signifie la mort des de la carotte).

2.3 Fonctions générales du logiciel

Les principales fonctions devant être assurées par le logiciel sont donc les règles de vie du jeu. Soit la naissance, la vie et la mort des lapins dans un champ. Et de façon analogue, l'évolution des cycles de vie des carottes.

2.4 Règles de vie et structure du jeu

Le jeu est une simulation de la vie des lapins et des carottes. Tous ces acteurs évoluent sur une grille de jeu de 20x20 cases étant vides ou contenant un acteur.

Le temps est représenté par des tours de jeu.

A chaque tour, les règles de vie du jeu définies sont les suivantes :

- Les lapins, pouvant être mâles ou femelles et adultes ou bébés, se déplacent de façon aléatoire sur le champ (grille de jeu) et perdent un certaine quantité de vie (1 point pour les adultes et 2 pour les bébés)
- Lorsque la vie d'un lapin atteint 0, il meurt (disparaît de la grille)
- Les carottes, pouvant être normales ou empoisonnées, sont fixes sur des cases de la grille
- Un lapin mâle et un lapin femelle sur des cases adjacentes donnent naissance à 2 bébés lapins ayant initialement 10 points de vie, sur des cases adjacentes au couple. Ces dernières majorent le nombre de bébés, c'est-à-dire que si elles sont trop peu nombreuses pour accueillir les 2 bébés, leur totalité sera utilisé et moins de bébés seront nés.
- Lorsqu'un lapin se déplace sur une case contenant une carotte normale, il la consomme et gagne de la vie (3 point pour les adultes, 8 pour les bébés)
- Lorsqu'un lapin se déplace sur une case contenant une carotte empoisonnée, il la consomme et meurt
- Lorsqu'un bébé lapin a survécu pendant 6 tours, il devient adulte
- Indifféremment de sa nature, lorsqu'une carotte est mangée, un carotte prends racine sur une case aléatoire de la grille. Au bout de 5 tours, si cette case est vide la carotte pousse, sinon son apparition est retardée d'un tour
- Si une carotte n'est pas consommée avant 10 tours depuis sa pousse, elle devient empoisonnée.

2.5 Exigences fonctionnelles

L'utilisateur entre à la main les paramètres du jeu, le nombre de lapins adultes, le nombre de carottes et le nombre de carottes empoisonnées.

Les éléments constituant le système sont les lapins, les carottes (empoisonnées ou non), et le champ. Ces éléments sont alors des objets qui doivent être manipulés selon une modélisation et une programmation adaptée à cet effet.

Les exigences fonctionnelles sont donc :

- le jeu doit être un jeu de la vie
- le programme doit mettre en valeur les interactions entre objet
- le jeu doit suivre précisément une modélisation orientée objet

2.6 Exigences non fonctionnelles

Les contraintes non-fonctionnelles sont :

- L'interface graphique qui est facultative mais qui a été réalisée.
- Ajout et choix d'autres règles de vie.
- Ajout d'ennemis (renards) et/ou de renaissance d'un lapin mort.

2.7 Exigences opérationnelles

2.7.1 Environnement matériel

Il est requis que Java 1.8 soit installé sur la machine exécutant le jeu.

2.7.2 Environnement logiciel

Il n'y a pas de logiciels imposés. Le projet doit se faire en langage orienté objet (C++, Java, C#...). Notre choix s'est porté sur Java. Le projet est développé sur l'environnement de développement intégré (IDE) Eclipse avec la version 1.8 de Java.

De plus, l'interface Homme-Machine fait appel à la bibliothèque graphique Awt/Swing.

Enfin, le logiciel de gestion de versions Git est également utilisé pour l'équipe et pour que le client puisse consulter l'avancement du projet quand il le souhaite.

Pour la répartition des tâches et l'organisation générale du projet, l'équipe a choisi de suivre la méthode agile via des sprints et des rapport d'avancement (cf 2.7.6 & annexes).

2.7.3 Mise en œuvre

Le logiciel se lance et s'utilise depuis un terminal ou à partir d'une interface graphique. L'utilisateur paramètre le jeu en début de partie en choisissant le nombre de lapins, de carottes et de carottes empoisonnées.

2.7.4 Performances

Le temps de latence entre chaque tour de jeu doit être raisonnable (pas trop long mais pas trop rapide non plus pour laisser le temps à l'utilisateur de bien visualiser l'état pendant le tour et pouvoir ainsi bien comprendre l'évolution de l'état du jeu au tour suivant).

Le temps d'évolution des lapins (pour que le bébé lapin deviennent adulte) est de quelques tours de jeu dans le but d'éviter d'amener le jeu à un moment inintéressant où il n'y a que des bébés lapins.

Le temps de pousse d'une carotte est très court car il est plus intéressant que le jeu tourne et que les carottes puissent être mangées plutôt que les lapins se déplacent sans action particulière.

2.7.5 Sécurité

Le jeu est considéré comme une simulation visuelle. Il n'y a pas de réelles contraintes de sécurité.

2.7.6 Gestion de projet

Le projet va être conduit suivant la méthode agile. Cette méthode consiste à enchaîner des sprints de deux semaines pour ce projet. A chaque fin des quinzaines de jours, une réunion avec le client est réalisée pour faire part de l'avancement (cf annexes) du projet et pour discuter des avancements et/ou modifications éventuelles.

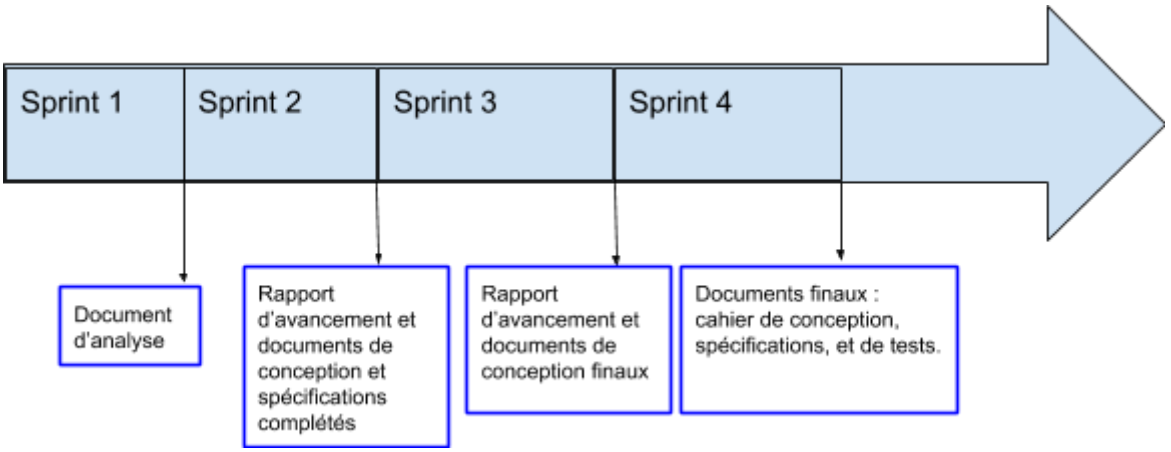
Dans le cadre de ce projet, le suivi du client de l'avancement sera aussi réalisé grâce à l'outil de gestion de versions Github.

Le tableau des sprints prévus pour la réalisation du projet.

Sprints	Description des activités durant le sprint	Dates
Sprint 1	Mise en place de la map et des acteurs : le champ et les lapins et carottes.	04/12/17– 18/12/17
Sprint 2	Mise en place du Controller	18/12/17 – 08/01/18
Sprint 3	Mise en place des interactions et règles de vie.	08/01/18 – 22/01/18
Sprint 4	Tests finaux et affinement du jeu	22/01/18 – 15/02/18

Les sprints sont donc composés d'une quinzaine de jours représentés sur la flèche ci-dessous. Chaque fin de sprint est marquée par une réunion avec le client où des documents d'analyse du projet seront rendus et les nouveautés mises en place pour le prochain sprint. A la fin de chaque sprint, un dossier de tests sera également rendu. Le dossier de spécification sera réalisé au début pour la globalité du projet. A

chaque sprint, il pourra éventuellement être modifié si cela est nécessaire au cours du projet.

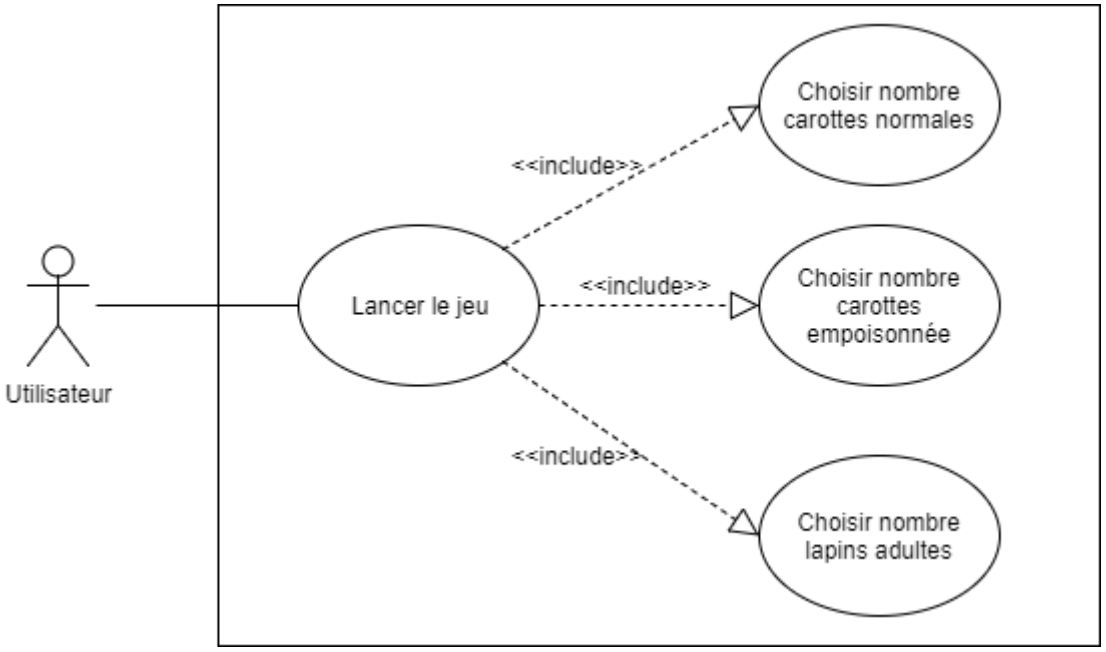


3. Modélisation orientée objet

3.1 Rôle de l'utilisateur

La seule interaction de l'utilisateur avec le jeu est au début d'une partie, lorsqu'il doit paramétrer l'état initial du jeu. Le paramétrage comprends : les nombres de lapins adultes, de carottes normales et de carottes empoisonnées initiaux.

Diagramme de cas d'utilisation



3.2 Les fonctionnalités

Les principales tâches de chaque acteur sont :

- Pour les lapins :
  - naître : provient de la reproduction de deux lapins (ref. FL1)
  - grandir (passage de l'âge bébé à l'âge adulte) (ref. FL2)
  - manger : interaction avec les carottes (ref. FL3)
  - se reproduire entre lapins : interaction entre lapins et lapines (ref. FL4)
  - mourir (ref. FL5)
- Pour les carottes :
  - pousser (ref. FC1)
  - être mangées par les lapins : interaction avec les lapins (ref. FC2)
  - pourrir (ref. FC3)
  - être mangées par les lapins puis les tuer : interaction avec les lapins par une carotte empoisonnée (ref. FC4)

3.3 Interactions entre objets

Le principe du langage objet est de réaliser des interactions entre des objets. Ainsi, ci-dessous sont listées les interactions entre les objets de notre jeu :

- Reproduction de deux lapins : Deux lapins adultes de sexe différents et adultes sur des cases côte à côte peuvent se reproduire. Ils créent 4 lapins enfants qui ne peuvent pas se reproduire et grandissent au bout d'un certains temps fixé.

- Ingestion de carotte par les lapins : Les carottes ne se déplacent pas lorsqu'elles ont poussées. Lorsqu'un lapin arrive sur une case où se trouve une carotte alors cette dernière est mangée. La grille est mise à jour, et la lapin gagne en vie. Le gain de vie est différent selon si le lapin est adulte (+3) ou bébé (+8).

- Ingestion de carotte empoisonnée : Les carottes empoisonnées sont des carottes étant restées trop longtemps sans avoir été mangées. Lorsqu'un lapin mange une carotte empoisonnée, la grille est mise à jour, la carotte disparaît et le lapin aussi (ils meurent tous les deux).

Si un lapin vit trop longtemps sans manger, ce dernier meurt. La fin du jeu se définit par la mort de tous les lapins.

Diagramme de séquence lié au cas d'utilisation FL5 (mourir)

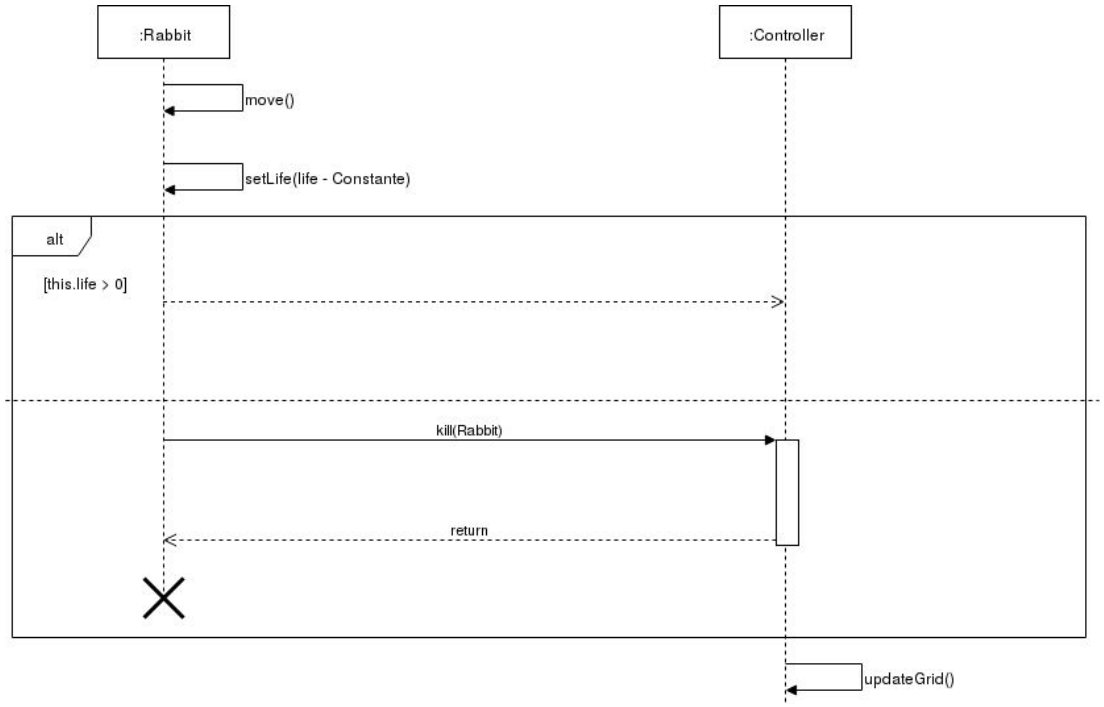


Diagramme de séquence associé au cas d'utilisation FL3, FL5 et FC4 (manger, mourir, ingestion carottes empoisonnées)

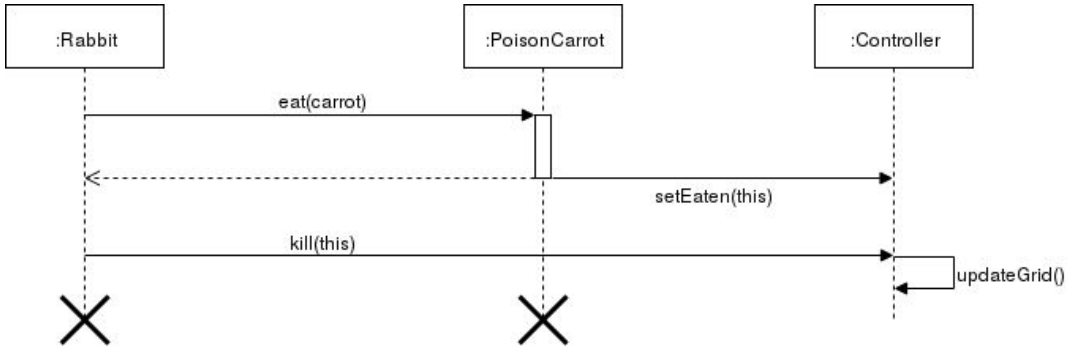


Diagramme de séquence associé au cas d'utilisation FL3 et FC2 (lapin mange, carottes sont mangées)

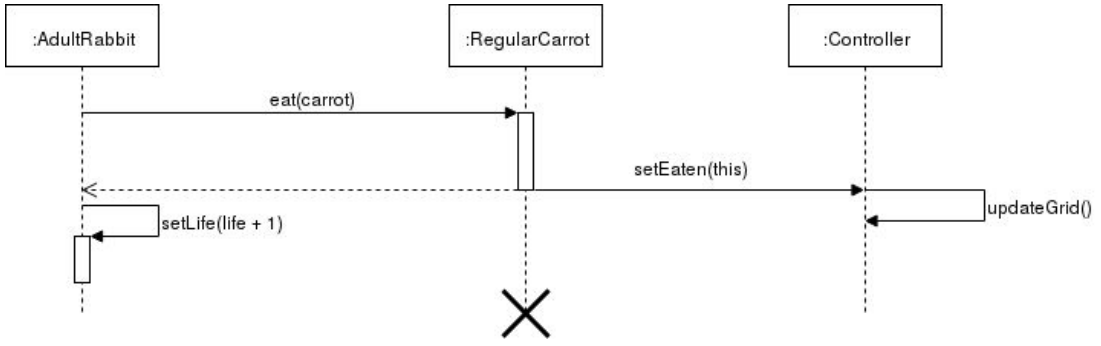


Diagramme de séquence associé au cas d'utilisation FL2 (grandir)

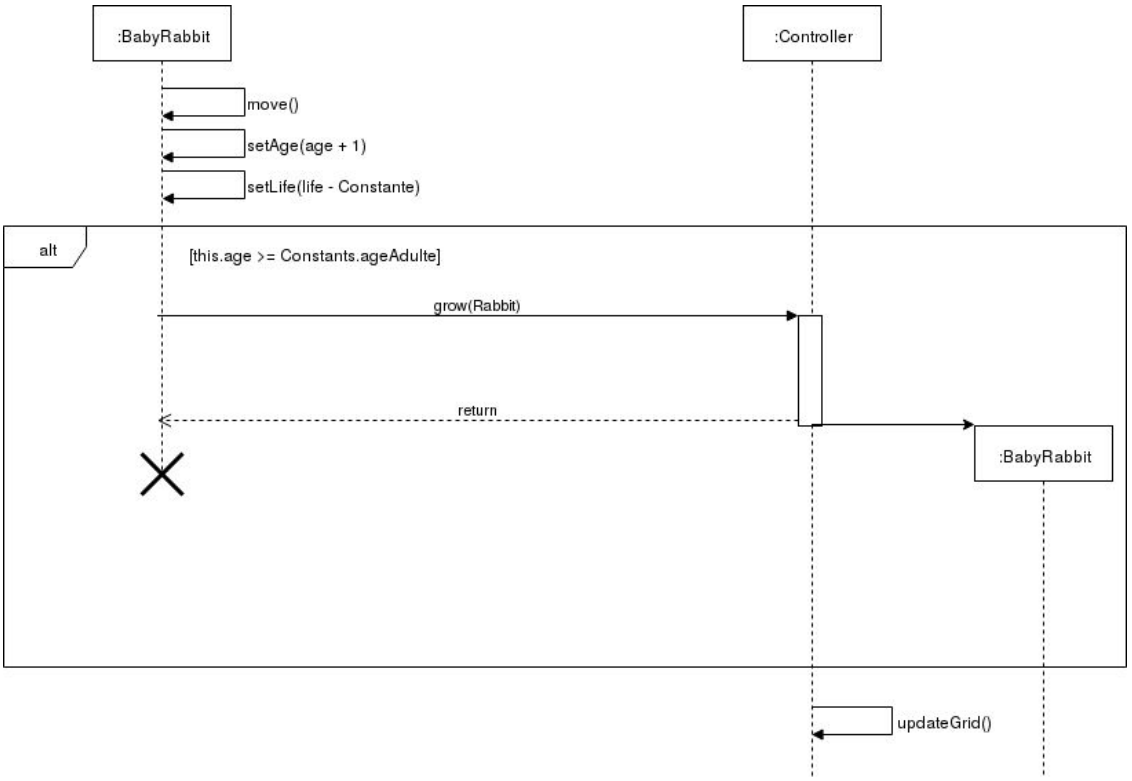


Diagramme de séquence du cas d'utilisation FC1 (pousser)

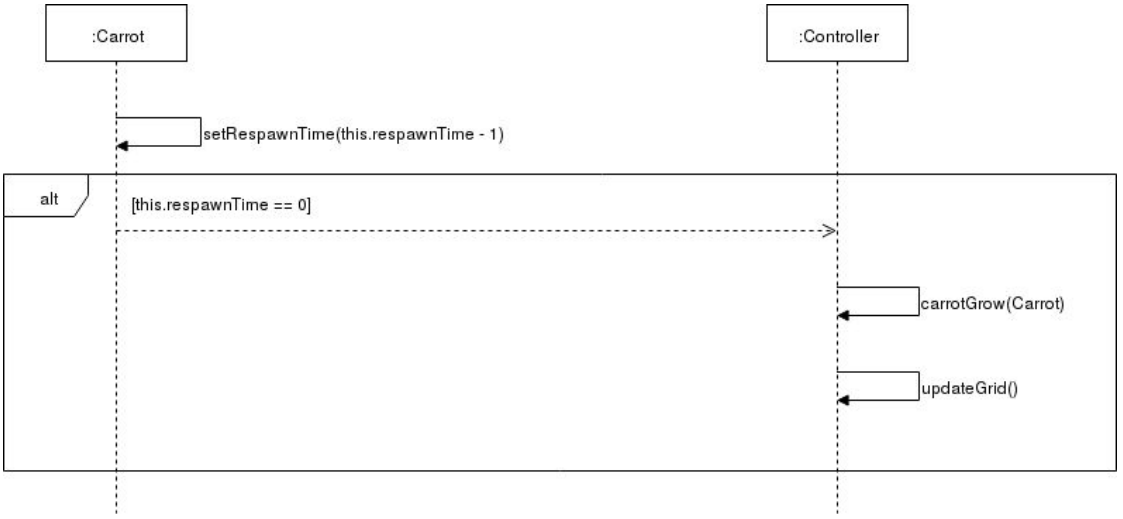


Diagramme de séquence du cas d'utilisation FL1 et FL4 (naître, se reproduire)

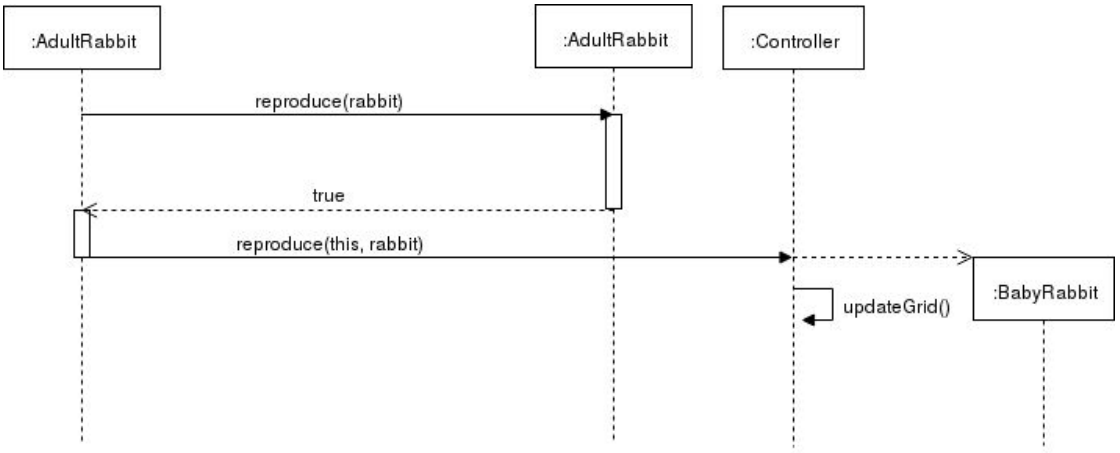
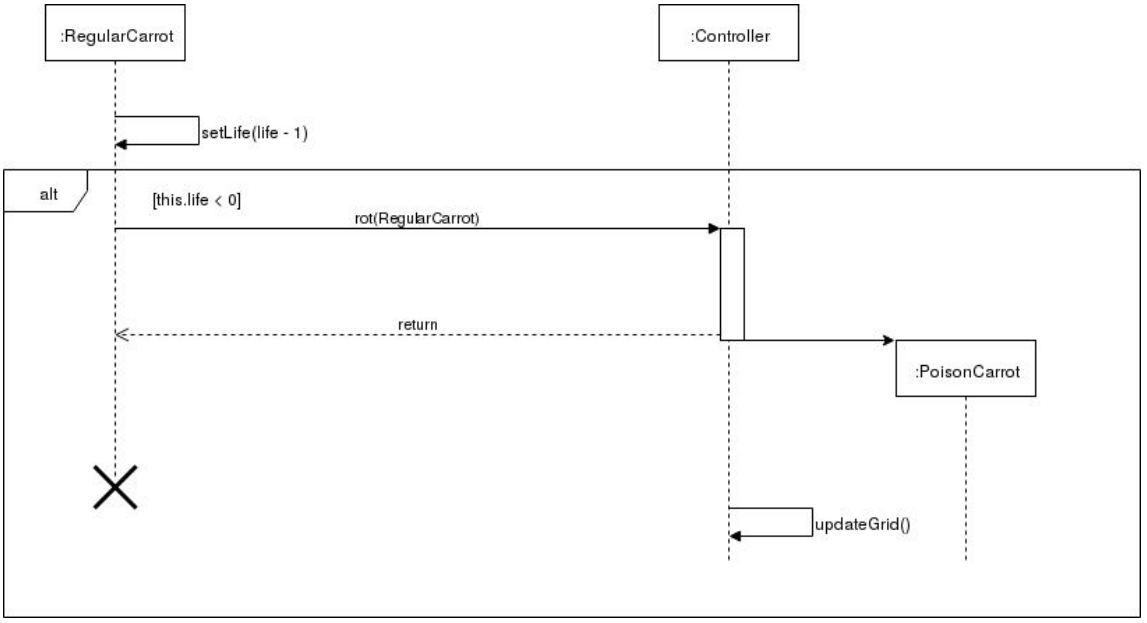


Diagramme de séquence du cas d'utilisation FC3 (pourrir)



Le reste du jeu repose simplement sur le déplacement des lapins sur le champ à chaque "unité de temps" (dans notre cas il s'agit d'un tour de jeu, c'est-à-dire à chaque actualisation de la grille). Chaque cas d'utilisation apparaîtra dans le cahier de test.

3.4 Modélisation

Le jeu est découpé en quatre modules différents. De manière générale, un pour les acteurs, un pour le fonctionnement du jeu, un pour l'exécutable et un pour l'IHM. Ces derniers se découpent comme tel:

3.4.1 Package gameActors

❖ Les lapins : Le lapin est une classe nommée Rabbit. Cette classe possède un héritage. Un lapin peut être :

- un lapin adulte AdultRabbit qui peut manger, se reproduire et se déplacer
- un bébé lapin BabyRabbit qui ne peut que se déplacer et manger.

Un bébé devient adulte au bout d'un certains temps (8 tours de jeu). Les lapins auront un attribut de sexe qui sera choisi aléatoirement lors de la naissance (l'instanciation) du lapin.

A chaque tour, un Rabbit perd de la vie au moment de se déplacer, traduisant l'accentuation de sa faim. Un bébé perd de la vie plus rapidement qu'un adulte.

Attribut de Rabbit *:*

- id (int) : identifiant d'un lapin utilisé pour le débuggage.
- age (int) : age du lapin pour différencier adulte et bébé et savoir quand faut-il faire grandir un bébé.
- sex (int) : booléen indiquant le genre du lapin.
- life (int) : représente les points de vie du lapin. Ils varient à chaque tour selon les actions qui s'y passent (déplacement, ingestion de carotte normale, ingestion de carotte empoisonnée)

Fonctions associées à Rabbit *:*

- eatCarrot(RegularCarrot) : void : *fonction abstraite.* Mange la carotte passée en paramètre et augmente la vie du lapin.
- eatCarrot(PoisonCarrot) : void : mange la carotte passée en paramètre et fait mourir le lapin.
- setLife(int) : void : affecte l'entier passé en paramètre en tant que vie du lapin et le fait mourir si la valeur est inférieure ou égale à zéro.
- reproduce(Rabbit): boolean : indique si le lapin peut se reproduire avec celui passé en paramètre.
- move() : void : *fonction abstraite.* Fait se déplacer le lapin sur une case adjacente aléatoire à sa position actuelle. Elle modifie les données membres d'adultRabbit et babyRabbit différemment puis choisit aléatoirement une case dans laquelle un lapin peut se déplacer au tour suivant à partir de sa position actuelle.

Fonctions associées à AdultRabbit *:*

- reproduce(AdultRabbit) : si c'est possible (sexes opposés), notifie le Controller que ce lapin veut se reproduire avec celui passé en paramètre.
- eatCarrot(RegularCarrot) : void : augmente la vie du lapin de 3 points.
- move() : void : un lapin adulte perd 1 point de vie lors d'un déplacement.

Fonctions associées à BabyRabbit *:*

- eatCarrot(RegularCarrot) : void : augmente la vie du lapin de 8 points.
- move() : void : un lapin adulte perd 2 points de vie lors d'un déplacement.

❖ Les carottes : La carotte est une classe nommée Carrot. Cette classe possède un héritage.

Une carotte peut être:

- une carotte normale RegularCarrot qui est simplement mangée
- une carotte empoisonnée PoisonCarrot qui tue le lapin dès qu'elle est mangée.

Attributs de Carrot *:*

- id : int : identifiant d'une carotte utilisé pour le débuggage.
- respawnTime : int : le temps de repousse restant à la carotte. Une fois écoulé, elle apparaît sur la grille.

Fonctions associées à la classe Carrot *:*

- setEaten() : void : *fonction abstraite.* Notifie le Controller que la carotte a été mangée.

Attributs de RegularCarrot *:*

- life : int : vie restante à la carotte.

Fonctions associées à RegularCarrot *:*

- setEaten() : void : réaffecte à la carotte le temps de repousse d'une RegularCarrot (spécifié dans la classe statique Constants) qui sera utilisé dans le Controller.
- setLife(int) : void : affecte l'entier passé en paramètre en tant que vie de la carotte et la fait pourrir si la valeur est inférieure ou égale à zéro.

Fonctions associées à PoisonCarot *:*

- setEaten() : void : réaffecte à la carotte le temps de repousse d'une PoisonCarrot (spécifié dans la classe statique Constants) qui sera utilisé dans le Controller.

❖ Dirt : Classe symbolisant une case vide.

3.4.2 Package gameEngine

❖ Le contrôleur : Classe nommée Controller qui représente le contrôleur de jeu. Elle a pour rôle de gérer les tours, distribuer les différents évènements aux différent acteurs de la grille et mettre à jour la grille en conséquence. C'est cette classe qui gère les lapins adultes, les lapins bébés, les lapins morts, les carottes mangées, les carottes qui repoussent, les carottes qui pourrissent... etc. Il s'agit de la classe qui gère les règles de vie du jeu. Par conséquent, elle implémente le design pattern Singleton car il n'y a besoin que d'une instance pour gérer le jeu.

Fonctions :

- rabbitSpawn(boolean, int, int) : void : fait apparaitre un lapin dans le jeu.
- carrotGrowth(boolean, int, int) : void : naissance d'une carotte dans le jeu.
- init(boolean ihm) : void : invite l'utilisateur à paramétrer le jeu et initialise la grille en conséquence ou lance la fenêtre de saisie en mode IHM.

```
public void init(boolean ihm) {
        if(!ihm)
                *saisis du nombre de lapins adultes initial
                *saisis du nombre de carottes normales initial
                *saisis du nombre de carottes empoisonnées initial
                *initialisation de la map
```

```
            else
                    *lancer la fenêtre de saisie
    }
```

- kill(Rabbit) : void : tue le lapin passé en paramètre.
- setEaten(Carrot) : void : fait disparaître du jeu la carotte passée en paramètre.
- rot(RegularCarrot) : void : carotte passée en paramètre devient empoisonnée.
- nextTurn() : void : effectue toutes les modifications à faire pour le tour suivant dans le respect des règles de vie soit le déplacement des lapins et la disparition et apparition de certains éléments en conséquence. Lorsque tous les lapins sont morts le jeu est fini et un message de fin de jeu est affiché dans la console. L'utilisateur peut alors relancer une nouvelle partie en appuyant sur entrée. Cette information est affichée à l'utilisateur dans la console. (version IHM expliquée p 13).

```
public void nextTurn() {
    tant_que(Lapins vivants) {
        *création d'un tableau pour les traitements communs entre lapins adultes et lapins bébés
        *déplacement des lapins :
            - possible passage à l'âge adulte
            - possible de consommation des carottes (empoisonnées ou non) + possible mort
              des lapins par famine ou empoisonnement
        *mise à jour de la map par rapport aux éléments précédents
        *reproduction des lapins adultes
        *repousse des carottes et vieillissement des certaines
        *mise à jour de la map pour les nouveaux lapins
    }
}
```

- grow(BabyRabbit) : void : fait passer le bébé passé en paramètre à l'âge adulte
- reproduce(AdultRabbit, AdultRabbit) : void : fait se reproduire les deux lapins et fait naître les bébés lapins.

❖ Constantes : Classe statique nommée Constants contenant toutes les constantes utilisées dans le code source.
Attributs :

- mapWidth : int : longueur de la grille de jeu.
- mapHeight : int : hauteur de la grille de jeu.
- adultAge : int : âge adulte pour un bébé lapin.
- carrotLife : int : vie d'une RegularCarrot qui vient de pousser.
- respawnRegularCarrot : int : temps de repousse d'une RegularCarrot.
- respawnPoisonCarrot : int :temps de repousse d'une PoisonCarrot.

❖ GameElement : Classe abstraite des éléments du jeu. Représente un objet contenu dans une Cell dans la grille de jeu localisé grâce à ses indices de ligne et de colonne.
Attributs :

- posLi : int : position en ligne du GameElement dans la grille.
- posCo : int : position en colonne du GameElement dans la grille.

❖ Displayable : Interface qui permet un affichage console.

Fonctions :
- display() : void : affichage de l'objet dans la console.

❖ Grid : Cette classe contient les caractéristiques de la grille.
Attributs :
- li : int : nombre de lignes de la grille.
- co : int : nombre de colonnes de la grille.
- cells[][] : Cell : la grille elle-même.

Fonctions :
- display() : void : fonction d'affichage dans la console.

❖ Cell : Cette classe contient les caractéristiques d'une cellule.
Attributs :
- empty : boolean : indication si la case est occupée par un lapin ou pas.
- content : GameElement : contenu de la case.

Fonctions :
- isEmpty() : boolean : indique si la case est occupée par un lapin ou pas.

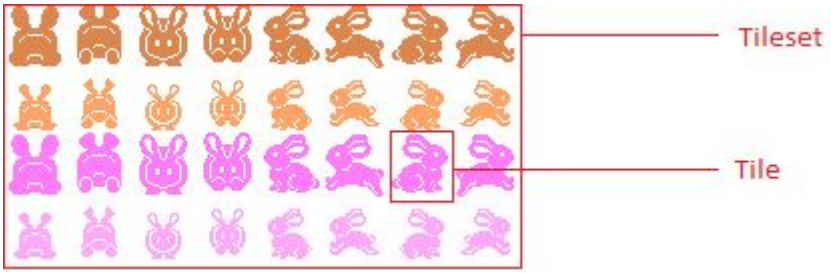
3.4.3 Package exe

❖ Client : contient la fonction main du programme.

3.4.4 Package gameInterface

Le package gameInterface contient toutes les classes liées à l'IHM, exigence non fonctionnelle du projet et donc optionnelle du projet. L'équipe a néanmoins eu le temps de l'implémenter. Voici les différentes informations à propos de ce package.

Terminologie :
Tileset : image globale composée de sprites de même dimension (=> la dimension du tileset est un multiple de la dimension d'un Tile).
Tile : zone spécifique du Tileset représentant un élément visuel du jeu correspondant à un sprite.
Un tile peut se traduire par un crop (extraction d'une portion) du Tileset.



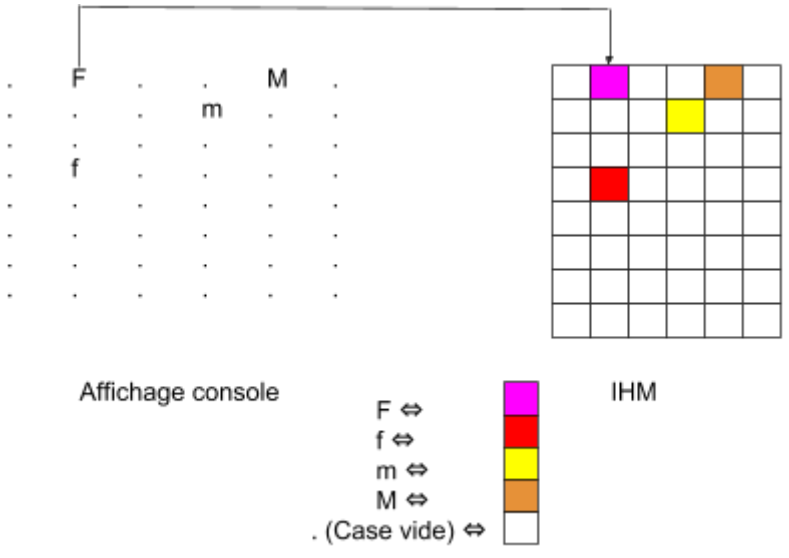
Tileset créé exceptionnellement pour le projet par les membres de l'équipe

Classes :
❖ WindowParameters : Fenêtre (JFrame de l'API Swing/AWT de Java) de saisie des paramètres contenant des zones de saisie dans lesquelles l'utilisateur

indique le nombre d'acteurs désiré. Les données ainsi saisies seront prises en compte pour initialiser la grille de jeu en appelant la fonction init(ihm) de Controller avec ihm = *true* (cf package gameEngine p10). *Non mise dans le diagramme de classe par manque de place.*

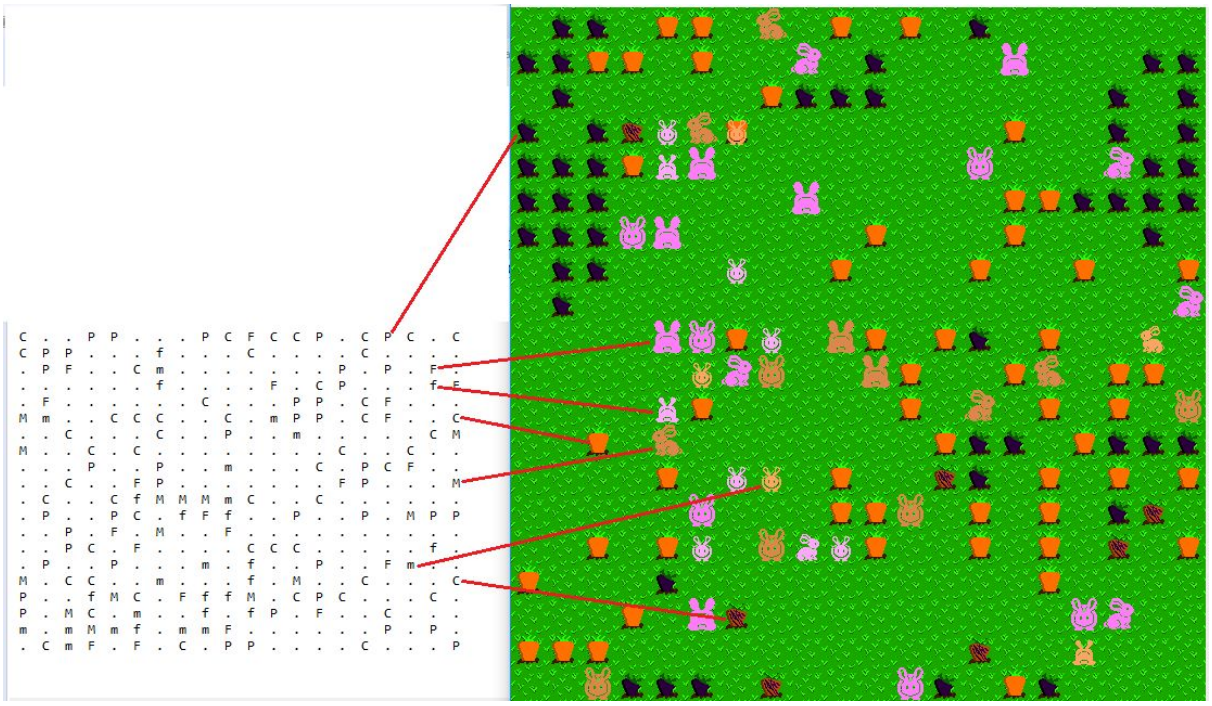
- Tileset (interface) : représente un Tileset.
- CharsetRW : Tileset des lapins. Implémente l'interface Tileset *(Non mise dans le diagramme de classe par manque de place, le principe de fonctionnement étant illustré par l'interface Tileset).*
- TilesetRW : Tileset des éléments statiques du jeu (herbe pouvant représenter les cases vides, carottes et carottes empoisonnées). Implémente l'interface Tileset *(Non mise dans le diagramme de classe par manque de place, le principe de fonctionnement étant illustré par l'interface Tileset).*
- Tile : Gestion des tiles d'un Tileset. Le tileset correspondant sera une donnée membre de la classe. Cette classe possède la fonction drawTile appelée par les fonctions draw des GameElements. Elle s'occupe de dessiner le Tile sur la Map.
- Window : Fenêtre de jeu contenant la Map.
- Map : Panneau (JPanel de l'API Swing/AWT de Java) représentant la grille. Utilise la grille du Controller pour faire un affichage basé sur le même principe que l'affichage console. A ceci près que l'affichage console utilise des caractères pour afficher une case alors dans l'IHM une case est représenté par un Tile.

Equivalence entre l'affichage de la grille et l'IHM



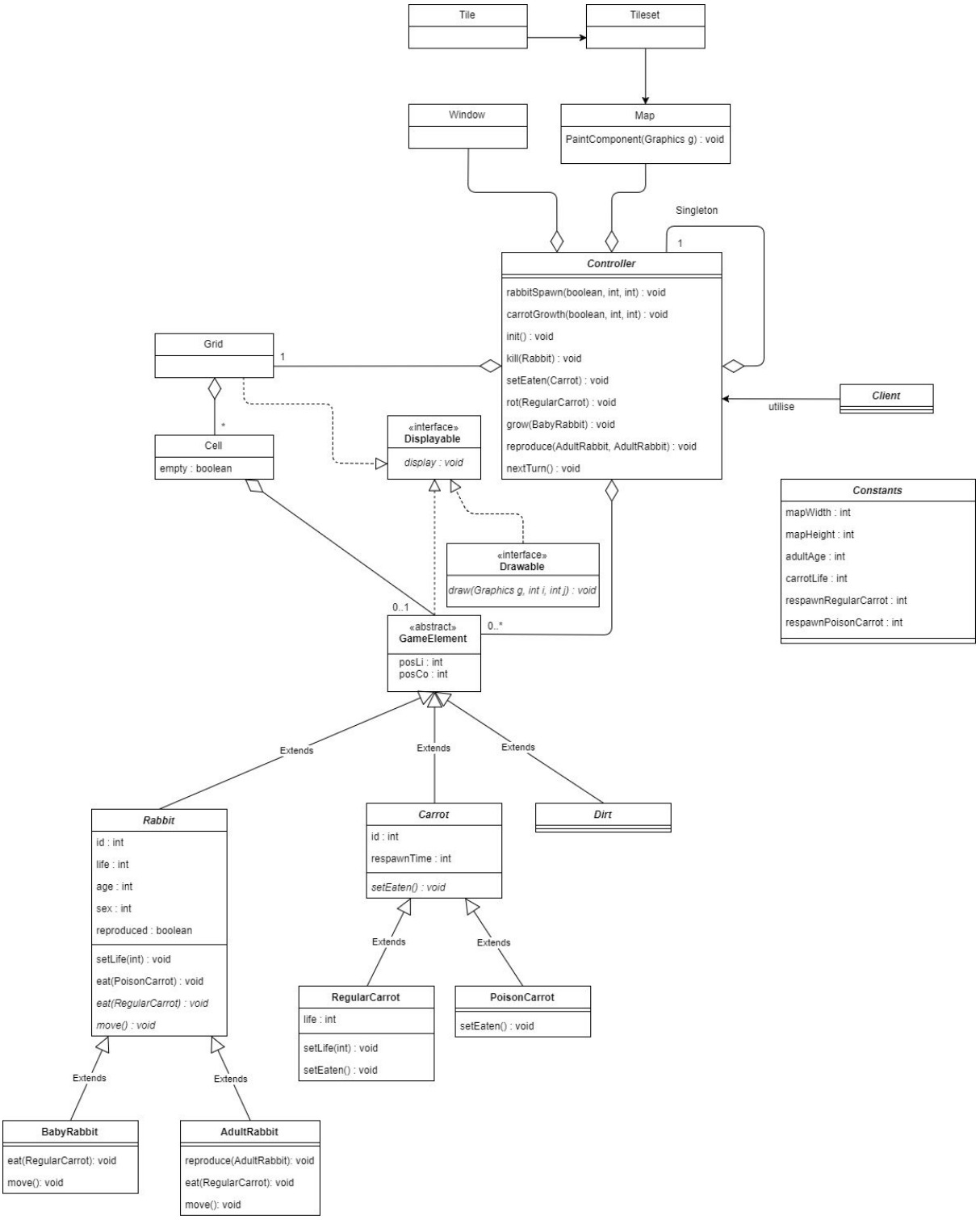
Ce schéma ne sert qu'à montrer comment se fait l'équivalence entre les 2 modes d'affichage. Les tilesets réellement utilisés sont des tilesets créés exclusivement par les membres pour le projet (comme celui montré plus haut).

Exemple concret (Il s'agit de 2 parties lancées séparément)



Lors de la fin de jeu, la map est figée et mise en transparence. Au premier plan un message de fin de jeu est affiché. Pour relancer une nouvelle partie, le joueur ferme la fenêtre de jeu et retourne dans la fenêtre de saisie (encore ouverte) pour initialiser à nouveau une partie.

Diagramme de classe du projet RabbitWorld


Tile
Tileset
Window
Map
PaintComponent(Graphics g) : void
Singleton
1
Controller
rabbitSpawn(boolean, int, int) : void
carrotGrowth(boolean, int, int) : void
init() : void
kill(Rabbit) : void
setEaten(Carrot) : void
rot(RegularCarrot) : void
grow(BabyRabbit) : void
reproduce(AdultRabbit, AdultRabbit) : void
nextTurn() : void
Grid
1
Client
utilise
*
Cell
empty : boolean
«interface»
Displayable
display : void
Constants
mapWidth : int
mapHeight : int
adultAge : int
carrotLife : int
respawnRegularCarrot : int
respawnPoisonCarrot : int
«interface»
Drawable
draw(Graphics g, int i, int j) : void
0..1
0..*
«abstract»
GameElement
posLi : int
posCo : int
Extends
Extends
Extends
Rabbit
id : int
life : int
age : int
sex : int
reproduced : boolean
setLife(int) : void
eat(PoisonCarrot) : void
eat(RegularCarrot) : void
move() : void
Carrot
id : int
respawnTime : int
setEaten() : void
Dirt
Extends
Extends
RegularCarrot
life : int
setLife(int) : void
setEaten() : void
PoisonCarrot
setEaten() : void
Extends
Extends
BabyRabbit
eat(RegularCarrot): void
move(): void
AdultRabbit
reproduce(AdultRabbit): void
eat(RegularCarrot): void
move(): void

3.5 Détails des différents états des objets

Les différents objets passent par plusieurs états décrits par les diagrammes ci-dessous. Ces derniers permettent de mettre en valeur leurs états et les actions à apporter pour les faire changer d'états.

Diagramme d'états-transitions pour les lapins :

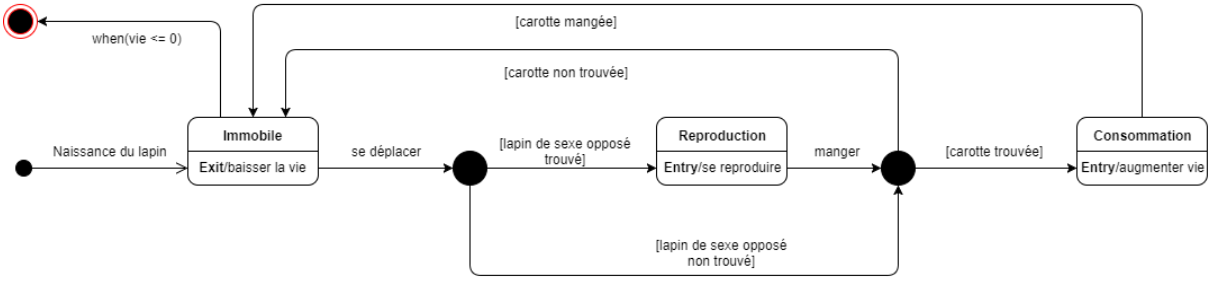
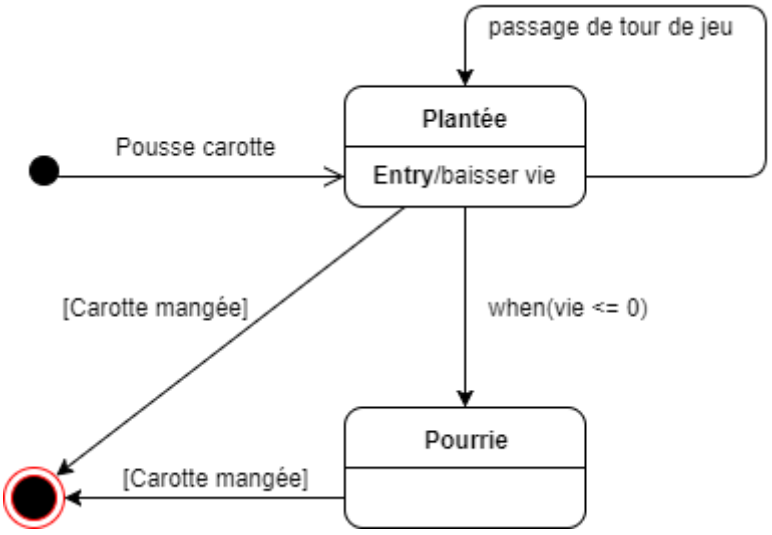


Diagramme d'états-transitions pour les carottes :



3.6 Techniques utilisées

L'une des exigences fonctionnelles du client est d'avoir un jeu de la vie ayant une modélisation orientée objet particulièrement bien agencée. Il est décrit plus haut la modélisation des objets, ci-dessous est décrit différents mécanismes orientés objet intervenant au sein du projet pour une meilleure ergonomie.

- Le polymorphisme et la liaison de données sont présents dans le projet à travers les classes Rabbit et Carrot pour respectivement la quantité de points de vie perdue lors des déplacements (les bébés arrivent en famine plus rapidement que les adultes) et les interactions avec les lapins lors de la consommation de la carotte (augmentation de la vie ou mort du lapin selon si normale ou empoisonnée).

- La classe Controller implémente le design pattern Singleton car il n'y a besoin que d'une seule instance du contrôleur pour gérer le déroulement du jeu. De même pour les Tilesets (CharsetRW et TilesetRW).
- La structure du projet suit une architecture MVC :
    - les modèles étant les acteurs (package gameActors)
    - les vues étant de façon minoritaire l'affichage console (au final relayé au simple rôle de debugger) et surtout l'IHM (package gameInterface)
    - le contrôleur étant la classe Controller (package gameEngine)

4. Exigences du client

4.1 Interface Homme / Machine

Le jeu est ouvert à tout type d'utilisateur. Il peut s'agir de l'utilisateur lambda étant seulement au courant du principe du jeu et qui cherche à le découvrir visuellement. Ce type d'utilisateur a une connaissance normale du système d'exploitation sous lequel sa machine fonctionne. Ou il peut aussi être lancé par un type d'utilisateur plus expérimenté, connaissant le principe du jeu ainsi que son fonctionnement et pouvant deviner quelles solutions techniques ont potentiellement pu être mises en application pour le réaliser.

Dans tous les cas, il n'est pas nécessaire de restreindre et privilégier l'accès selon les profils d'utilisateurs.

Le jeu peut se lancer et s'exécuter sur un terminal. Dans ce cas, la représentation des différents acteurs est très limitée. Un caractère différent est affiché selon l'acteur représenté : "F"/"M" pour représenter un lapin adulte femelle/mâle et "f"/"m" pour les bébés lapins femelle/mâles, un "C" pour les carottes et un "P" pour les carottes empoisonnées. En début de partie, l'utilisateur aura des données à saisir dans la console concernant le nombre de lapins et carottes initialement présents sur le terrain.

Le jeu peut également, et c'est ce qui est le plus intéressant pour l'utilisateur, se lancer grâce à une interface graphique. Cette dernière possède une fenêtre de paramétrage où l'utilisateur doit entrer les nombres de départ de lapins, carottes et carottes empoisonnées. En cas d'erreur, une autre fenêtre s'ouvre et demande à l'utilisateur de vérifier ses paramètres, il n'y a pas de limites dans le nombres d'erreurs de saisie que l'utilisateur peut faire. Ensuite, une interface graphique se lance représentant un champ avec des lapins de différentes couleurs selon leur sexe et âge, et des carottes noires (empoisonnées) ou orange (normales). A ce moment là le jeu est en pause pour laisser le temps à l'utilisateur de visualiser les positions initiales des différents acteur. En appuyant alors sur entrée, l'utilisateur lance la partie. En fin de jeu, c'est-à-dire lorsque tous les lapins sont morts, le jeu se fige à nouveau et un message "Game Over" s'affiche pour annoncer la fin de la partie.

Voir partie 2.4.4 package gameInterface pour un début de représentation graphique du jeu.

4.2 Exigences concernant la conception et la réalisation

Le jeu doit contenir au minimum des lapins se déplaçant dans un champ, mangeant des carottes et se reproduisant. Le facteur qualité principal requis est que le jeu fonctionne c'est-à-dire que l'utilisateur puisse entrer les paramètres de son choix et regarder les lapins se déplacer et manger jusqu'à la fin du jeu. Le deuxième facteur qualité qui est une exigence non-fonctionnelle mais qui est un réel point positif dans le rendu du projet est l'interface graphique. En effet, elle apporte une qualité au jeu bien supérieure.

L'objectif de l'équipe est de rendre en temps voulu les fichiers sources, les cahiers de conception et de spécifications ainsi que les différents rapport d'avancement. Une démonstration sera faites au client à la fin du dernier sprint, d'où le fait que l'interface graphique a une certaine importance dans le projet.

Le jeu fonctionnera à minima sur des ordinateur sous Unix avec Java 1.8 d'installé.

Le jeu se lance avec une fenêtre de paramétrage en début de jeu où il suffit d'entrer le nombre de lapins, le nombre de carottes et le nombre de carottes empoisonnées. Une fois le jeu lancé, ce dernier sera totalement autonome et prendra fin à la mort de tous les lapins.

5. Tests

Plusieurs problèmes peuvent apparaître lors du développement du projet à travers les tests.

Ces erreurs peuvent être liées aux différents algorithmes définissant les règles de vie du jeu :

- Déplacement non autorisé d'un lapin (extrémité de la map, collision entre 2 lapins)
- Mise à jour erronée des points de vie des acteurs à chaque tour de jeu
- Reproduction entre un lapin adultes et un lapin bébé et/ou entre deux lapins du même sexe
- Tout type de problème d'affichage lié à l'IHM

Les erreurs peuvent être autant perçues à travers l'affichage console que l'IHM.

L'objectif des tests de validation est d'assurer la conformité fonctionnelle de chaque module (gameActors, gameEngine et gameInterface) ainsi que de leurs différentes communications par rapport aux spécifications fonctionnelles et techniques. Il s'agit de parcourir l'ensemble des chemins des modules en utilisant des jeux de données pertinents (avec données valides et invalides pour s'assurer du déroulement pertinent des scénarios en toutes circonstances). Pour cela on applique des séries de scénarios correspondant aux modèles des traitements.

Les use case sont définis dans la phase de spécification du projet. La nature de notre projet, le jeu de la vie, se prête peu au terme de use case dans la mesure où il y a peu d'interactions avec l'utilisateur (hormis pour la saisie des acteurs initiaux qui sont dans tous les cas nécessaires pour l'exécution de chaque test). Lorsqu'elles ont

été établies, nos fiches de tests vérifiaient donc plus la validité des fonctionnalités référencées dans les dossier de spécification et conception que les use case dont la vérification était intégrée dans les scénarios de chaque test de fonctionnalité.

Les différentes fonctionnalités impliquent un certain nombre de risques et doivent donc être testées comme indiqué dans les fiches de test.

6. Les difficultées rencontrées

- L'une des principales difficultées rencontrées, qui a nécessité beaucoup de temps de réflexion à l'équipe, a été d'adapter l'API du jeu développé à l'API Java/Swing/Awt. En effet, certaines règles régissent les API déjà présentes et il a fallu développer le projet en tenant compte de ces règles.

  L'exemple le plus représentatif de ce problème à été un bug concernant l'interface graphique du jeu. Le champ (la grille IHM) ne s'affichait pas et il fallait redimensionner la fenêtre de jeu pour forcer un appel à la fonction repaint() de l'API Swing/Awt pour l'obliger à s'afficher. Ceci était dû au fait que, de par notre API résultant de notre modélisation, notre classe Map avait un objet Graphics en donnée membre et c'était sur ce dernier qu'étaient appelées les fonctions de dessin des éléments du jeu.

  Solution : Pour régler ce problème, il a fallu ajouter des paramètres aux fonctions de dessin de notre API faisant référence au paramètre Graphics g présent dans la fonction paintComponent(Graphics **g**) de la classe Map (héritée de la classe JPanel qu'elle étend). Ainsi, à la place de manipuler une donnée membre Graphics, on manipule directement la référence du paramètre **g**.

Schéma explicatif :

Map

```
~~Graphics myG;~~

@Override
paintComponent(Graphics g) {
        …
        // un AdultRabbit dessiné aux coordonnées i, j
    ~~r.draw(i, j);~~
        r.draw(g, i, j);
        …
}
```

AdultRabbit

```
~~draw(int i, int j) {~~
draw(Graphics g, int i, int j) {
        …
        ~~Controller.getMap().myG.draw(...)~~
        g.draw(...)
        …
}
```

- Une autre grosse difficulté s'est manifestée lors de la réalisation de la fenêtre de paramétrage. En effet, il fallait ensuite initialiser la grille et il y a eu quelques difficultées pour donner les paramètres entrés dans la fenêtre à la fonction d'initialisation de la grille située dans la classe Controller puis à exécuter suivant le module exe. La difficulté a donc été de passer les bonnes données entre les classes.

  Solution : Pour pallier à ce problème, il a été mis en place des attributs correspondants aux entiers saisis dans la fenêtre, dans la classe Controller accompagnés de Setter. Dans la fonction d'initialisation, il y a donc deux cas : un pour un lancement avec IHM (avec les attributs comportant les setters) et un pour un lancement sur console. On retrouve donc les attributs des entiers des champs de saisie dans le cas avec ihm. Enfin, dans l'action listener du bouton START, les setter sont appelés et modifient les attributs en question. L'exécutable Client quant à lui, n'a pas eu de changements.

  Classe de la fenêtre des paramètres :

```
ntroller.getInstance().setIhm(true);
(rabbit_param != -1 &&
      reg_carrot_param != -1 &&
      pois_carrot_param != -1 &&
      sumActors <= Constants.getMapWidth() * Constants.getMapHeight()) {
  try {
      Controller.getInstance().setNb_field_rabbits(rabbit_param);
      Controller.getInstance().setNb_field_reg_carrots(reg_carrot_param);
      Controller.getInstance().setNb_field_pois_carrots(pois_carrot_param);
      Controller.getInstance().init(true);
      Controller.getInstance().setGameInited(true);
      Controller.getInstance().setGameStarted(false);
      JFrame win = Controller.getInstance().getWindow();
```

  Classe Controller partie ihm = true :

```
  } else {
      for(int i = 0; i < nb_field_rabbits; i++) {
          placed = false;
          do {
              rli = this.random.nextInt(this.grid.getLi());
              rco = this.random.nextInt(this.grid.getCo());
              if(this.grid.getCells()[rli][rco].isEmpty()) {
                  this.rabbitSpawn(true, rli, rco);
                  placed = true;
              }
```

7. Conclusion

Le projet répond aux exigences fonctionnelles du client. Le jeu fonctionne sur une interface graphique et correspond à un jeu de la vie avec des lapins. Le projet suit bien une modélisation objet et utilise des concepts objets.

La méthode agile a été plutôt instinctive à mettre en place, et la composition des tâches dans les sprint n'a pas semblait difficile à mettre en place pour l'équipe. Il n'y a pas eu de désaccords majeurs au sein de l'équipe, ni même mineurs.

Plusieurs petites difficultées ont été rencontrées. Les difficultées principales majeures qui ont occupé beaucoup de temps à l'équipe et ont risquées de faire prendre du retard sont répertoriées ci-dessus (6. Les difficultés rencontrées). Ces dernières ont été réglées.

La communication avec le client, du point de vu de l'équipe, s'est bien passée. Les contacts par mail et grâce aux rapports d'avancement et de conception ont été suffisant. De plus, le client pouvait suivre la programmation via le lien GitHub.

8. Annexes

Les rapports d'avancements :

Derniers Sprints (15/01/2018 - 13/01/2018)

Ce que l'on a conçu :

Le projet a été terminé et les derniers sprints et la version finale du dossier de conception a été réalisée. Le résumé des derniers sprints ainsi que le déroulement général du projet y sont répertoriés.

Ce que l'on a programmé :

Nous avon principalement fini de programmer l'IHM dans ce dernier sprint. Nous avons amélioré quelques détails dans le reste du jeu.
Alexandre : la fenêtre de jeu
Isis : la fenêtre des paramètres
Beaucoup de tâches ont ensuite été réparties au fur et à mesure dès leur apparition.

Ce que nous avons testé :

Nous avons tout testé. Soit les fonctions que nous avons codés et l'ensemble du jeu. Nous avons essayé différents jeux de paramètres et avons en conséquence amélioré le jeu si nécessaire.

Ce qui a été dit/vu durant le sprint :

Les difficultés majeures sont apparues durant ce sprint (décrites dans le cahier de conception partie 6). Une bonne communication, régulière au sein de l'équipe a permis de pallier les problèmes.

Conclusion :

Le jeu est fini et fonctionne. L'utilisateur peut y jouer aisément. L'équipe aurait aimé prendre le temps d'améliorer l'IHM et peut-être d'ajouter encore différentes règles de vie dans le but d'améliorer le projet.

Sprint 2 (18/12/2017 - 15/01/2018)

Objectifs du sprint 2 :

Fin d'implémentation des règles de vie.
Amélioration et approfondissement du cahier de conception.

Ce qu'on a conçu :

Le package gameActors contenant les acteurs et le package gameEngine qui contient principalement les règles de vie (grâce notamment à l'ajout d'un contrôleur) valident les exigences fonctionnelles pour les acteurs. Quelques tests sur chaque fonctions grâce à un affichage dans la console ont été réalisés mais doivent être approfondis.
La partie composée des exigences fonctionnelles est terminée. A présent, nous commençons les parties "optionnelles" notamment l'IHM.

Ce qu'on a programmé :

Nous avons choisi de répartir le travail selon les fonctionnalités plutôt que les classes entre les membres de l'équipe pour que chacun ait connaissance de l'ensemble du projet.
Isis : Gestion reproduction des lapins, passage à l'âge adulte pour les bébés lapins.
Alexandre : Déplacement, vieillissement, repousse et consommation des carottes (empoisonnées ou non), mort des lapins.
Les autres classes ont été continuellement mises à jour par les deux membres du groupe en fonction des différents messages échangés avec le contrôleur.

Ce qu'on a testé :

Via l'affichage console, nous avons testé le déplacement des lapins, leur reproduction, le fait que les carottes (empoisonnées ou non) sont mangées. Nous avons donc testé toutes les règles de vie grâce à un affichage à chaque tour de la grille dans la console.

Ce qu'on a fini :

Nous avons fini les exigences fonctionnelles soit les règles de vie, les acteurs et le contrôleur (donc le jeu tourne convenablement) mais sans IHM.

Ce qui a été dit/vu durant le sprint 2 :

La plus grosse difficulté a été l'implémentation totale du contrôleur qui est composé des fonctions les plus difficiles, qui gère toutes les règles de vie, et qui a le plus de communications avec les classes du package gameActors.

Tâches à réaliser :

Mise en place de l'IHM.
Plusieurs sessions de tests.
Trouver des jeux de données pour la démonstration dans le but d'illustrer chaque fonctionnalités.
Approfondissement du cahier de conception.

Conclusion :

Tous les objectifs ont été atteints. Nous avons empiété et commencé le sprint 3 en commençant l'IHM ce qui nous permet de prendre un peu d'avance.
Prochain sprint : 15/01/2018-Date de rendu du projet

Sprint 1 (04/12/2017 - 18/12/2017)

Objectifs du sprint 1 :

Mise en place du cahier de conception.
Programmation des acteurs (lapins) et de la grille.
Tests.

Ce qu'on a conçu :

En se répartissant les tâches, nous avons élaboré un cahier de conception. Nous avons créé les diagrammes et leurs description. Nous sommes en train de réfléchir sur une nouvelle classe “contrôleur de jeu”. En effet, nous avons que des classes représentants des acteurs comme les lapins mais qui ne peut pas créer en soit un lapin dans le jeu. Elle prendrait en charge notamment la disparition des lapins et des carottes ou leur apparition.

Ce qu'on a programmé :

Nous avons programmé les acteurs (carottes et lapins) fait par Isis, ainsi que l’élaboration d’un semblant d’API personnalisée pour le projet et la map fait par Alexandre. Dans le sprint, il était prévu de réaliser les lapins et la map. Nous sommes légèrement en avance sur la programmation.

Ce qu'on a testé :

Grâce à une sorte d’affichage dans la console, nous avons testé les fonctions de la map qui tournent correctement.

Ce qu'on a fini :

Nous pensons avoir fini la map. Cependant, comme nous réfléchissons encore à une nouvelle classe, nous n’avons pas terminé le reste.

Ce qui a été dit/vu durant le sprint 1 :

La plus grosse difficulté de ce sprint a été l’apprentissage de git.
Nous avons commencé le premier sprint du projet et nous avons programmé les acteurs et la grille.
Le cahier de conception et le rapport d’avancement est donné au client à la date du 18/12/2018.

Tâches à réaliser :

Mise en place des interactions entre objets.
Amélioration du cahier de conception.

Conclusion :

Tous les objectifs ont été atteints. Nous avons empiété et commencé le sprint 2 en réalisant les acteurs ce qui nous permet de prendre un peu d’avance.

Prochain sprint : 18/12/2017-08/01/2018