Rapport final

Jeu des Lemmings

Image22.png

ENSEEIHT

Groupe 5

Thebault Mark, Teixeira Antony, Popineau Paul

Table des matières

1ère partie : Analyse

[1. Diagramme de cas d'utilisation 1](#_Toc328008445)

[a. Cas Général 1](#_Toc328008446)

[b. Voir Jeux 3](#_Toc328008447)

[c. Jouer 4](#_Toc328008448)

[2. Diagramme de classes 5](#_Toc328008449)

[a. Analyse 5](#_Toc328008450)

[3. Diagramme de séquence 6](#_Toc328008451)

[a. Contextuel Partie simple (un lemmings devient bloquant et explose) 6](#_Toc328008452)

[b. Affectation d'une aptitude à un lemming 7](#_Toc328008453)

[c. Charger un niveau valide 8](#_Toc328008454)

[4. Maquette de l’interface utilisateur 9](#_Toc328008455)

[a. Maquette 9](#_Toc328008456)

[5. Méthode de développement 9](#_Toc328008457)

[6. Planning et tâches 9](#_Toc328008458)

[a. Planning prévisionnel 9](#_Toc328008459)

[b. Tâches et répartition 9](#_Toc328008460)

2ème partie : Conception

[1. Le modèle 11](#_Toc328008461)

[a. Gestion des deplacements 11](#_Toc328008462)

[b. Architecture et affectation des aptitudes 11](#_Toc328008463)

[Architecture des aptitudes 11](#_Toc328008464)

[Affectation des aptitudes 12](#_Toc328008465)

[c. Gestion du niveau 13](#_Toc328008466)

[d. Architecture de la carte et des cases 13](#_Toc328008467)

[e. Encapsulation par niveau, et par partie 13](#_Toc328008468)

[2. Les vues 14](#_Toc328008469)

[Vue du modèle (carte et lemmings) 14](#_Toc328008470)

[Vue du contrôleur (aptitudes, contrôles du temps). 14](#_Toc328008471)

[3. Les contrôleurs 14](#_Toc328008472)

[Le contrôleur de jeu 14](#_Toc328008473)

[Le contrôleur de partie 14](#_Toc328008474)

[Le contrôleur de niveau 15](#_Toc328008475)

[4. Interactions entre les composants 15](#_Toc328008476)

[5. Planning et tâches effectifs 15](#_Toc328008477)

[Planning Effectif 15](#_Toc328008478)

[1ere semaine: Analyse 15](#_Toc328008479)

[2ème SEMAINE: Conception et developpement 17](#_Toc328008480)

[3ème semaine: Développement et finalisation 17](#_Toc328008481)

[Tâches 17](#_Toc328008482)

[6. Difficultés rencontrées 17](#_Toc328008483)

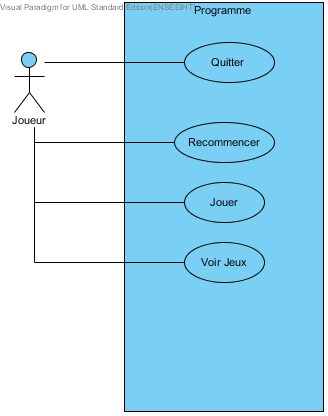
1ère partie :

Analyse

# 

# Diagramme de cas d'utilisation

## Cas Général



#### Documentation

Cas d'utilisation général du programme.

Il représente les interactions de plus haut niveau accessibles à l'utilisateur.

Jouer et Voir Jeux seront détaillées.

#### Détails

Image2.png Jouer

*Cas d’utilisation principal du jeu. Autorise l’utilisateur à manipuler les lemmings et leurs aptitudes.*

Image2.png Quitter

Ce cas définit qu’on doit être capable de quitter le jeu.

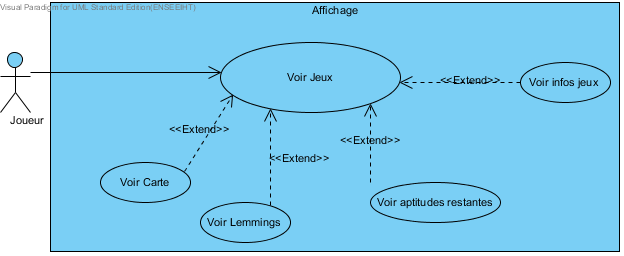
Image2.png Recommencer

Ce cas définit que l’on doit être capable de recommencer (le niveau en cours, la partie, le jeu).

Image2.png Voir Jeux

Ce cas n’est pas réellement un cas d’utilisation. Il permet de faire l’inventaire de ce que l’interface doit faire apparaitre, et éventuellement les actions que peut faire l’utilisateur sur elle.

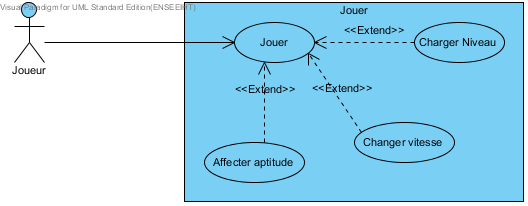
## Voir Jeux



#### Documentation

Ce diagramme permet de représenter plus en détails l'interface utilisateur et ce qu'elle doit montrer. Il n'a pas vocation et réellement décrire des cas d'utilisation, mais plutôt aider à spécifier l'interface.

## Jouer



#### Documentation

Détail du cas "Jouer". Permet de montrer les différentes actions possibles lors de la phase de jeu.

#### Détails

Image2.png Affecter aptitude

Ce cas consiste à affecter une aptitude à un lemming. Il faut d’abord cliquer sur l’aptitude à affecter, puis sur le lemming.

Image2.png Changer vitesse

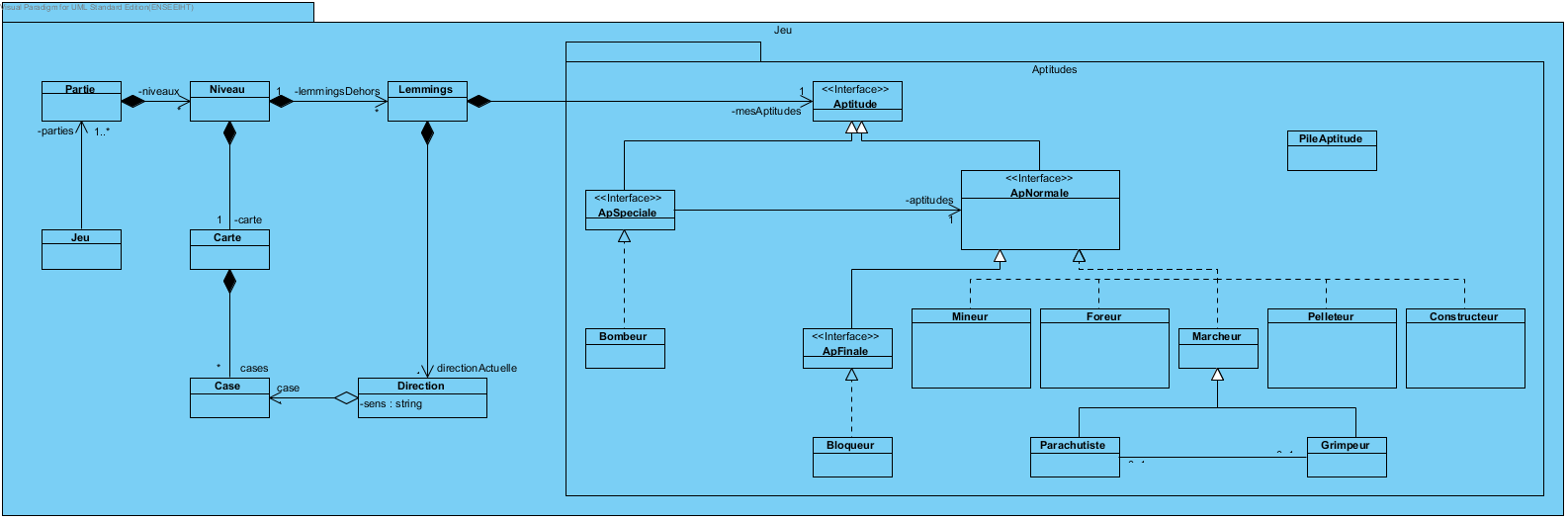
Ce cas permet à l’utilisateur de changer la vitesse du jeu pendant le déroulement de celui-ci.

Image2.png Charger Niveau

Ce cas permet au jeu de charger un fichier, dont la syntaxe explicite, définie toutes les informations nécessaires à sa création.

# Diagramme de classes

## Analyse



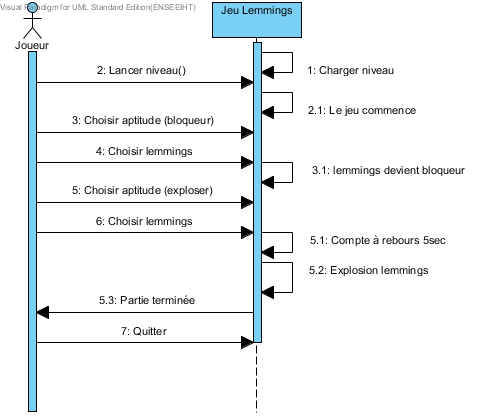
#### Détails

Diagramme de classe d'analyse représentant une esquisse aboutie des classes utilisées.

Le design pattern Stratégie a été utilisé pour les aptitudes des lemmings. De plus, nous avons ajoutés des contraintes entre elles pour montrer les interactions possibles.

# Diagramme de séquence

## Contextuel Partie simple (un lemmings devient bloquant et explose)



#### Détails

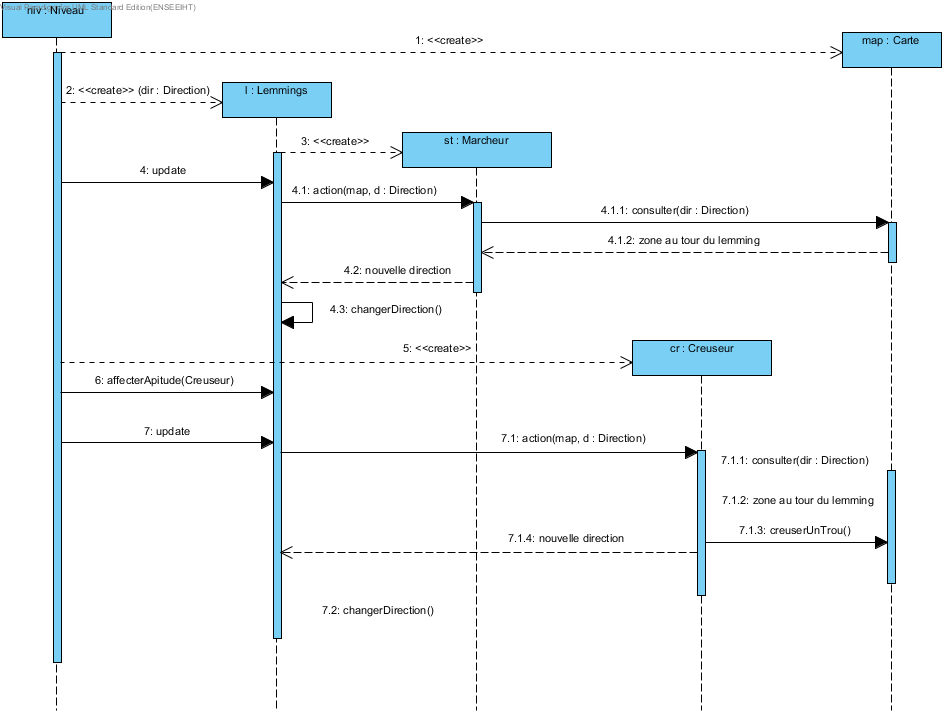
Ce diagramme représente le plus simplement possible le déroulement typique d’une partie.

Tout d’abord, au démarrage du jeu, celui charge le niveau. L’utilisateur émet ensuite l’ordre de le lancer. Le niveau démarre alors.

Pendant le déroulement du niveau, on tente d’affecter des aptitudes à un lemming (en sélectionnant d’abord l’aptitude, puis le lemming). On illustre ici le cumul d’aptitudes telles que Bloqueur et Bombeur.

Puis une fois le niveau terminé, l’utilisateur demande à quitter le jeu.

## Affectation d'une aptitude à un lemming



#### Détails

Lors de la création des lemmings, ils ont par défaut l’aptitude Marcheur. À chaque front d’horloge, le niveau se mettra à jour et mettra à jour les lemmings par le biais de la fonction update().  
Quand le lemmings se met à jour, il effectue action() sur son aptitude courante (Marcheur dans la première partie).

L’aptitude Marcheur, consulte la carte pour savoir si elle peut effectuer un déplacement et déduis le type de déplacement (marcher, monter une case, tomber …). Après le calcul du déplacement, l’aptitude indique au lemming sa nouvelle direction (Position + sens de déplacement).  
Dans certains cas il peut également indiquer au lemming s’il doit mourir, ou s’il est sorti de la carte.

Nous allons maintenant ajouter une aptitude au lemming, cela se fait par la fonction *affecterAptitude*() de lemming. Cette fonction contrôle les aptitudes que nous voulons placer sur le lemming, nous ne pouvons donc pas faire n’importe quoi. Dans notre cas, il est tout à fait possible ajouter au lemming la faculté de Creuseur.

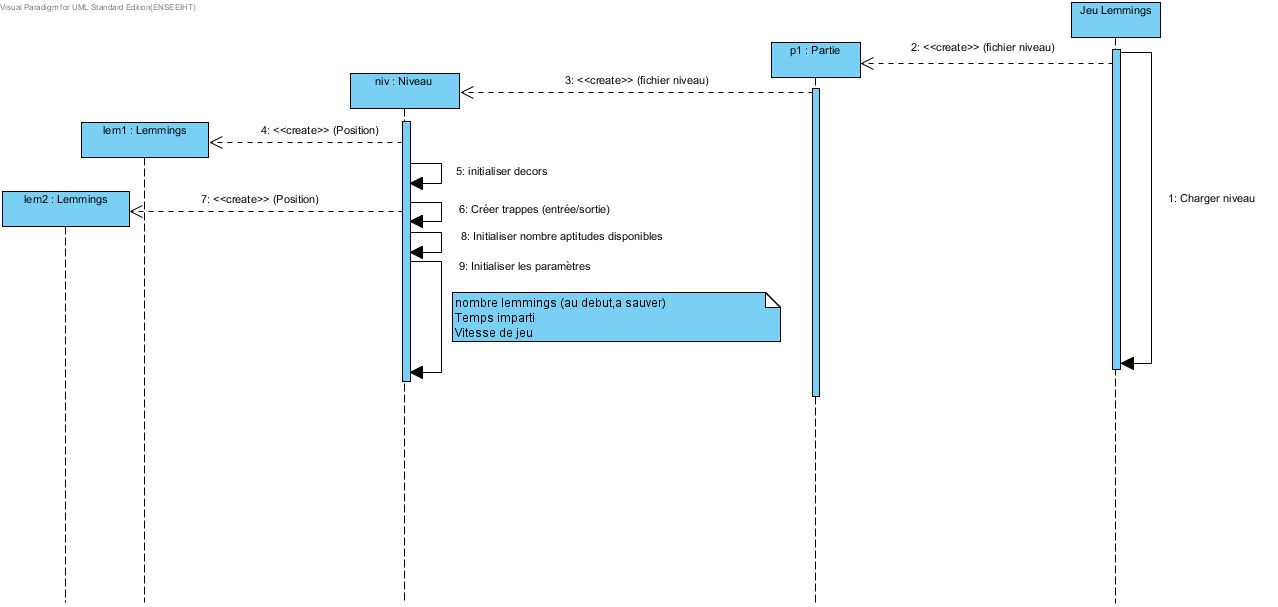
Le niveau crée l’aptitude Creuseur qui l’envoi au lemming. Le lemming vérifie s’il peut recevoir cette aptitude, pour ensuite la placer en tête des aptitudes courantes.

Lors du prochain front d’horloge, quand le lemming se met à jour, il va maintenant appeler la méthode action() de son aptitude Creuseur (car c’est l’aptitude courante).

Cette aptitude va vérifier la carte pour savoir si elle peut creuser un trou. Si tel est le cas, l’aptitude va supprimer une case de la Carte. Puis changer la position du lemming pour se déplacer vers le bas.

Dans la suite du cas traité ici, quand l’aptitude à finit de creuser, il indique au lemming de perdre cette aptitude. Marcheur, précédemment stocké, devient l’aptitude courante du lemming.

## Charger un niveau valide



#### Détails

Pour charger un niveau valide, il faut tout d’abord lancer le jeu avec un fichier en paramètre qui contiendra toutes les informations du niveau à charger.

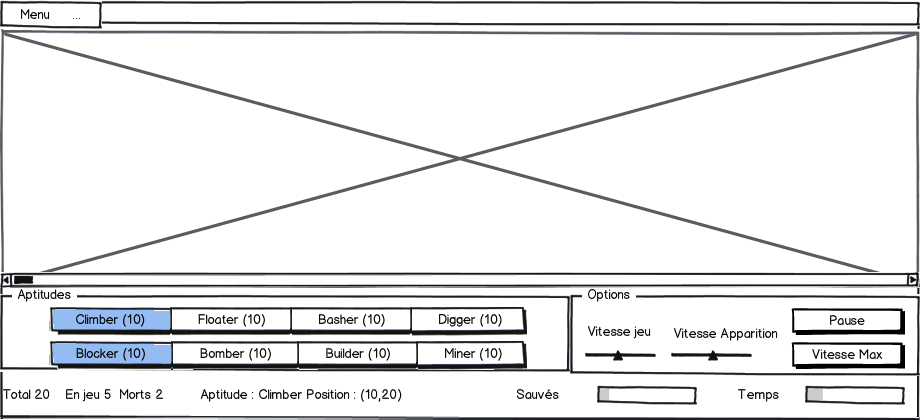
Le jeu créera par la suite une partie qui créera à son tour un niveau, toujours avec le fichier du niveau en paramètre, car c’est le niveau qui va s’occuper d’initialiser toutes les informations telles que le nombre d'aptitudes, le nombre de lemmings, ….

Ici nous n’avons pas détaillé l’initialisation de la carte avec les différentes cases mais ces actions seront déléguées à une classe (Carte).

Pour terminer nous avons décidé que ce soit le niveau qui s’occupe de la création des lemmings puisque c’est le niveau qui sait combien il doit en faire apparaître.

# Maquette de l’interface utilisateur

## Maquette



# Méthode de développement

Nous avons adopté une méthode de développement en suivant un cycle de vie par incrément afin d’avancer le projet étapes par étapes.

# Planning et tâches

## Planning prévisionnel

Tout d’abord, nous développerons une partie du modèle, en particulier les éléments de bases comme le lemming, l’aptitude « marcheur », la carte, le jeu, la partie, et les cases. Nous testerons le bon fonctionnement de notre première partie du modèle grâce à une vue console.

Puis nous implémenterons les contrôleurs et la vue pour avoir un rendu graphique et pour pouvoir tester plus rapidement les autres parties de notre modèle que nous ajouterons.

Ensuite nous avons implanterons les autres aptitudes et nous les testerons au fur et à mesure.

Pour finir nous finaliserons les contrôleurs et les vues pour un affichage plus convivial et une jouabilité accrue.

## Tâches et répartition

Le modèle est subdivisé en trois parties distinctes : le lemming et les aptitudes, la carte, et le niveau.

Chaque parties sera confiée à une personne afin que le modèle soit rapidement terminé et soit fonctionnel.

Puis, nous nous attaquerons à la vue et au contrôleur.

Nous ne spécifions pas les rôles par avance que il nous a semblé important que chacun puisse découvrir tous les aspects du projet. Nous avons privilégié l’apprentissage par rapport à l’efficacité.

2ème PARTIE:

Conception

# Le modèle

## Gestion des deplacements

Afin de gérer plus facilement les déplacements des lemmings, nous avons choisi de créer une classe Direction qui contient, la position du lemming et son sens de déplacement qui est une énumération de type SensDeplacement : VERS\_GAUCHE, VERS\_DROITE, VERS\_HAUT, VERS\_BAS.

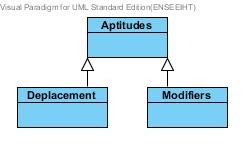
Elle pourrait s’avérer particulièrement utile si nous décidons un jour de de changer les axes (X,Y) de la carte.

## Architecture et affectation des aptitudes

### Architecture des aptitudes

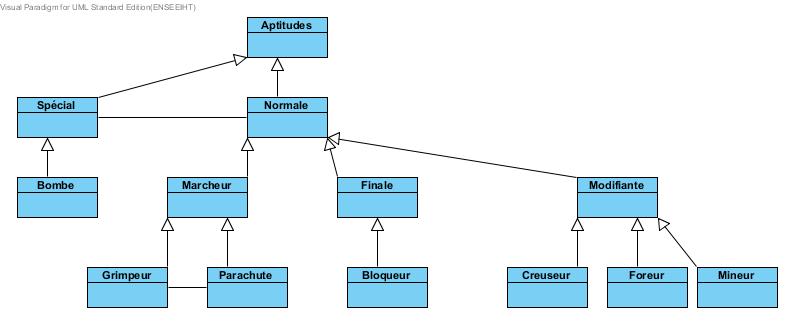
Le fil rouge du modèle a été la gestion des aptitudes. En effet, elles ont des spécifications bien différentes. Les aptitudes peuvent être permanentes, semi-permanentes, mettre les lemmings dans un état final, et certaines aptitudes peuvent se cumuler.

À première vue, on décèle les aptitudes permettant de déplacer les lemmings, de les modifier, et même de modifier la carte. Puisqu’un lemming peut changer d’aptitude en temps réel, nous avons tout de suite pensé au patron Stratégie ou *Aptitude* serait une interface avec une méthode *executer*().

Cependant, un simple patron ne permet pas de gérer la compatibilité entre les aptitudes.  
Nous nous sommes donc tournés vers une architecture séparant les aptitudes de modification et les aptitudes de déplacement.

Le lemming aurait dans ce cas une aptitude de déplacement et une aptitude de modification. Cependant un lemming peut être grimpeur et parachutiste en même temps, toutes deux des aptitudes de déplacement.  
La même analogie peut être faite pour les aptitudes de modification, le lemming pouvant être « creuseur » et avoir « exploseur ».

Une telle architecture est peu évolutive, et engendre une gestion des aptitudes assez lourde au niveau du lemming. C’est pourquoi nous avons exploré le patron décorateur, celui-ci semblait assez adaptable au niveau combinaison de différentes aptitudes. L'inconvénient de cette méthode serait la lourdeur du code. Chaque aptitude devrait connaître celle qu’elle englobe et effectuer un traitement différent selon celle-ci. De plus, la disparition des aptitudes temporaires aurait été très compliquée à gérer.

Ne trouvant pas de solution, nous nous sommes tournés vers un système d’aptitudes hiérarchique (Illustré par l’image ci-dessous). On remarque que l’aptitude spéciale est associée à une aptitude normale.

Ici Grimpeur et Parachutiste sont des « évolutions » de l’aptitude « Marcheur », en effet, par exemple un Grimpeur marche normalement jusqu’à ce qu’il rencontre un mur.

Il restait toujours le problème de l’Athlète, nous avons donc choisis, dans le cas de ce projet et par manque de temps d’explorer d’autres solutions, de lier Grimpeur et Parachutiste.

Ainsi, un Grimpeur peut disposer d’un parachute, et un Parachutiste peut avoir un grappin.

Les aptitudes de modification seront considérées comme temporaires par le lemming. Lorsque l’on attribue l’aptitude Creuseur au lemming, celui-ci creuse s’il le peut. Quand il a fini de creuser, il perd cette faculté.

Les classes Aptitudes, Spécial, Normale et Modifiante sont des classes abstraites. Nous n’avons pas choisis d’utiliser des interface car nous avions besoin de regrouper des traitements communs, par exemple la méthode *setSpécial().*

### Affectation des aptitudes

Quand le niveau décide d’affecter une aptitude à un lemming, celui-ci contrôle s’il peut la recevoir. S’il ne peut pas, il déclenche une exception du type *AptitudeInvalideException*.

Pour gérer les différentes aptitudes, nous avons choisis d’utiliser une Pile. Car le lemming exécute la dernière aptitude qu’il a. Cela est particulièrement intéressent pour les aptitudes semi-permanentes.  
Par exemple un Creuseur, lorsqu’il a fini de creuser, indique au lemming qu’il faut supprimer cette faculté.

Quand un lemming Marcheur deviens Grimpeur, on dépile la faculté Marcheur et on la empile celle de Grimpeur (pour rappel un grimpeur sait marcher et grimper). Il en est de même avec le Parachutiste.

Le lemming peut ajouter à son aptitude courante une Bombe.   
Cependant, la bombe étant une aptitude Spéciale, elle n’est pas empilée mais affectée dans l’aptitude actuelle, par exemple Marcheur.

Quant au Bloqueur, étant donné que c’est une aptitude Finale, la Pile est vidée avant que cette aptitude soit empilée. Le lemming gère le fait qu’aucune aptitude ne puisse être ajoutée après un Bloqueur, exception faite du Bombeur. Cette aptitude est assez particulière car elle introduit un type de Case particulier, une case non grimpable.

## Gestion du niveau

Le niveau est la classe qui reçoit le fichier et a pour responsabilités de construire la carte et les lemmings.

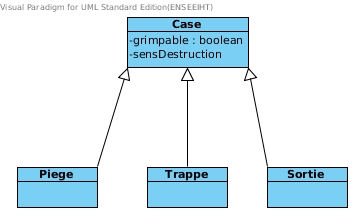
En ce qui concerne la construction de la carte, cette tâche est déléguée à la classe Carte elle-même. Le niveau s’occupe simplement de parser les informations telles la vitesse du jeu, le nombre de lemmings, et le nombre d’aptitudes par rapport à chacune d’entre elles.

Le niveau est le cœur du modèle. Il s’occupe de notifier tous les changements d’état de la carte, des lemmings ou de ses informations. Il a également le contrôle du temps, et effectue les mises à jour au bout d’un certain nombre de mise à jour en fonction de la vitesse de jeu spécifiée. Il crée également les lemmings au fur et à mesure, en fonction de la vitesse d’apparition spécifiée et les met ensuite à jour. C’est également par lui que toutes les requêtes de changement d’aptitude doivent transiter.

Dans un souci de clarté, il a été décidé d’extraire toutes les informations fonctionnelles du niveau (vitesse, nombre d’aptitudes, …) dans une classe spécifique. Cette classe devient donc celle qui notifie les changements de ces informations.

## Architecture de la carte et des cases

Afin de décrire la Carte, nous avons choisi d’utiliser une HashMap dont la clé est une position (définie par ses coordonnées, x et y), et dont la valeur associée est une Case. De ce fait pour une position donnée de la Carte, nous avons une unique Case. De plus grâce à cette méthode, nous associons dans la Carte la position et la Case, plus particulièrement le type de la case (destructible, trappe, sortie, piège, grimpable). De ce fait une case ne comporte pas de position ce qui nous évite de gérer ce problème au niveau de la case.

Pour la gestion des différentes cases, nous pensions au départ à faire un classe Case qui représenterait les cases indestructibles avec seulement un attribut grimpable pour gérer le cas particuliers du Bloqueur. Les autres classes en auraient hérité (destructible, piège, trappe, sortie). Finalement nous avons pensé à simplifier les classes représentant les cases, en créant une énumération SensDeDestruction (AUCUN, TOUS, VERS\_DROITE, VERS\_GAUCHE, VERS\_BAS). Grâce au sens de destruction AUCUN nous pouvions alors ajouter l’attribut sens de destruction à la classe Case, et de ce fait, simplifier la gestion de la destruction d’une Case (on évite donc de tester le type d’une case par un instanceof). De plus avec cet ajout, nous pouvons supprimer la classe destructible.

## Encapsulation par niveau, et par partie

Nous avons choisi de diviser le jeu avec plusieurs classes, en l'occurrence les classes Partie, et Niveau afin de pouvoir faire évoluer le jeu. Si nous avions eu le temps nous aurions pu créer différentes parties avec plusieurs niveaux.

# Les vues

Nous avons décidé de créer deux vues distinctes pour séparer la vue du modèle et la vue du contrôleur (bien que celle-ci comporte également des éléments du modèle). De plus il nous a semblé que la carte et les lemmings sont assez dissociés des informations du niveau, obtenues à partir du fichier.

### Vue du modèle (carte et lemmings)

Cette vue s’occupe de l’affichage de la carte et des lemmings. Elle reçoit les évènements du niveau spécifiant qu’une mise à jour a eu lieu et met à jour la position des cases et des lemmings.

Pour éviter tout problème de gestion de la coloration des cases, la carte est d’abord mise à jour. Puis les lemmings sont rafraîchis, et remplacent les cases déjà présentes. De plus, dans un souci de simplicité, nous avons décidé de représenter les cases de la même taille que les lemmings.

### Vue du contrôleur (aptitudes, contrôles du temps).

Cette vue s’occupe des actions utilisateurs et les transmet au contrôleur. Elle comporte donc les boutons de sélection des aptitudes, les réglages de  vitesse du jeu et d’apparition des lemmings, et enfin l’affichage des informations du niveau (nombres de lemmings morts, temps restant.

# Les contrôleurs

Chaque contrôleur dispose de ses propres vues et agit différemment sur le modèle.

### Le contrôleur de jeu

Le contrôleur de jeu est le contrôleur principal. C’est celui qui démarre l’application et initialise les vues. Il permet, en théorie, la sélection des parties.

Malheureusement, le temps nous manquant, il n’a actuellement pas de vue et n’offre donc pas toutes les fonctionnalités voulues. Il ne permet donc pas de choix entre les parties et se contente de transmettre le fichier du niveau.

Il crée la partie et délègue ensuite le travail au contrôleur de partie en lui transmettant la fenêtre principale et la partie crée.

### Le contrôleur de partie

Le contrôleur de partie est le contrôleur secondaire. Il permet de visualiser les différents niveaux d’une partie.

Le temps nous manquant encore une fois, nous n’avons pu lui définir de vue. Il ne permet donc pas de choisir entre les niveaux. Il permet juste de transmettre le fichier du niveau au contrôleur du niveau.

Il crée le niveau et le transmet au contrôleur de niveau. Il délègue la gestion de la fenêtre à ce dernier.

### Le contrôleur de niveau

Le contrôleur de partie est le contrôleur le plus spécifique. Il permet de visualiser le niveau, d’agir sur les lemmings et sur leurs aptitudes, ainsi que de modifier la vitesse de jeu et d’apparition des lemmings.

Nous nous sommes concentrés sur son développement, puisque c’est lui qui permet la fonctionnalité principale, à savoir “jouer”.

Il récupère les actions utilisateur reçues par la vue et modifie le modèle en conséquence.

# https://lh6.googleusercontent.com/ZLky6GX7bxRGaqrfUf6aF2sIPnPx6QPj-KSH-JNGo5sq0DH6xYUfgwBHFFCmsbOkXqM4jNVr4ubOQsTTOWbVwuKuh7i2r4SiL0xECdzNlYXcE4g1OQ4Interactions entre les composants

Comme expliqué plus haut, le contrôleur traduit les actions utilisateur venant des vues, en opération sur le modèle. Le modèle alors modifié, il informe alors ses vues/listeners du changement opéré.

Ce diagramme illustre le choix de vouloir séparer la vue du contrôleur et celle de l’affichage du modèle, ainsi que la présence de deux Observable agissant chacun sur une vue différente.

# Planning et tâches effectifs

## Planning Effectif

### 1ere semaine: Analyse

Au début lorsque nous avons reçu le sujet, nous n’avons pas vu la complexité de la gestion des aptitudes. C’est au fur et à mesure de l’analyse ou nous nous sommes rendu compte de la difficulté du projet.

Nous avons donc choisis de partir tous sur les mêmes bases. L’analyse a donc été faite en groupe.  
Cela nous a également permis d’avoir différents points de vue sur un domaine précis (tel que les aptitudes).

### 2ème SEMAINE: Conception et developpement

La 2ème semaine nous a permis de concevoir et de commencer à développer le jeu des lemmings. Elle a été riche d’enseignement sur l’API Swing, et sur les différents Design Pattern.

De même que pour l’analyse, la conception a été faite en groupe. Bien qu’elle fut en grande partie lors du développement, mis à part pour la gestion des aptitudes.

### 3ème semaine: Développement et finalisation

Nous avons dédié la dernière semaine à l’implantation, avec parfois des retours sur la conception. Notamment au niveau du modèle MVC, nous avons approfondi la connaissance que nous avions acquise en cours.

Par manque de temps nous avons effectué en parallèle de l’implantation des tests d’intégrations.

## Tâches

La répartition des tâches s’est faite lors de l’implantation. Au départ nous avons séparé le modèle en trois grosses parties, le niveau, la carte avec les cases et le lemming avec marcheur.

Une fois fini, deux personnes du groupe ont exploré les possibilités de la vue et du contrôleur.  
Pour en arriver à une personne par élément du modèle MVC : un pour le modèle, un pour le contrôleur et le dernier pour la vue.

Nous avons à tour de rôle changé, celui qui été sur la vue, et passé sur le modèle de même pour le contrôleur.

# Difficultés rencontrées

Une des difficultés que nous avons rencontrées a été l’utilisation de SWING. D’une part car nous avions commencé à faire l’interface graphique avec NETBEANS qui nous a générer du code peu compréhensible. D’autres parts, il nous a fallu le modifier par la suite. Ensuite nous avons eu des problèmes pour régler la disposition des éléments de nos vus pour les même raison. Finalement nous aurions mieux fait soit de faire toute l’interface graphique sur NETBEANS, soit de la faire entièrement à la main pour maîtriser totalement l’affichage.

Le second problème que nous avons rencontré s’est porté sur les choix de conception du modèle, notamment pour la conception des différentes aptitudes. En effet, nous avons fait la conception ensemble, nous avions donc des opinions différentes qui nous ont obligé à prendre du temps pour prendre une décision.

Ensuite, d’après nous, le plus grand défaut que nous avons eu lors de la réalisation de ce projet a été le manque de communication qui nous a fait perdre un peu de temps sur certains points. Ce défaut peut s’expliquer en partie par le manque de temps et l’absence de marc en fin de semaine dernière pour des raisons médicales.

Le dernier problème qui nous a perturbés a été la recherche d’une Case dans la HashMap. Pour récupérer une case nous utilisions un objet de type Position qui était égal (logiquement) à une Position présente dans la HashMap. Nous pensions d’après la spécification de la HashMap qu’un equals() était réalisé sur les clés. Néanmoins cela ne fonctionnant pas, nous avons donc nous même implémenter une méthode qui réalise la recherche dans la HashMap afin de récupérer notre valeur.