Ecole Ingénieurs 2000 - Université de Marne-la-Vallée
Filière Informatique et Réseaux
2ème année

Documentation développeur

******

Manuel du jeu HMM 2000

******

Ce manuel explique en détail l'architecture du jeu HMM2000. Ce dernier est un clone simplifié du jeu Heroes of Might & Magic. Ce jeu a été développé en Java dans le cadre de notre cursus scolaire au sein de l'école Ingénieurs2000.

Compte rendu de : **Tom MIETTE, Sébastien MOURET**

Fait le : **8 janvier 2008**

Sommaire

SOMMAIRE **2**

INTRODUCTION **2**

I. ARCHITECTURE 3
1. Gestion des événements *3*
2. Les cartes (« maps ») et états *4*
3. Les managers *5*
a) Le manager de sélection 5
b) Le manager de mouvements 5
c) Le manager de jours 5
d) Le manager de position de combat 5
e) Le manager d'échange 6
f) Le manager de combat 6
4. L'interface graphique *6*
II. L'AJOUT D'UNITES ET AUTRES 6
1. Les ressources *7*
2. Les entités vendeuses *7*
3. Les héros et monstres *7*
4. Les unités de combat *7*
5. Les autres ensembles modifiables *8*
6. Le système de sauvegarde *8*
III. LIMITES ET BUG CONNUS 8
1. Premier rafraichissement du JPanel des caractéristiques *8*
2. Exception dans l' « EventQueue » *9*
3. Jar exécutable et chemins relatifs *9*
4. Ralentissement de la Thread principale pour le mouvement *9*
5. « lawrence » et les différentes cartes *10*

Introduction

Le but de ce manuel est de présenter l'architecture mise en place pour la réalisation de cette application. Nous expliquerons ici comment fonctionnent les classes et comment les paquetages interagissent entre eux. Puis nous décrirons le mode opératoire à suivre pour facilement ajouter de nouvelles fonctionnalités au programme telles que la création de nouveaux types de créatures, de sorts ou encore la modification des règles de combat.

Pour une meilleure compréhension de ce manuel, il est préférable de lire auparavant le manuel de l'utilisateur afin de s'imprégner des règles du jeu et de comprendre en détails le fonctionnement du jeu.

I. Architecture

HMM2000 est une application basée sur le modèle MVC sans pour autant respecter scrupuleusement son comportement. Toutes les actions de jeu sont provoquées par l'utilisateur via l'interface graphique soit par un clic de souris, soit par une frappe au clavier Tout le principe de l'application réside donc dans la gestion de ses actions, aussi appelées événements utilisateur.

1. Gestion des événements

L'architecture que nous avons mise en place pour répondre à ce problème peut se résumer d'après le schéma suivant :

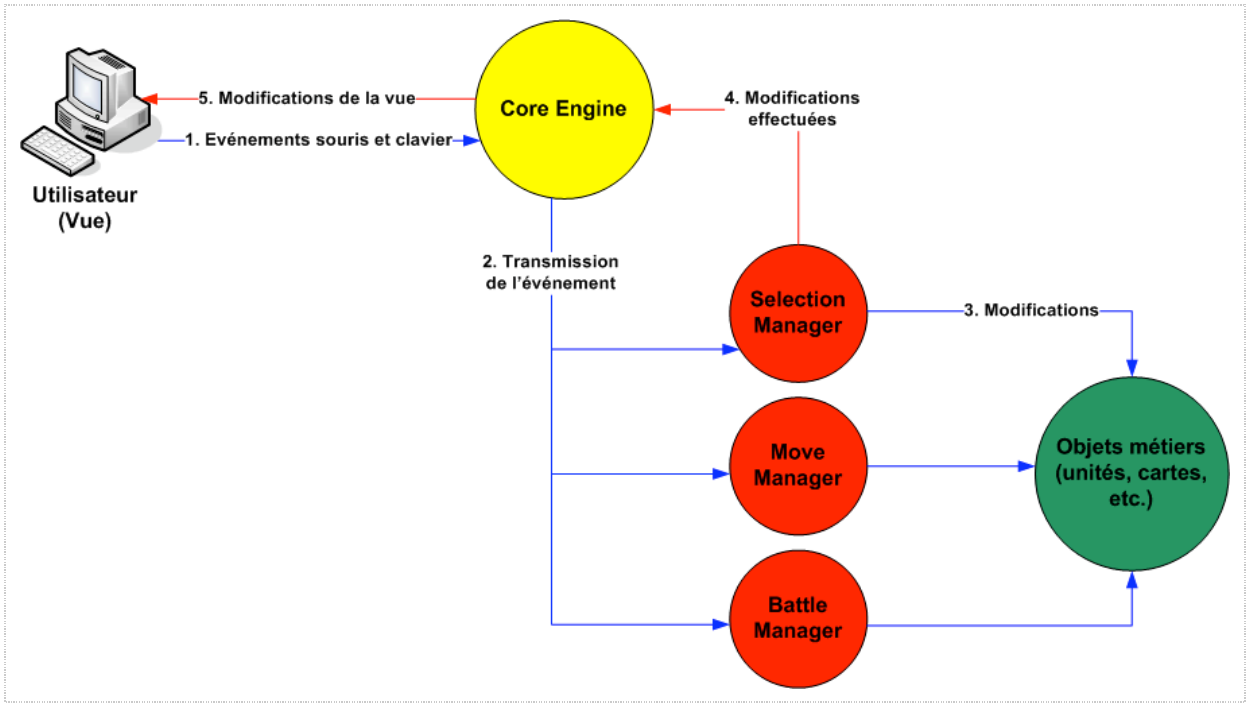

Schéma 1 : Gestion des événements utilisateur dans HMM2000.

La gestion de chaque événement (clic ou frappe) qui modifie l'état du jeu est géré de la même façon suivant cinq grandes étapes :

1. L'utilisateur effectue un clic de souris ou une frappe au clavier pour demander une action, « déplacer une unité » par exemple. Cet événement est envoyer au moteur de gestion de l'application « **CoreEngine** ». Ce moteur a la charge d'un groupe de managers qui ont chacun un rôle bien particulier.

2. En fonction de la configuration actuelle du jeu (combat, déplacement, sélection, etc.), le moteur décide quel est le manager qui doit interpréter l'événement envoyé par

l'utilisateur. Dans notre cas de déplacement d'une unité, c'est le manager de mouvements « **MoveManager** » qui sera sollicité.

3. Ensuite, le manager qui reçoit l'événement le traite selon ses propres paramètres. Toujours dans le cas du déplacement, le manager de mouvements affichera le chemin à emprunter, déplacera l'unité et gérera les éventuelles rencontres avec les autres unités. Pour ce faire, chaque manager manipule et modifie les objets métiers (les objets qui constituent réellement le jeu).

Dans notre cas, la modification qui aurait pu intervenir est « déplacement de l'unité X à la position Y ».

4. Une fois la modification correctement effectuée, le manager transfert les informations sur cette modification au moteur (via des méthodes « fire »).

5. Le moteur analyse la modification et informe en ensuite la vue (et donc l'utilisateur). Dans notre cas, la vue affichera l'unité X à la position Y (en pensant à effacer l'unité de sa précédente position ».

2. Les cartes (« maps ») et états

Dans notre programme, les cartes de jeu sont représentées sous la forme de deux couches. Une couche dite de « background » qui stocke tous les éléments d'arrière-plan qui constituent l'environnement dans lesquels les unités se déplacent. La seconde couche est la couche dite de « foreground » qui stocke les éléments d'avant-plan tels que les héros, les monstres, les châteaux et toutes les unités qui peuvent avoir une interaction quelconque avec l'utilisateur du jeu.

Ce mécanisme permet de bien distinguer les éléments utiles à la gestion des événements et de ne pas mélanger les deux notions d'avant- plan et d'arrière-plan. De plus, cette représentation facilite grandement le travail d'affichage de la vue qui construit en deux temps l'interface graphique. Les images qui devront être modifiés au cours de la partie ne sont donc que des images représentant des éléments d'avant-plan.

Dans notre application, il existe trois types de cartes qui sont affichées successivement par la vue en fonction de l'état actuel du jeu. Ces trois cartes sont :

1. La carte principale dans laquelle les unités peuvent se mouvoir, acheter des sorts ou encore gérer les caractéristiques du château.

2. La carte de position de combat qui est celle qui représente les 12 slots d'un héros pouvant contenir des unités de combat. Cette carte permet la modification de cette position de combat.

3. La carte de combat qui gérer à la fois les combats entre deux entités ennemies et les échanges d'unités entre les entités amies.

Ces cartes constituent les quatre différents (deux pour la carte de combat) états dans lesquels peut se trouver le moteur du jeu à un instant donné. C'est précisément cet état qui déterminera le manager sollicité lors d'un événement utilisateur.

3. Les managers

Les différents managers que possèdent HMM2000 ont chacun un rôle définit et ils agissent les uns avec les autres afin de manipuler les objets métiers. Ces managers sont les suivants :

a) Le manager de sélection

Ce manager gère l'unité ou l'élément qui est actuellement sélectionnée sur la carte du jeu. Lorsqu'un événement arrive, ceci permet de savoir comment interpréter l'interpréter en fonction de l'unité sélectionné. Ce manager est partagé par tous les autres afin d'accéder à cette fameuse unité depuis n'importe quel manager.

b) Le manager de mouvements

Ce manager a pour rôle la gestion des mouvements d'unités sur la carte. C'est lui qui calcule les chemins praticables sur la carte entre deux positions données grâce à l'algorithme A*. Il s'occupe également du déplacement de l'unité sur la carte et gère la rencontre avec d'autres unités.

C'est lui qui enclenchera un combat entre deux unités ennemies ou qui commencera un échange d'unité entre un château et son héros.

c) Le manager de jours

Ce manager gère le tour par tour en fonction du nombre de joueurs présents dans la partie. Une fois que tous les joueurs ont joué, le manager incrémente le nombre de jours et informa tous les éléments d'avant-plan du nouveau jour qui commence.

Il est a noté qu'à chaque début de combat, un sous manager de jours est crée avant de gérer le tour par tour au sein même du combat. En effet, la gestion des tours hors du combat et dans le combat sont indépendantes et donc gérées par deux managers distincts.

d) Le manager de position de combat

Ce manager gère la carte de position de combat. Il a un rôle très simple qui consiste à gérer les positions des unités de combats au sein même d'un héro, d'un monstre ou d'un château.

e) Le manager d'échange

Ce manager est très similaire au précédent. Il permet de mettre en place l'échange d'unités de combat entre deux entités amies. Il utilise la carte de combat pour remplir ce rôle.

f) Le manager de combat

Ce manager gère tout l'environnement de combat. Il a la charge d'éliminer les unités tuées ou encore de d'exécuter les capacités des héros. C'est lui qui détermine quand le combat est achevé et qui informe le moteur de l'unité perdante.

Par la suite, le moteur décide si oui ou non le joueur qui a perdu le combat perd également la partie en fonction.

4. L'interface graphique

L'interface graphique qui est utilisée dans HMM2000 est celle gérée par la libraire graphique « **lawrence** » à laquelle nous avons ajouté une fenêtre swing pour plus de convivialité.

Cependant, le moteur du jeu est capable de fonctionner avec toute autres interfaces graphiques répondant à certains critères. Ces critères sont en fait définit par une interface « **HMMUserInterface** » que toute interface graphique doit implémenter pour afficher le jeu.

Cette interface est minimaliste puisqu'elle demande simplement de savoir dessiner une carte (avec les deux couches d'avant et d'arrière-plan décrites ci-dessus), d'ajouter et de supprimer un sprite à une position donnée.

Chaque interface graphique peut également implémenter le visiteur d'affichage « **UIDisplayingVisitor** » afin d'afficher les caractéristiques des unités lorsqu'elles sont sélectionnées. C'est le rôle que nous avons donné à la fenêtre swing dans notre application. De plus, l'interface graphique doit également mettre en place un « **UIChoicesManager** » qui est sollicité au moment ou un choix est demandé à l'utilisateur (pour l'achat d'un objet, par exemple). Le gestionnaire de choix doit simplement être capable de renvoyer un choix sélectionné parmi une liste de choix.

Ces trois interfaces définissent à elles seules ce que doit être une interface utilisateur pour HMM2000. Il n'est pas obligatoire que cette interface soit graphique, puisque les affichages peuvent se faire en console par exemple.

II. L'ajout d'unités et autres

Dans notre architecture, nous avons porté une attention particulière à la gestion de la création des unités, des ressources, des marchands, etc. En effet, les diverses catégories

dont l'ensemble est modifiable sont toutes gérées sous la forme d'enum java. Il est ainsi très simple de supprimer ou d'ajouter un élément à ces ensembles.

Dans tous les cas, il est préférable ajouter un sprite particulier pour l'élément ajouter et de l'enregistrer dans l'enum « **Sprite** » qui recense les images déjà existantes. Pour les unités de combat, il faut également ajouter un icône type PNG afin d'afficher les caractéristiques des unités dans la fenêtre swing.

1. Les ressources

Il n'existe qu'un type de ressource par défaut dans le jeu, à savoir l'or. Pour supprimer ou ajouter une nouvelle ressource, il suffit d'ajouter un membre à l'enum « **Kind** » de la classe « **Resource** ». De cette façon, la ressource sera automatiquement prise en compte et pourra apparaître sur les cartes et dans les prix des objets.

2. Les entités vendeuses

Dans le jeu, ils existent deux entités capables de vendre des objets au joueur (les marchands et les arènes). Pour ajouter une nouvelle entité, il suffit de l'enregistrer dans l'enum « **SalesEntityEnum** » de la classe « **SalesEntity** ».

Lors de son instanciation, il faut alors lui ajouter les objets qu'elle est en mesure de vendre (objets « **Sellable** » qui possède un prix).

3. Les héros et monstres

Les héros ou les monstres sont gérés de la même façon. Ce sont des unités capables de contenir 12 unités de combat. Les types de héros ou de monstres sont définit sous la forme de profils.

En effet, l'instanciation d'un objet héro se fait via une multitude de paramètres (vie, vitesse, capacités, etc.) qui sont fastidieux à spécifier à chaque construction. Pour simplifier ceci, nous avons mis en place des profils qui définissent pour chaque type de héros ou de monstres les caractéristiques par défaut que possède ce dernier.

Ainsi, en définissant un nouveau profil ou en modifiant un profil existant, nous pouvons aisément définir de nouvelles créatures. Ces ajouts doivent ce faire dans les deux enum « **HeroProfile** » et « **MonsterProfile** ».

A titre d'exemple, nous pouvons ajouter une nouvelle capacité (« **Skill** ») au héro de type « **Archer** » en ajoutant cette capacité dans son profil.

4. Les unités de combat

De la même façon que les héros, les unités de combats (fantassins, unités volantes, unités magique, vampires, etc.) sont également représentées sous la forme de profils dans l'enum « **WarriorProfile** ».

Cependant, en plus de ce profil, l'instanciation d'une unité de combat est associée à un niveau (« **Level** »). Le niveau définit un ratio qui sera appliqué sur toutes les caractéristiques de cette unité de combat.

De cette manière, des unités de combat de même profil pourront êtres de capacités complètement différentes en fonction de leur niveau.

5. Les autres ensembles modifiables

Comme nous le disions en introduction de cette partie, la plupart des autres éléments du jeu sont également facilement modifiables via l'utilisation des enums. Cette architecture offre la possibilité à d'autres développeurs d'agrémenter le jeu avec leurs nouveaux éléments de façon légère et simple.

Ce mécanisme est en particulier utilisé pour les sorts achetés chez les marchands (« **Spell** »), pour les niveaux des unités de combat (« **Level** »), pour les capacités spéciales des héros (« **Skill** ») ou encore pour les comportements d'attaque des unités de combat (« **AttackBehaviour** »).

6. Le système de sauvegarde

Lorsqu'un élément est ajouté à ces ensembles, il faut définir sa façon de se sauvegarder dans le système de sauvegarde. Ce mécanisme n'est pas très optimisé mais permet tout de même d'arrêter une partie et de la reprendre plus tard.

L'enregistrement du nouvel élément se fait à la fois dans les classes « **CharacterTranslator** » et « **MapBuilder** » afin de définir la façon de sauvegarder l'élément (sous forme de chaine de caractères).

Il faut également définir la méthode de chargement de cet élément en implémentant l'interface « **MapForegroundElementInitializer** ».

Ce mécanisme de sauvegarde n'est pas très robuste mais ne faisait pas partie de nos objectifs de développement prioritaires. Nous pourrions améliorer notre format de sauvegarde afin que toutes les unités ajoutés n'aient pas à définir leur comportement de sauvegarde.

III. Limites et bug connus

Certaines contraintes d'implémentation entraînent certaines limites à notre programme auxquelles nous n'avons pas encore pu apporter de solution stable. Voici une brève liste des principaux problèmes qui subsistent dans notre application.

1. Premier rafraichissement du JPanel des caractéristiques

Le JPanel swing qui constitue le panel d'affichage des caractéristiques dans la fenêtre swing qui accompagne « **lawrence** » ne s'affiche pas correctement lors de sont premier chargement.

En effet, lorsque nous cliquons sur une unité, la composition du panel est modifiée ce même panel est rafraîchit. Cependant, lors de la première sélection d'un type d'unité (associée à un type de panel), le panel possède une taille de 0x0 et n'apparaît pas dans la fenêtre swing. Ce comportement restreint un peu l'interactivité avec l'utilisateur qui est forcé de sélectionner une autre unité pour voir son panel se rafraichir.

2. Exception dans l' « EventQueue »

A un instant donné dans la partie, il nous est arrivé de voir apparaître une exception de l' « EventQueue » de swing. Nous n'avons pas pu déterminer à quel moment précis cette exception est levée, mais elle n'interrompt en aucun cas le bon déroulement du jeu.

Cette exception est levée car un rafraichissement graphique n'a pu s'effectuer correctement, mais ce dernier sera accomplit plus tard. L'application n'est donc pas affectée.

La cause supposée de cette exception est la cohabitation entre la libraire graphique « **lawrence** » et la fenêtre swing que nous avons développée parallèlement.

3. Jar exécutable et chemins relatifs

Le chargement et la sauvegarde des parties de notre jeu se fait via des fichiers textes .map et .sav qui sont tous référencés en chemins relatifs dans les fichiers source de notre programme (les fichiers n'étant pas dans le répertoire de source, nous ne pouvons faire appel au getResource()).

Ce mécanisme entraîne un problème au niveau du lancement de notre programme depuis l'archive jar exécutable car il faut que la commande « **java –jar** » soit exécutée depuis le répertoire racine du projet afin que les chemins relatifs restent corrects et que les cartes se chargent et se sauvegardent normalement.

4. Ralentissement de la Thread principale pour le mouvement

Pour plus de convivialité graphique avec l'utilisateur, nous avons décidé de ralentir les mouvements des unités sur la carte afin que les déplacements ne se fassent pas d'un seul coup sans que l'œil puisse suivre le mouvement.

Cette méthode est réalisée grâce à l'appel **Thread.sleep()** qu'il est déconseillé d'utiliser. Cependant, dans notre cas, il nous semblait difficile d'envisager une autre solution aussi peu couteuse en développement pour ralentir l'affichage graphique. Ce point particulier constitue potentiellement une limite du programme.

5. « lawrence » et les différentes cartes

Par défaut, la librairie graphique « **lawrence** » affiche la carte par défaut de 100x100 pour HMM2000 afin d'assurer que toutes les autres cartes pourront être contenues dans le même « **GridModel** » sans avoir à relancer un nouveau « **GridPane** ».

C'est pour cette raison que toutes les cartes s'affichent dans la même fenêtre de taille fixe. Ce point provoque plus une limite graphique que fonctionnelle puisque la fenêtre « **lawrence** » est toujours scrollable et comblée avec des sprites neutres même si la carte est réduite. De plus, aucune carte ne pourra posséder une taille supérieure à 100x100, sauf si nous modifions la carte par défaut.