Projet Long Java
Jeu des Lemmings :
Rapport final

HORNIK Jessica LELEUX Philippe

12 juin 2012

Résumé

Ce rapport a pour objectif d'expliquer les choix d'implémentation issus de l'analyse du problème dont nous avons déjà rendu le rapport. Y figureront différentes informations relatives par exemple aux problèmes rencontrés durant l'implémentation de notre solution au problème posé ou encore à l'aspect protégé de l'application.

Table des matières

1 Introduction **3**

2 Ajouts et suppressions de l'analyse première **3**
2.1 Un ordre des compétences 3
2.2 Concernant la vue 4
2.3 Concernant le séquencement des actions 4
2.4 Pour aller plus loin 4

3 Des choix d'implémentation **5**
3.1 La classe Walker : basique mais non triviale 5
3.2 Un condensé de requêtes : la classe Action 5
3.3 La classe Lemming : un effort conceptuel 6
3.4 Une couche protectrice : Level_proxy 6
3.5 Le maître du jeu 7

4 Modèle, Vue et Controleur **7**
4.1 Un but adaptatif 7
4.2 Une vue qui observe 8
4.3 Un modèle qui regroupe les éléments du jeu 8
4.4 Un controleur qui les lie 8
4.5 Ce concept appliqué aux lemmings 9

5 Répartition des tâches **9**

6 Pour aller plus loin **10**

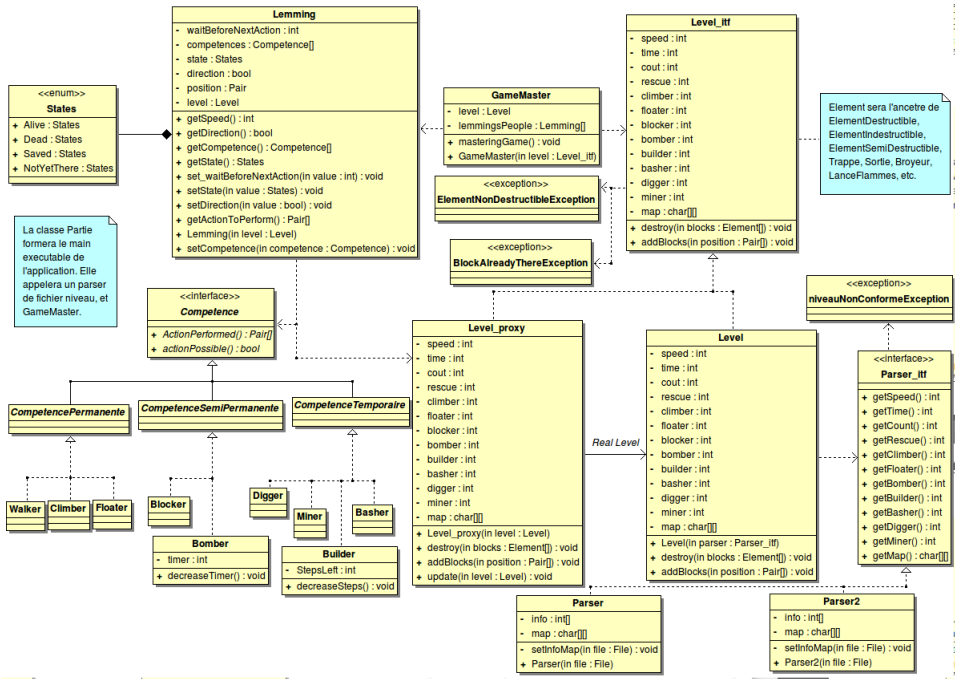

FIGURE 1 – Diagramme des classes préliminaire

1 Introduction

En guise d'introduction, rappelons le diagramme des classes qui résume l'analyse préliminaire que nous avions faite du sujet :

Nous avons, dans l'implémentation de cette analyse, tenté de respecter le plus scrupuleusement ce choix de conception afin de faciliter, par exemple, la répartition des tâches, et d'avoir un débuggage plus efficace. Néanmoins, comme la conception n'est qu'un modèle qui n'a pas encore été mis à l'épreuve, notre implémentation nous a forcé à ajouter ou, au contraire, enlever, certaines des composantes de notre analyse première, ce qui nous conduit en section 2.

2 Ajouts et suppressions de l'analyse première

2.1 Un ordre des compétences

Le tableau que nous avions mis au point lors du rapport d'analyse précisant les différentes interactions entre les compétences a donné naissance à la nécessité d'ordonner les compétences en fonction de leur ordre de priorité, ce

dont nous n'avions pas réellement pensé lors de ladite analyse. Pour cela, nous utilisons la hiérarchie propre aux classes issues de la super-classe compétence. A l'aide de l'opérateur `instanceof` et de méthodes parfois surchargées, parfois non en fonction des cas, nous avons réussi à définir un ordre total sur l'ensemble des compétences, permettant ainsi un comportement vis-à-vis de celles-ci conforme à ce que nous avions prévu, mais de manière détournée cependant.

2.2 Concernant la vue ...

Certaines modifications concernant la vue ont été apportées par rapport au patron que nous avions présenté lors du premier rapport. Ces changements sont les suivants :

- L'utilisateur a à sa disposition 3 niveau de vitesse, dont il a un affichage dans la zone d'état à la droite de l'écran.
- L'utilisateur dispose de boutons "+" et "-" qui permettent de gérer le débit des lemmings à travers la trappe d'entrée.
- Nous avons décidé de ne pas mettre de bouton settings qui, après mûre réflexion, n'avait pas l'utilité que nous avions espéré.

2.3 Concernant le séquencement des actions

Le rapport d'analyse ne mentionnait pas de séquencement particulier des différentes actions. Nous avons néammoins eu l'obligation d'imposer une horloge sur laquelle notre modèle se synchronise. Ceci a été effectué grâce à un Timer, comme conseillé, de la librairie swing. Grâce à une gestion d'événement, nous parvenons à mettre à jour la vue de manière régulière afin de donner l'illusion du mouvements à nos lemmings. Ceci ayant été dit, nous n'avons cependant pas pu rendre invisible à l'oeil humain les discontinuités dans ce mouvement. Le problème n'était pas tant la fréquence de rafraîchissement que la trop grosse pixellisation de l'image (un JLabel par bloc). Les lemmings, se déplaçant de manière saccadée blocs après blocs, ne peuvent pas avoir un mouvement aussi fluide que désiré ; cela ne nuit cependant pas au plaisir du joueur, qui recherche davantage l'aspect stratégique que la douceur des mouvements.

2.4 Pour aller plus loin

Comme mentionné plus haut, il est très difficile de respecter scrupuleusement le modèle que l'on s'est imposé au début de l'analyse, mais il est aussi très prolifique de l'exploiter au maximum ... Dans la section suivante, nous

allons détailler quelques choix d'implémentation inspirés fortement de notre analyse sous-jacente ou simplement très intéressant pour évaluer la qualité du travail fourni.

3 Des choix d'implémentation

3.1 La classe Walker : basique mais non triviale

Puisque les choix d'implémentation dépendent, en amont, de ceux fait lors de la conception, nous allons présenter dans un premier temps ici la classe Walker, qui est un parfait exemple de cette synergie.
La classe Walker représente l'aptitude qu'a un Lemming à se déplacer de manière intuitive dans son environnement. Ce déplacement s'effectue némmoins selon certaines règles que voici :

- Un walker se déplace par défaut de la gauche vers la droite.
- Un walker qui heurte un bloc de taille 1 l'escalade.
- Un walker qui heurte un bloc de taille supérieure à 1 ou qui se situe en haut de la carte change de sens de marche.
- Un walker qui n'a pas de sol sous lui tombe.
- Un walker qui tombe depuis plus de 5 bloc s'écrasera quand il atteindra le sol.
- Un walker qui atteind un bord de la carte meurt.

Grâce à ces préceptes de bases, il nous est possible, à un instant t donné, de calculer la position (et / ou l'état) d'un walker à l'instant $t + 1$ grâce au fait qu'un Lemming (et donc un walker) possède une copie de la carte (ce sur quoi nous reviendrons plus tard). En outre, tout Lemming possède par défaut la compétence walker, celle-ci permet donc, simplement, de mener à bien tous les niveaux qui ne demandent que de se déplacer. Cela a constitué la base de nos tests pour notre vue, et donc un état de fait très intéressant pour le développement de l'application.

3.2 Un condensé de requêtes : la classe Action

N'étant composée que d'un constructeur et de *getter* et *setter*, la classe Action n'en demeure pas moins indispensable à notre conception du jeu. Cet objet comporte, entre autres, la position où le lemming requiert d'aller, celle où il demande à construire, celles qu'il veut détruire, etc. De plus, elle renferme des informations sur le lemming en question, comme le fait d'être en train de tomber, d'escalader, etc.

Nous l'avons utilisée dans toutes les classes de compétence, en tant que requête d'action à effectuer, par le lemming, et fournie au GameMaster, dont le rôle sera approfondi un peu plus tard.

3.3 La classe Lemming : un effort conceptuel

Nous avons jusqu'ici traité des compétences, et de comment les lemmings avec une certaine compétence renvoyaient une requête d'action au maître du jeu. A ce moment-là, alors que nous savions comment faire effectuer une action à un lemming avec une compétence, le problème a été de savoir quelle action attribuer à un lemming avec plusieurs compétences. En effet, certaines des compétences étant cumulables, il fallait savoir comment gérer celles-ci, quelle(s) compétence(s) prendre en compte pour l'étape en question.
Comme évoqué précédemment, les compétences se sont vues attribuer un ordre de priorité - total -, qui nous a permis de les ranger dans un tableau dans ce même ordre. Par exemple, le Bomber devant effectuer son action dans tous les cas, il se place prioritairement à toutes les autres compétences. En outre, il fallait tenir compte du fait que les compétences temporaires n'étaient effectuables qu'à l'instant où l'on affectait la compétence au lemming. Toutes ces contraintes nous ont conduit à la méthode `setCompetence`, qui range la compétence à attribuer dans la liste de compétences du lemming, et dans un second temps, à l'action du maître du jeu.

3.4 Une couche protectrice : Level_proxy

Comme nous l'avions expliqué dans le rapport d'analyse, nous avons choisi de représenter le niveau non pas sous la forme d'une classe simple mais sous celle d'une interface dans le but de pouvoir créer un niveau virtuel qui serait en tout point une copie du niveau réel à ceci-près qu'on n'y autoriserait aucune modification.
Nous avons donc notre interface Level_itf qui est réalisée par le Level ET le Level_proxy. Le level est le seul à pouvoir être modifié et il ne l'est que par l'intermédiaire du maître du jeu qui vérifie que les actions demandées ne sont pas frauduleuses (destruction d'un bloc indestructible par exemple). Le level_proxy, quant à lui, est une copie en lecture du level à travers lequel les lemmings vont pouvoir récupérer des informations afin, par exemple, de calculer leur position future ou les éléments avec lesquels ils vont pouvoir interagir. En aucun cas, avec notre choix d'implantation, les lemmings ne peuvent modifier directement le niveau, afin d'éviter toute triche. Il ne font

que soumettre des intentions d'action au gamemaster qui, si elle sont valides, les réalise.
La combinaison du GameMaster et du Level_proxy octroie donc une certaine sécurité à notre niveau dans le sens où il ne sera modifié que lorsque tout indique que l'action demandée est possible. En pratique, les méthodes d'écriture dans Level_proxy renvoient directement une exception, du fait que les lemmings n'ont pas le droit de les appeler, même si elles existent. Tout doit passer par le gameMaster, dont nous parlons ci-après.

3.5 Le maître du jeu

Nous avons déjà expliqué plus haut le rôle de maître du jeu, mais il est néammoins toujours intéressant de s'y attarder, afin de détailler quelque peu son rôle premier. Il est celui qui centralise toutes les classes métier en une seule et unique méthode : `public void nextStep();` Cette procédure, comme son nom l'indique, calcule, à partir des données actuelles, l'état du système à l'instant suivant (au prochain événement envoyé par le Timer).
Le maître du jeu, au sein de cette méthode, va récupérer toutes les actions à réaliser pour chaque lemming, et les mettre en oeuvre si possible, afin d'obtenir en définitive la position (si elle existe *i.e.* si il n'est pas mort, sauvé, ou non encore apparu) de chacun des lemmings ainsi que les potentielles modifications qu'ils auraient ajouté à la carte (création d'un bloc pour un builder, destruction de plusieurs blocs pour un bomber ...).
Il s'agit de la méthode phare du projet, qui regroupe à elle seule toutes les données manipulées par les différents acteurs du métier, afin de les corréler pour servir à la génération de l'affichage effectif (qui sera physiquement géré par le controleur).
Puisque nous en sommes venus à parler du controleur, nous allons à présent traiter du patron MVC que nous avons voulu utiliser pour l'aspect graphique de notre application.

4 Modèle, Vue et Controleur

4.1 Un but adaptatif

Nous avons choisi, pour notre vue et sa relation avec les classes métier d'utiliser le patron Modèle, vue et controleur, dans le but de séparer de manière tranchée la vue, qui n'est que le reflet du modèle, de celui-ci. Le but est ici d'autoriser d'autres développeurs à créer d'autres vues avec des API différentes de Swing tout en rendant possible sa connection avec notre

métier. Dans la suite de cette section, nous allons détailler la vision que nous avons eu de ce patron, sans prétendre à son exactitude la plus complète.

4.2 Une vue qui observe

La vue est une classe contenant comme composant principal une JFrame, à l'intérieur de laquelle s'afficheront les différents éléments du jeu ainsi que les boutons permettant à l'utilisateur d'interagir avec lui. Cette vue est déclarée comme réalisant l'interface `Observer`. Cette interface ne contient qu'une unique méthode `public void update(Observable o, Object arg);` dont l'implémentation est cependant primordiale, puisque c'est par son intermédiaire que notre vue se mettra à jour sur le modèle.

4.3 Un modèle qui regroupe les éléments du jeu

Le modèle, quant à lui, est la réunion des éléments qui composent le jeu. Il sert aussi de regroupement des informations qui le concernent et de passerelle entre les classes métier et le reste du patron. On lui adresse les requêtes concernant le métier, et il s'efforce d'y répondre. Son seul lien direct avec la vue est le suivant : il étend une classe `Observable` dans laquelle figure notamment les méthodes suivantes :

- `public void addObserver(Observer o);` Cette méthode permet de mettre à jour la liste des observateurs de ce modèle.
- `public void setChanged();` Ici, on déclare qu'il y a eu des changements dans le modèle.
- `public void notifyObservers();` Là, on informe les différents observateurs de ces changements.

Le lien entre le modèle et la vue est donc très étroit mais nous ne le manipulons jamais en dehors de ces trois méthodes couplées avec la méthode update de la vue. Le reste de ce lien se fait par l'intermédiaire du controleur.

4.4 Un controleur qui les lie

Le controleur sert d'intermédiaire de rafraichissement entre le modèle et la vue. En pratique, c'est à lui que la vue demande de se mettre à jour. Il reçoit la requete, demande au modèle les informations dont il a besoin pour les traiter et rafraîchit ainsi la vue. Il est donc appelé dans la méthode update de la vue mais aussi dans les différentes actions conséquences aux événements de l'utilisateur (on clique sur un bouton, on sélectionne un menu ...).

4.5 Ce concept appliqué aux lemmings

Dans le cadre de notre jeu de lemming, c'est le modèle qui dispose d'un `Timer`. A chaque événement de ce timer, il demande au maître du jeu de mettre à jour le métier (de recalculer la position des lemmings et l'état du niveau). Une fois cette mise à jour effectuée, le modèle, par l'intermédiaire des deux méthodes citées ci-dessus, informe la vue qu'il y a eu des modifications. En réponse, celle ci exécute sa méthode update, c'est à dire demande au controleur de la mettre à jour. Ce dernier va demander au modele le niveau mis à jour ainsi que la liste des lemmings mise à jour et va, grâce à ces informations, rafraîchir la vue afin qu'elle reflète correctement l'état courant du modèle.
Ceci se passe de la même manière pour les différentes actions liées au événements qui passent toutes par l'intermédiaire du controleur pour s'effectuer.
Tous ces choix de conception et ces difficultés pratiques ont nécessité une répartition des tâches efficace, ce dont traite la section suivante.

5 Répartition des tâches

Le rapport d'analyse expliquait comment, à deux, nous allions parvenir à nous répartir les tâches relatives à toute la partie métier de l'application. Nous avons tenu à respecter cette répartition du travail, mais il est rapidement apparu que les classes traitant des compétences seraient la partie la plus conséquente à implémenter, à cause de la grande diversité des cas particuliers. Chacune des classes de compétence devait évaluer, en fonction de la position du lemming et de son environnement, quelle action celui-ci devait faire. De plus, la classe Lemming se devait de classer les compétences par ordre de priorité. C'est cet état de faits qui nous a poussé tous deux à centrer nos efforts sur la gestion des compétences.

Pour centraliser nos travaux, nous avons utilisé un dépot Mercurial hébergé sur `bitbucket.org`. Ceci nous a permi de travailler très efficacement en parallèle tout en étant constamment au courant des modifications apportées au projet par notre binôme.
Une fois les classes métier terminées, nous avons décidé de la répartition suivante pour le reste de l'application :

- Jessica ferait le patron MVC
- Philippe s'occuperait de toutes les classes de tests unitaires

Nous avons cependant regretté de n'être que deux à travailler sur ce projet. Cela nous a coûté beacoup de temps et d'énergie à faire en si peu de temps à

deux un projet que nous aurions dû faire à trois. Néammoins, nous sommes parvenus à une répartition des tâches équitable et prolifique.

6 Pour aller plus loin

Malgré tous nos efforts, force est de constater que notre application n'est pas à la hauteur de nos espérances décrites dans notre rapport d'analyse. Le fait que nous n'ayons été que 2 nous a fortement pénalisés et nous a empêcher d'aller aussi loin que nous l'aurions espéré (génération de niveaux, de compétences ...). Ceci ayant été dit, nous avons appris beaucoup grâce à ce projet (notamment au niveau du partage des tâches, de l'API Swing ou encore des tests unitaires) et nous aurions bon espoir, avec un peu plus de temps, d'exploiter toutes nos idées au maximum afin de rendre notre jeu des lemmings aussi attrayant que nous l'avions à l'origine souhaité. Nous pourrions par exemple représenter chaque bloc par plusieurs JLabels afin de fluidifier le mouvement des Lemmings. Nous pourrions aussi prendre plus en compte la vitesse des lemmings (qui diminuerait sur sol collant, par exemple) ou encore imaginer une succession dynamique de niveau. Toutes ces idées restent en suspens jusqu'à, sans doute, un nouveau projet !

Annexe : Diagramme des classes définitif

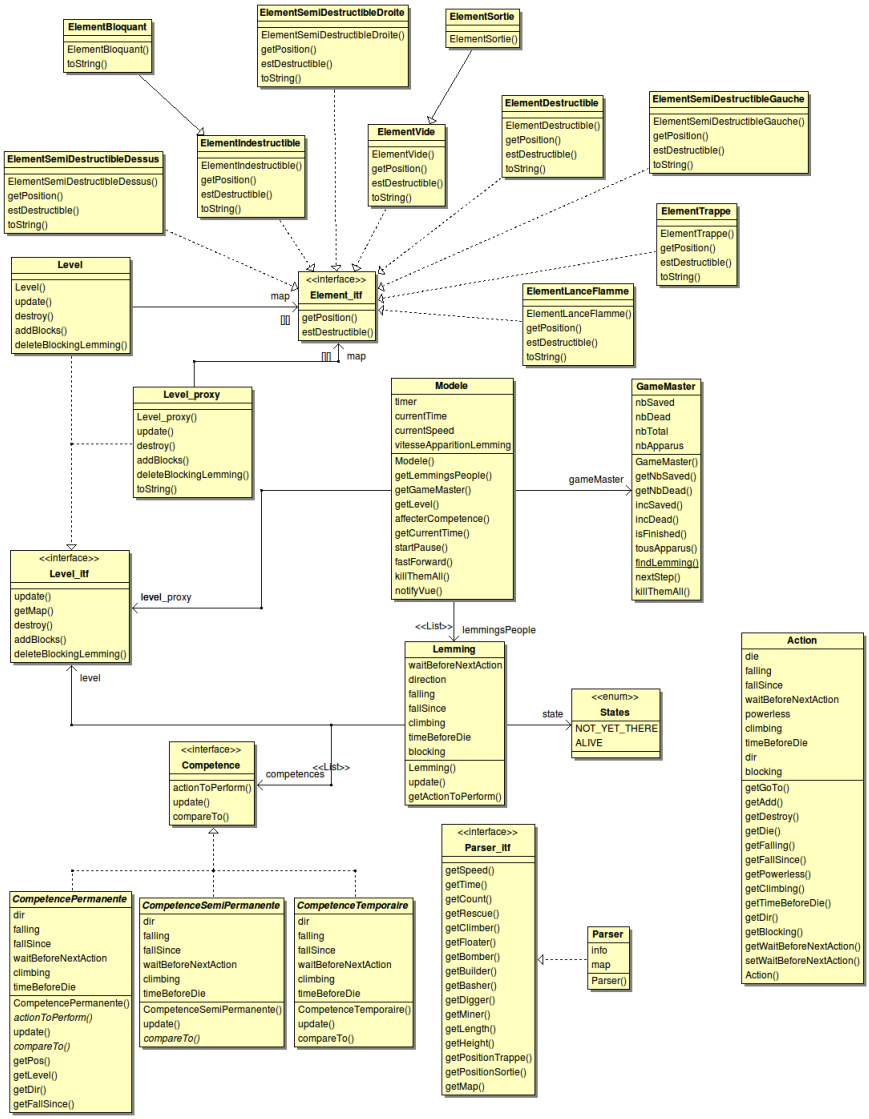

FIGURE 2 – Diagramme des classes métiers définitif