Projet Logiciel « Fantasy World »

Kusan THANABALASINGAM – Benoît LAFON – Yifei ZHANG

Illustration 1 : Fantasy World

SOMMAIRE

1 Objectif

1.1 Présentation générale

L'objectif de ce projet est la réalisation d'un jeu de stratégie tour par tour inspiré de jeux semblable à « Dofus ».

Illustration : 2 Dofus

Illustration : 3- Final Fantasy Tactics

Illustration : 4 - Exemple de sélection des personnages

1.2 Règles du jeu

Le joueur déplace une équipe de personnages sur une carte et doit vaincre l'équipe adverse. Chaque personnage a des compétences et des propriétés (points de vies, attaque, défense, etc.) différentes. Le joueur avance de niveau en niveau, le joueur ayant la possibilité de refaire un niveau pour augmenter son expérience.
Ainsi une montée de niveau donne des points de compétences au personnage que le joueur peut librement attribuer (PV, attaque, défense, etc.). De plus le joueur gagne de nouveaux personnages et sorts au fur et à mesure de son avancée.

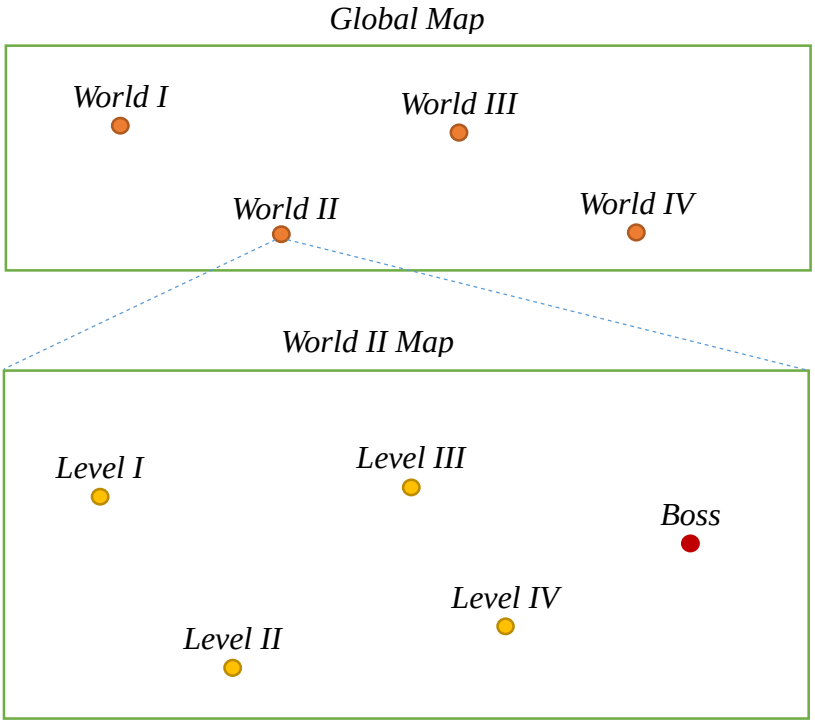
- Fonctionnement des cartes et des niveaux :

-

Le jeu est constitué de différentes cartes dont la « *Global Map* » est la plus générale. La « *Global Map* » recense tous les mondes possibles.
Chaque monde est constitué d'une carte qui lui est propre tel que la « *World I Map* » pour le premier monde. Sur chacune des cartes est représentée des niveaux que le joueur doit surmonter pour atteindre le prochain niveau. Une carte s'achève avec un *boss* de fin.

- Règles générales :
- Le monde *n+1* est débloqué si le *boss* du monde *n* a été vaincu.
- Le *boss* d'un monde *n* est débloqué si tous les niveaux de la même carte ont été terminés avec succès.
- Le niveau *n+1* d'un monde *m* est débloqué est débloqué si le niveau *n* a été terminé avec succès.

Enfin, un monde spécial contenant une map 1v1 (en réseau) pour les affrontements entre joueurs est disponible.



- Constitution et évolution des personnages :

Le joueur démarre une partie avec un seul personnage. Au fur et à mesure de l'avancée des mondes, le joueur gagne de nouveaux personnages.
Chaque personnage a des attributs et spécificités qui leurs sont propre en fonction de leur classe (points de vies, attaque, défense…).
Au terme d'un combat, chaque personnage gagne de l'expérience en fonction de son utilisation lors du combat ainsi que du niveau de l'ennemi.
Le joueur peut rejouer un même combat autant de fois qu'il le souhaite pour monter le niveau de ses personnages.
Le passage au niveau supérieur d'un personnage est accompagné d'un certain nombre de point que le joueur est libre d'attribuer au personnage.
De plus, le passage au niveau supérieur peut s'accompagner de l'apprentissage de nouvelles compétences.

- Déroulement d'un combat

Le déroulement d'un combat s'effectue en équipe et au tour par tour sur une carte composée de cases prédéfinis. L'objectif étant de vaincre l'équipe adverse. À chaque tour le joueur doit choisir une action selon le personnage sélectionné sachant que quatre personnages au maximum peuvent effectuer une action.
Chaque attaque/sort a des portés différents, ainsi le joueur doit placer son personnage à porter du personnage qu'il veut atteindre pour pouvoir effectuer l'action désiré sachant que chaque personnage possède un nombre de cases de déplacement possible fixe et réinitialiser à chaque tour.
Une fois le nombre de personnages d'une équipe réduit à zéro ; le combat s'achève. Le joueur peut ainsi passer au niveau suivant s'il gagne ou retenter un ancien niveau pour s'améliorer.

1.3 Conception Logiciel

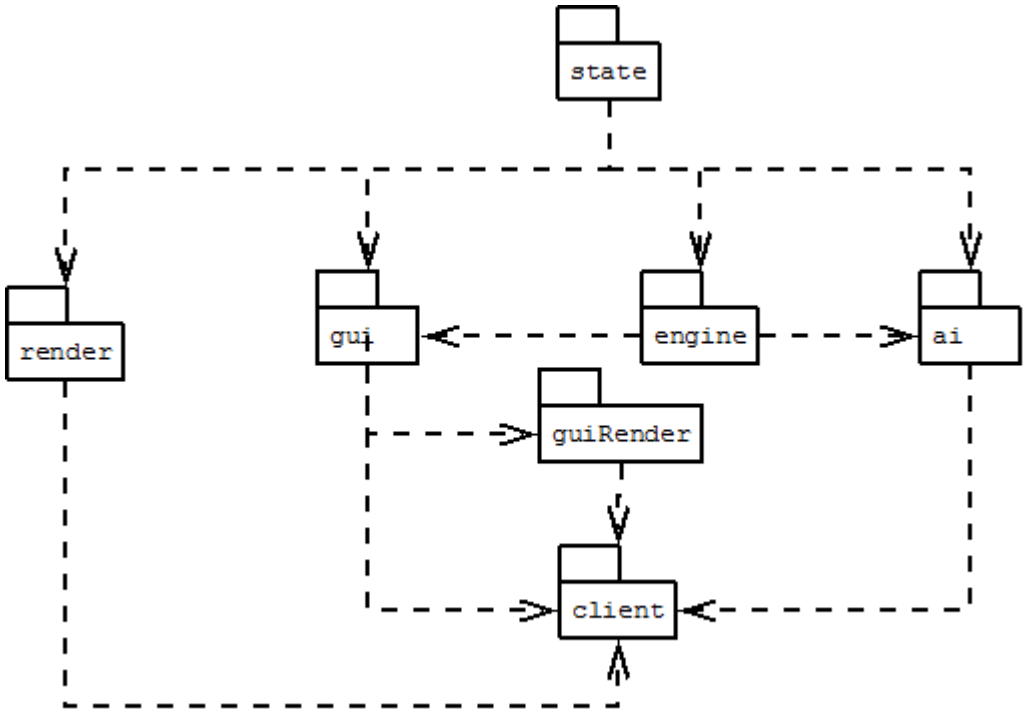


Illustration : 5 - Diagramme des packages

2 Description et conception des états

2.1 Description des états

Un état du jeu est formé par un état dit globale composés de deux sous-états correspondant à un état concernant l'état d'un niveau « Level » et d'un état de sélection du monde « World ».
Une propriété présente dans l'état global permettra de définir le sous-état actuellement utilisé.
De plus, l'état contiendra les informations relatives au joueur (équipe, score, etc.).

2.1.1 Description du sous-état « Level »

Le sous-état « Level » est formé par un ensemble d'éléments fixes (le niveau), d'un ensemble d'éléments mobiles (personnages du joueur et personnages de l'ennemi) et d'une fenêtre.
Tous les éléments possèdent les propriétés suivantes :

- Coordonnées (x,y) dans la grille
- Identifiant de type d'éléments : ce nombre indique la nature de l'élément (ex : classe)
- Si l'élément est « actif » ou non (l'élément actif sera par exemple en surbrillance). Une seule case peut être actif à la fois.

2.1.1.1 Etat éléments fixes

La carte est formée par une grille d'éléments nommée « cases ». La taille de cette grille est fixée au démarrage du niveau. Les types de cases sont :

- ***Cases « Obstacle » :*** les cases « obstacle » sont des éléments infranchissables pour les éléments mobiles. Le choix de la texture est purement esthétique, et n'a pas d'influence sur l'évolution du jeu.
- ***Cases « Espace » :*** les cases « espace » sont les éléments franchissables par les éléments mobiles.

On considère les types de cases « espace » suivants :

- Les espaces « vides »
- Les espaces « énergie », qui définissent une boule d'énergie.
- Les espaces « départ », qui définissent une position initiale possible pour chaque personnage.

2.1.1.2 Etat éléments mobiles

Les éléments mobiles possèdent :

- Une direction (aucune, gauche, droite, haute ou basse)
- Une position. Une position à zéro signifie que l'élément est exactement sur la case ; pour les autres valeurs, cela signifie qu'il est entre deux cases (l'actuelle et celle définie par la direction de l'élément). Lorsque la position est égale à la vitesse, alors l'élément passe à la position suivante. Ainsi, plus la valeur numérique de vitesse est grande, plus le personnage aura un déplacement lent. Ce système est choisi pour pouvoir être toujours synchronisé avec une horloge globale.

- ***Elément mobile «PlayerCharacter» :*** ces éléments sont dirigés par le joueur. Ces éléments ont les propriétés suivantes :
  - Statut :
    - Statut « normal » : cas le plus courant, où le personnage peut se déplacer d'un nombre de cases prédéfini sur la carte et effectuer une action à chaque tour.
    - Statut « super » : cas où le héros double le nombre de cases prédéfini de déplacement et peut effectuer deux actions à chaque tour et ce pendant un certain nombre de tours.
    - Statut « mort » : cas où le personnage a été vaincu par un ennemi.
  - Une information sur si l'élément est sélectionnable ou non
  - Une information sur si l'élément est sélectionné ou non

- ***Elément mobile «Monster».*** Ces éléments sont commandés par la propriété de direction provenant d'un IA. Ces éléments possèdent également deux propriétés particulières. La première est le « type » de monstre ; affectant ses caractéristiques. La seconde propriété particulière est le « statut », qui peut prendre les valeurs suivantes :
  - Statut « normal » : cas le plus courant, où le monstre attaque les personnages adverses sans se soucier du nombre de dégâts qu'il inflige.
  - Statut « mort » : cas où le fantôme a été vaincu par un personnage adversaire.

2.1.1.3 Etat fenêtre d'information

La fenêtre d'information est une « fenêtre » ayant les propriétés suivantes :
- Cordonnées x,y
- Texte et/ou images

Cette « fenêtre » sera à afficher lorsqu'un personnage est sélectionné. La présence ou non de la fenêtre pourra avoir des répercussions sur l'état (ex : pause des animations des personnages lorsque la fenêtre est ouverte).

2.1.1.4 Etat général du sous-état « Level »

De plus, l'état possède les propriétés générales suivantes :
- Epoque : représente « l'heure » correspondant à l'état, ie c'est le nombre de « tic » de l'horloge globale depuis le depuis de la partie.
- Vitesse : le nombre d'époque par seconde, ie la vitesse à laquelle l'état du jeu est mis à jour
- Tour actuel (tour du joueur ou de l'ennemi)
- Nombre de personnage restant dans l'équipe joueur
- Nombre de personnage restant dans l'équipe ennemi
- Une id correspondant au niveau (permettra de mettre à jour le second sous-état World)

2.1.2 Description du sous-état « World »

Le sous-état « World » est composé d'une liste d'éléments ; éléments pouvant se trouver n'importe où sur l'écran. Ces éléments ont ainsi des propriétés d'id et de cordonnées.

2.1.2.1 Etat éléments ElementLevel

Les éléments « ElementLevel » de l'état « World » auront les propriétés suivantes :

- Difficulté : définit la difficulté du niveau représenté par cet élément
- Verrouillé : définit si le niveau représenté par cet élément est verrouillé ou non

2.1.2.2 Etat général du sous-état « World »

De plus, l'état possède les propriétés générales suivantes :

- Epoque : représente « l'heure » correspondant à l'état du sous-état, ie c'est le nombre de « tic » de l'horloge globale depuis le depuis de cet état.
- Vitesse : le nombre d'époque par seconde, ie la vitesse à laquelle l'état du sous-état est mis à jour
- Le nombre de niveau restant que le joueur n'a pas encore terminés.

2.2 Conception logiciel

Le diagramme des classes pour les états est présenté en Illustration 4, dont nous pouvons mettre en évidence les groupes de classes suivants :
State (en rouge) : État principal contenant les deux sous-états, sa propriété currentState permettra d'avoir l'information sur quel sous-état le jeu se trouve.

- **Pour le sous-état « Level » :**

Classes ElementLevel (en vert) : toute la hiérarchie des classes filles d'éléments permettent de représenter les différentes catégories et types d'élément. Nous utiliserons ainsi le polymorphisme pour les manipuler.
Fabrique d'éléments (en orange) : dans le but de pouvoir fabriquer facilement des instances d'Element du sous-état « Level », nous utilisons la classe ElementFactory. Cette classe, qui suit le patron de conception Abstract Factory, permet de créer n'importe quelle instance non abstraite à partir d'un caractère.
Dans le but de faciliter l'enregistrement des différentes instanciations possibles, nous avons créé le couple de classes AElementAlloc et ElementAlloc. La première est une classe abstraite dont le seul but est de renvoyer une nouvelle instance. La seconde sert à créer des implantations de la première pour une classe et une propriété d'identification particulière.
La carte monde étant unique, pour le sous-état « World » n'aura pas de fabrique d'éléments.
ConteneursLevel (en rose) : Viennent ensuite les classes StateLevel, ElementList et ElementGrid qui permettent de contenir des ensembles d'éléments ou une fenêtre. ElementList contient une liste d'éléments, et ElementGrid étends ce conteneur pour lui ajouter des fonctions permettant de gérer une grille. Enfin, la classe State est le conteneur principal, avec une grille contenant tous les éléments.

- **Pour le sous-état « World » :**

Classes ElementWorld (en bleu) : Toute la hiérarchie des classes filles d'éléments permettent de représenter les différentes catégories et types d'élément. Nous utiliserons ainsi le polymorphisme pour les manipuler.
ConteneursWorld (en violet) : Viennent ensuite les classes WorldState et ElementListWorld qui permettent de contenir des ensembles d'éléments ou une fenêtre. ElementListWorld contient une liste d'éléments. Enfin, la classe State est le conteneur principal, avec une liste contenant tous les ElementWorld.

2.3 Conception logiciel : extension pour le rendu

Observateurs de changements. Dans le diagramme de classes, nous utilisons des observateurs implantent l'interface StateObserver pour être avertis des changements de propriétés d'état. Pour connaître la nature du changement, ils analysent l'instance de StateEvent. La conception de ces outils suit le patron Observer.

2.4 Conception logiciel : extension pour le moteur de jeu

Au sein des différentes classes d'Element, ainsi que leurs conteneurs comme ElementList et LevelState, nous avons ajouté des méthodes de clonage et de comparaison :

- Méthodes clone() : elles renvoie une copie complète (les sous-éléments sont également copiés) de l'instance.
- Méthodes equals() : elles permettent de comparer l'instance actuelle avec une autre, et renvoie true si les deux instances ont un contenu identique.

2.5 Ressources

Pour le sous-état « Level », les niveaux sont codés en fichier texte, puis traduits en instance d’éléments grâce à la factory.

Exemple d’un fichier texte représentant le niveau 1 :

223222232223223322232223223323
232000000000000000000000022322
220000200000000000000002000000032
200000000000030000000000000023
300000000000000000030000000002
200000003000000000000000022000003
200000000000000000000000030000003
300000000000000000000000330000003
200020003000000000000000000000003
200000000000000300000000000000003
200000000000000000000002000000002
200003000000030000000000000000002
300000000011113333000000000000002
220000000111111133000000000000023
222000001111111133000000000002232
222333223222322322322322322323232

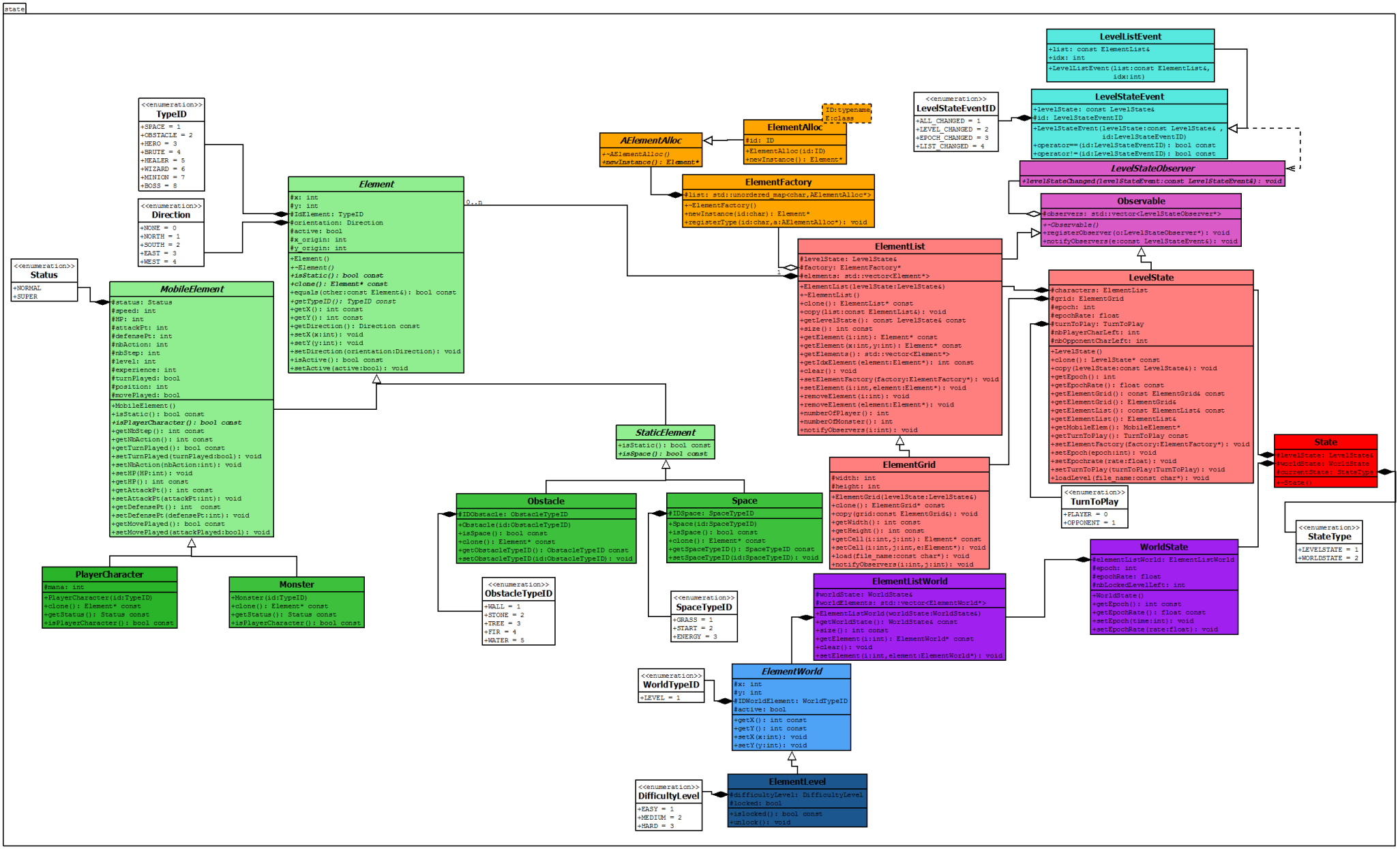


Illustration : 6 - Diagramme des classes d'état

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Le jeu Fantasy World possède deux niveaux : *LevelState* et *WorldState.* La stratégie de rendu pour les deux niveaux est identique avec des fonctionnalités avancées pour le rendu *LevelState* (matrice de position).

WorldState. Le rendu *WorldState* est le plus simple car chaque animation est lié à l'état et sont instantanées (mise à jour des données de chaque personnage, déplacement d'un curseur d'un monde à un autre, etc…) Ainsi, seule une horloge suffira pour gérer les changements d'état et les animations. De plus, étant donnée les contraintes peut importantes de ce rendu sur la position de chaque élément, une matrice de position n'a pas lieu d'être simplifiant ainsi la complexité de ce menu.

Chaque scène est découpée en plusieurs plans (ou « layers ») : un plan pour le fond d'écran (carte du monde), un plan relatif à diverses informations (personnages, accès au mode multijoueur, etc…), un plan pour l'élément mobile ; le curseur pouvant se déplacer de niveau en niveau sans positions intermédiaires. Au curseur est associé une boite de dialogue indiquant l'état et les actions possible d'un niveau donné. Enfin, un dernier plan regroupant les pop-ups (chargement, attente d'un joueur en mode multijoueur, etc…).

LevelState. Le rendu *LevelState* est plus complexe que le rendu *WorldState* puisque celui-ci contient des animations qui ne dépendent pas complétement à un état ; ces états seront définis comme des sous états fictifs. En effet, bien que déclenché par les états, les animations de mouvement des personnages devront être associées à leur propre horloge indépendamment de celle des états.

La particularité du rendu *LevelState* est que la fenêtre est scindée horizontalement en deux parties.

La partie haute contiendra le plateau du jeu et la partie basse ; les informations relatives à l'élément sélectionné.

- La partie *plateau* est décomposée comme suit :

Chaque scène est découpée en plusieurs plans (ou « layers ») : un plan pour le niveau (mur, arbres, rochers, boule d'énergie, etc.) et un plan pour les éléments mobiles (personnages, monstres). Chaque plan contiendra deux informations bas-niveau : une unique texture contenant les tuiles (ou « tiles »), et une unique matrice avec la position des éléments et les coordonnées dans la texture. En conséquence, chaque plan ne pourra rendre que les éléments dont les tuiles sont présentes dans la texture associée.

Pour la formation de ces informations bas-niveau, la première idée est d'observer l'état à rendre, et de réagir lorsqu'un changement se produit. Si le changement dans l'état donne lieu à un changement permanent dans le rendu, on met à jour le morceau de la matrice du plan correspondant. Pour les changements non permanent, comme les animations et/ou les éléments mobiles, nous tiendrons à jour une liste d'éléments visuels à mettre à jour (= modifier la matrice du plan) automatiquement à chaque rendu d'un nouveau frame.

- La partie *information* est décomposée comme suit :

Chaque scène est découpée en plusieurs plans (ou « layers ») : un plan pour le fond d'écran et un plan pour les informations (vies, scores, etc.) et les actions réalisables par le personnage sélectionné. La mise à jour des informations est synchronisée sur l'horloge d'état.

3.2 Conception logiciel

Le diagramme des classes pour le rendu général, indépendante de toute librairie graphique, est présenté en Illustration 5.

Plans et Surfaces. Les informations de bases se situent dans la classe *Layer*. Le principal objectif des instances de *Layer* est de donner les informations basiques pour former les éléments bas-niveau. Ces informations sont données à une implantation de *Surface*. La première information donnée est la texture

du plan, via la méthode *loadTexture()*. Les informations qui permettrons à l'implantation de *Surface* de former la matrice des positions seront données via la méthode *setSprite()*.
Scène. Toutes les instances de *Layer* seront regroupés au sein d'une instance *Scene*. Les instances de cette classe seront liées à un état particulier et permettront de mettre à jour les instances de *Layer* via les informations récupérées par l'interface *StateObserver*.
Tuiles. La classe *Tile* ainsi que ses filles ont pour rôle la définition de tuiles au sein d'une texture particulière, ainsi que les animations que l'on peut former avec. La classe *StaticTile* stocke les coordonnées d'une unique tuile, et la classe *AnimatedTile* stocke un ensemble de tuiles. Enfin, les implantations de la classe *TileSet* stocke l'ensemble des tuiles et animations possibles pour un plan donné.

3.3 Conception logiciel : extension pour les animations

Animations. Les animations sont gérées par les instances de la classe *Animation*. Chaque plan tient une liste de ces animations, et le client graphique fait appel aux méthodes de mise à jour pour faire évoluer ses surfaces. Nous faisons ce choix car certaines évolutions dans l'affichage ne dépendent pas d'un changement d'état.
Pour le cas des animations de mouvement, par exemple lorsqu'un personnage se déplace, nous avons ajouté toutes les informations permettant d'afficher le déplacement (direction et vitesse) sans dépendre de l'état. Ainsi, lorsqu'une information de mouvement parvient au plan, elle est pleinement définie dans une instance d'*Animation* et peut se prolonger de manière autonome.
En outre, cette animation est synchronisée avec les changement d'état grâce à la méthode *sync()*, qui permet de transmettre le moment précis où un état vient de changer.

3.4 Ressources

Exemple de texture employée :

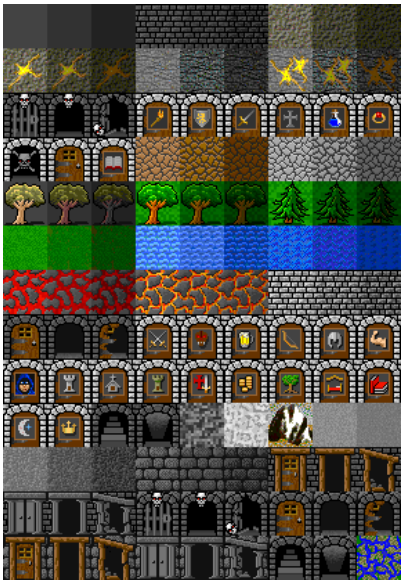


Illustration : 7 - Exemple de texture

3.5 Exemple de rendu

Echantillon d'un rendu du jeu « Fantasy World » :

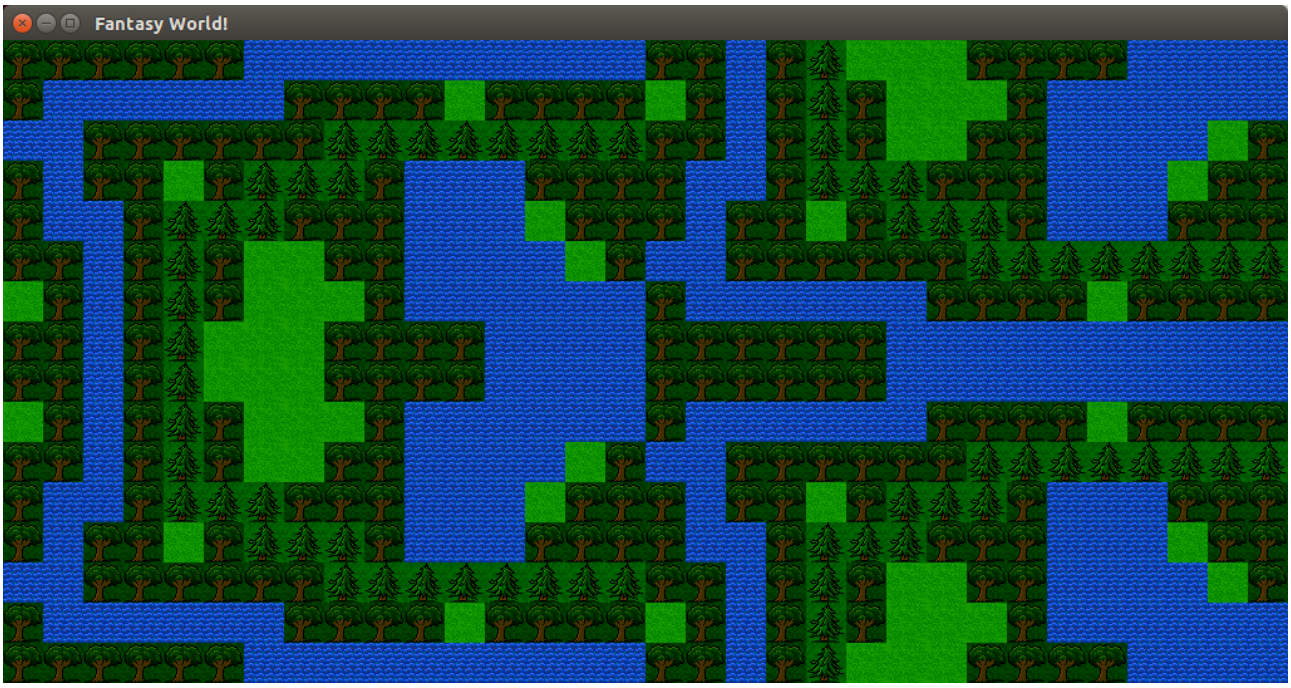


Illustration : 8 - Fantasy World 1.3

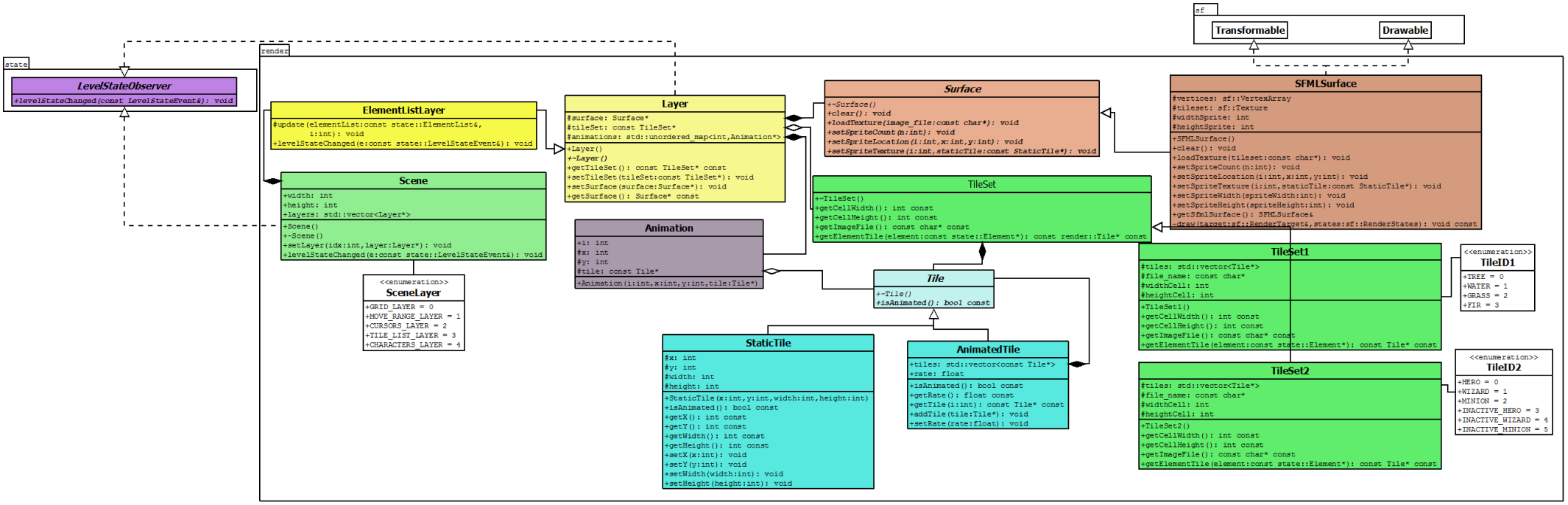


Illustration : 9 - Diagramme des classes pour le rendu

4 Règles de changement d'états et moteur de jeu

4.1 Epoques

Il y a un changement d'époque à chaque action construite par le moteur de jeu. Une liste d'action est créée

4.2 Changements extérieurs

Les changements extérieurs sont provoqués par des commandes extérieurs, comme la pression sur une touche ou le clic sur un bouton de la souris :

- Commandes principales « LOAD » :
  - « Charger un niveau » : Fabrication un état initial à partir d'un fichier
  - « Nouvelle partie » : Tout est remis à l'état initial
  - « Quitter niveau » : Quitte le niveau et retourne sur la World map
- Commandes « Mode » : Modification du mode actuel du jeu :
  - « Normal »
  - « Pause »
  - etc.
- Commandes « Move » : Permet de modifier la position d'un personnage
  - Character : character dont on souhaite modifier la position
  - x, y : position de destination (une position similaire entraine un fin de tour)

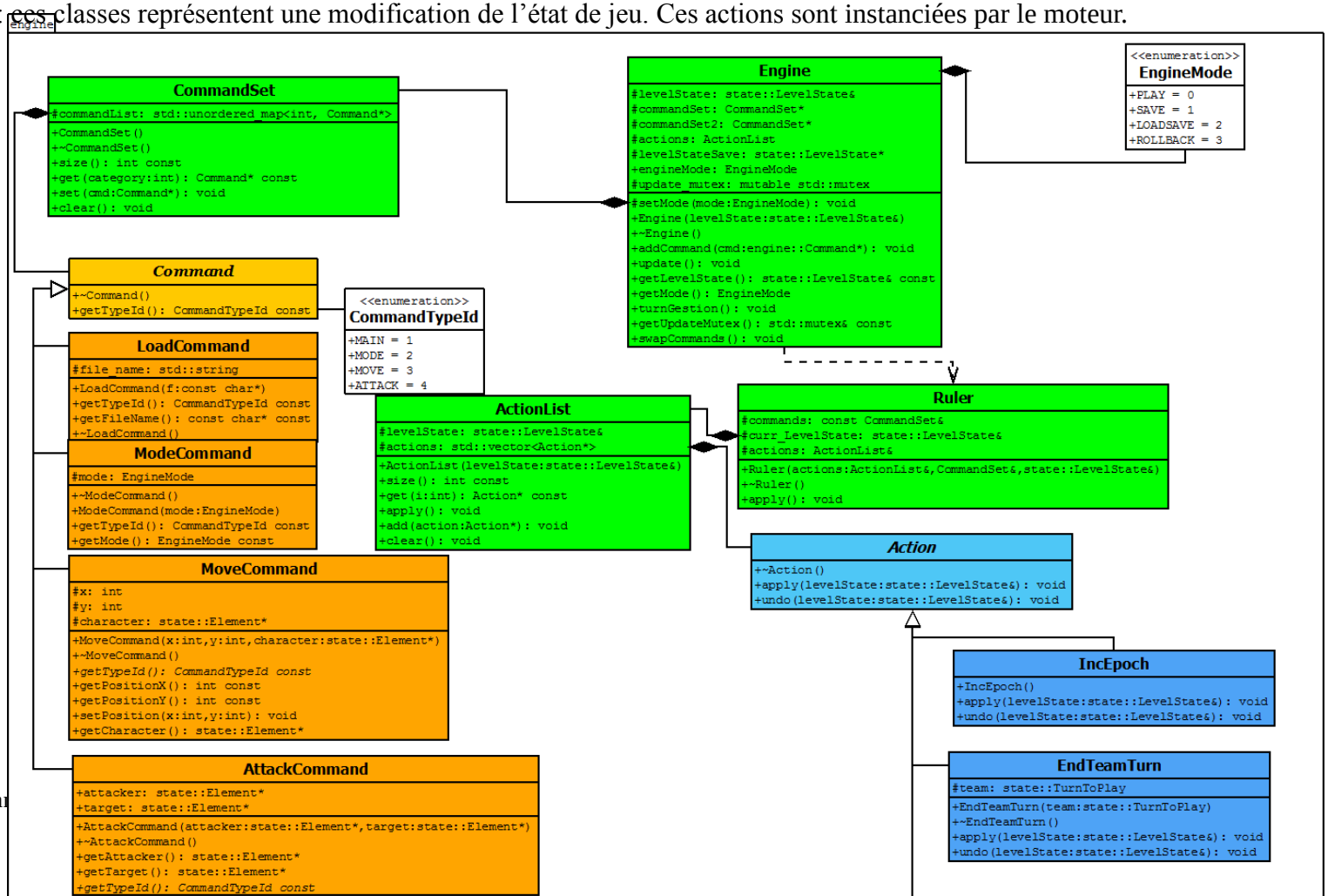
4.3 Changements autonomes

Les changements autonomes sont appliqués à chaque création ou mise à jour d'un état, après les changements extérieurs. Ils sont exécutés dans l'ordre suivant :

1. Si le personnage est en mode « MOVE » et qu'une case adjacente deviens actif, déplacer le personnage à la case actif si cela est possible et incrémenter son compteur de pas.
2. Si le personnage « HERO » est sur une case « ENERGY», il y a augmentation du compteur d'energie et actualisation du TypeID (« Energy » à « Empty »).
3. Si tous les personnages mobiles de l'équipe sont en mode « DONE », passer au « tour » de l'autre équipe.
4. Si une équipe ne possède plus de personnages mobiles, l'autre équipe est déclaré gagnante, il y a mise à jour des propriétés des personnages de l'équipe gagnante, suppression de l'état « Level » et l'état courant (currentState) passe à « WorldState ».

4.4 Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en Illustration 6. L'ensemble du moteur de jeu repose sur un patron de conception de type Command, et a pour but la mise en œuvre différée de commandes extérieures sur l'état du jeu.
Classes Command : le rôle de ces classes est de représenter une commande extérieure. Provenant par exemple du clavier ou de la souris. Ces classes ne gèrent pas l'origine des commandes.
Classe Engine : elle contient une instance de commandSet contenant une liste de commandes. Ces commandes sont ensuite envoyées à une instance de Ruler. Classe qui par la suite créera une liste d'actions composés d'instances Action qui seront exécutés (update) en fonction des commandes extérieurs et des mises à jours automatique de l'état en fonction des règles.
Nous avons ainsi un moteur qui modifie l'état de manière régulière indépendamment du rendu.
Classes Action : ces classes représentent une modification de l'état de jeu. Ces actions sont instanciées par le moteur.



5 L’interface graphique (IHM)

5.1 Etat

L’IHM regroupe des informations permettant aux joueurs de prendre connaissance de l’état du jeu ainsi que d’interagir avec le jeu à l’aide d’un curseur et d’autres éléments ne correspondant pas à l’état du jeu. La classe IHM en elle-même est construite de manière à être indépendant de toute librairie graphique ; la classe sfmlClient se chargeant de modifier l’état de l’IHM.
L’IHM est ainsi le moyen pour un utilisateur de créer les commandes à destination du moteur de jeu.

5.1.1 Description de l’état

L’état de l’IHM est formé par un ensemble d’éléments représentant un curseur de sélection et toutes les données n’étant pas propre à l’état de jeu.

5.1.2 Conception logiciel

Le diagramme des classes par l’illustration 11 définit les différentes classes de l’état de l’IHM. Nous pouvons mettre en évidence les groupes de classes suivants :

Elements (en rouge) : L’ensemble des classes représentant un élément. Nous avons une classe Cursor représentant le curseur et des élements tuiles représentant des tuiles (pour la représentation des déplacements autorisés par exemple).

Conteneurs (en vert) : Les classes GuiElementList, GuiMoveRange et GUI sont des conteneurs d’éléments.

5.1.3 Extension pour le rendu de l’IHM

Observateurs de changements. Dans le diagramme de classes, nous utilisons des observateurs implantent l'interface GUIObserver pour être avertis des changements de propriétés d'état. Pour connaître la nature du changement, ils analysent l'instance de GUIEvent. La conception de ces outils suit le patron Observer.

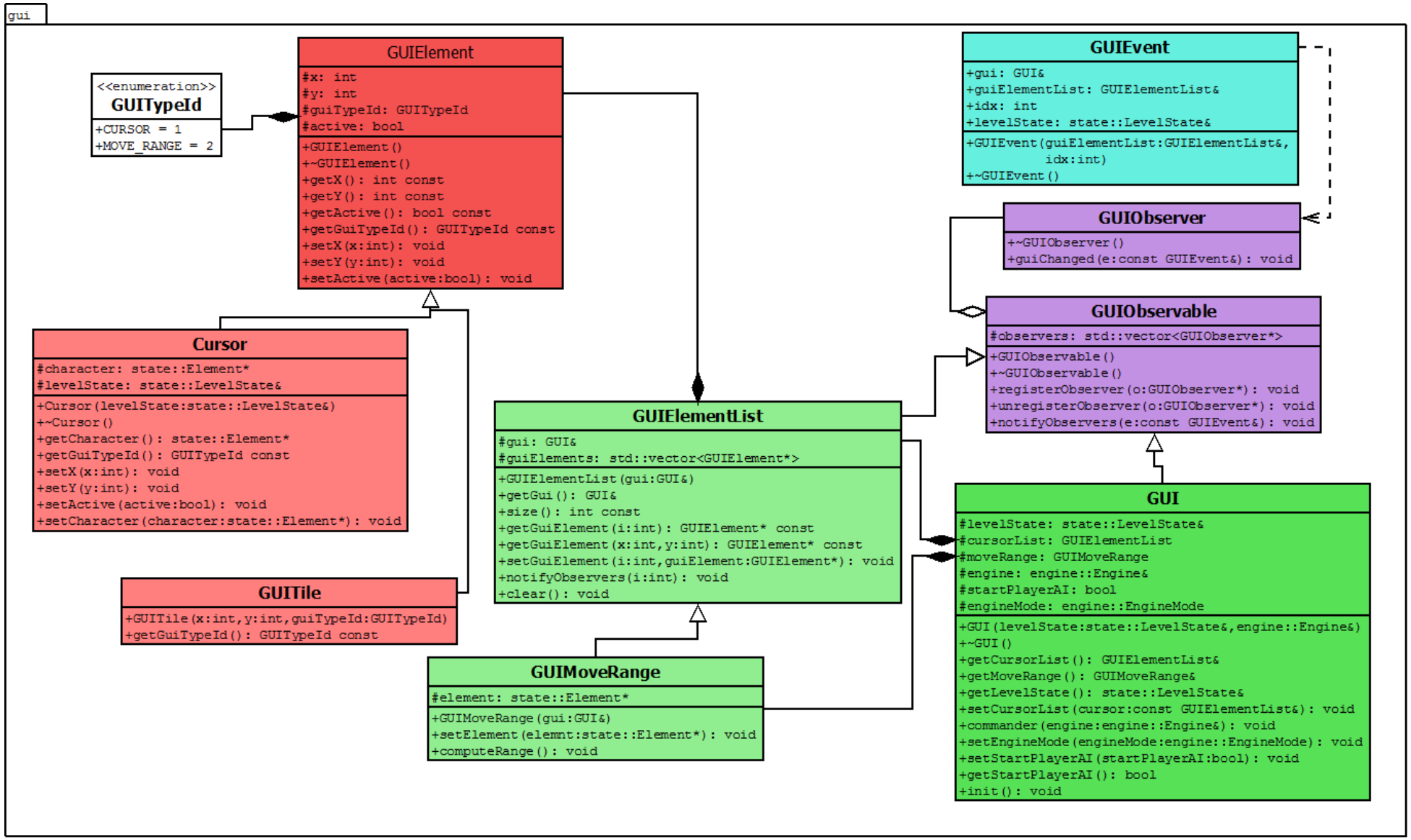


Illustration : 11- Diagramme des classes pour l'IHM

5.2 IHM Rendu

Un rendu IHM indépendant du rendu de l'état a été implémenté. Ce dernier observera et rendra uniquement compte des informations de rendu liés à l'état de l'IHM.

5.3 Stratégie de rendu d'un état de l'IHM

La stratégie de rendu d'un état de l'IHM est semblable à celui de l'état. En ce sens, le rendu de l'IHM propose les mêmes fonctionnalités avancées telles qu'une matrice de position.

GUI. Le rendu GUI est indépendant de l'état. Ainsi une horloge propre aux animations du GUI devra leur être consacrée. Ces animations inclues le déplacement du curseur ou encore l'affichage des déplacements possibles en fonction du nombre de pas propre à chaque personnage du jeu.
La grille du jeu est utilisée pour le placement des éléments du GUI et est décomposé suivant des couches :

- Une couche (ou « GUILayers ») pour les indications des déplacements possibles ou encore de la portée d'une attaque.
- Une couche contenant les informations du curseur sur la grille.

Chaque couche contient des informations deux informations bas-niveau : une unique texture contenant les tuiles (ou « tiles »), et une unique matrice avec la position des éléments et les coordonnées dans la texture. En conséquence, chaque plan ne pourra rendre que les éléments dont les tuiles sont présentes dans la texture associée.
Ces dernières fonctionnalités proviennent de classes déjà implémentées dans le rendu de l'état. Bien que le rendu de l'état et de l'IHM complétement indépendant, les classes *Surface* et *TileSet* sont suffisamment générique pour pouvoir être utilisé pour les deux rendus.
Toutefois, l'observation de l'état à rendre est complétement différent car les éléments constituants l'état et l'état de l'IHM leurs sont propres.
Néanmoins le procédé est le même ; l'idée est d'observer l'état à rendre, et de réagir lorsqu'un changement se produit. Si le changement dans l'état de l'IHM donne lieu à un changement permanent dans le rendu IHM, on met à jour le morceau de la matrice du plan correspondant.

5.4 Conception logiciel

Le diagramme des classes est présenté en Illustration 12. Les groupements de classes sont les suivantes :

Plans et Surfaces. Les informations de bases se situent dans la classe GUI*Layer*. Le principal objectif des instances de GUI*Layer* est de donner les informations basiques pour former les éléments bas-niveau. Ces informations sont données à une implantation de *Surface* situé dans le rendu de l'état. La première information donnée est la texture du plan, via la méthode *loadTexture()*. Les informations qui permettrons à l'implantation de *Surface* de former la matrice des positions seront données via la méthode *setSprite()*. Les instances de cette classe seront liées à un état particulier et permettront de mettre à jour les instances via les informations récupérées par l'interface GUI*StateObserver*.

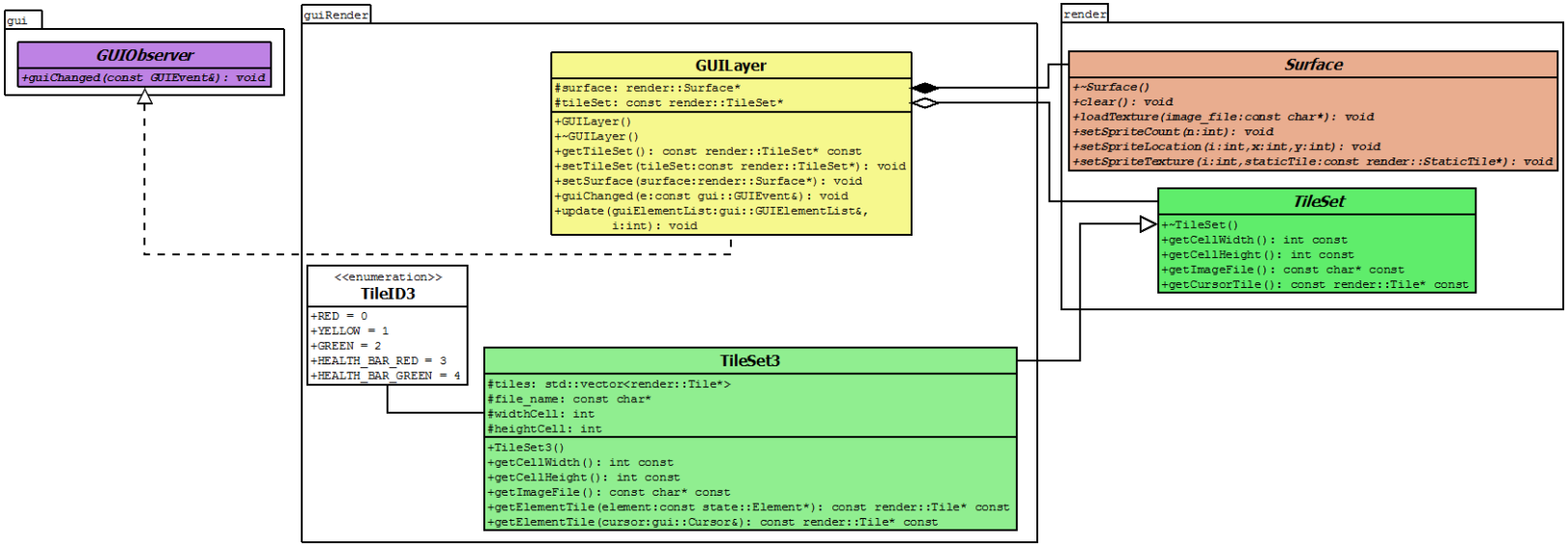


Illustration : 12 - Diagramme de classes pour le rendu de l'IHM

6 Intelligence Artificielle

6.1 Stratégies

6.1.1 Intelligence minimale

Dans le but d'avoir une sorte d'étalon, mais également un comportement par défaut lorsqu'il n'y a pas de critère pour choisir un comportement, nous proposons une intelligence extrêmement simple, basée sur les principes suivants :

- Chaque personnage d'une équipe cherche à se rapprocher du héros (ou du prochain personnage si celui-ci est mort) et ce jusqu'à portée d'attaque. Ces déplacements s'effectuent en respectant les collisions dictées par la grille du jeu c'est-à-dire en évitant les obstacles.
- Lorsqu'un personnage est en capacité d'attaquer un ennemi, il le fera jusqu'à la mort de l'ennemi ou jusqu'à sa propre mort.

6.1.2 Intelligence basée sur des heuristiques

Dans le but d'offrir un comportement plus efficace, un ensemble d'heuristiques sont proposées.
La principale amélioration concerne le choix de déplacement de l'IA. Jusqu'alors, des mouvements comportaient une partie aléatoire lors de ses déplacements et tous les personnages convergeait uniquement vers un seul personnage (le héros au départ).
Ainsi dans un souci d'authenticité, la version heuristique s'appuie sur des cartes de distance permettant à l'IA de converger efficacement vers la cible la plus proche.
Dès que l'IA sera à portée de son adversaire, l'IA ne tentera pas de mouvement de déplacement inutile autour de l'adversaire mais plutôt d'attaquer son ennemi.
Enfin l'IA pourra effectuer plusieurs actions durant son tour :

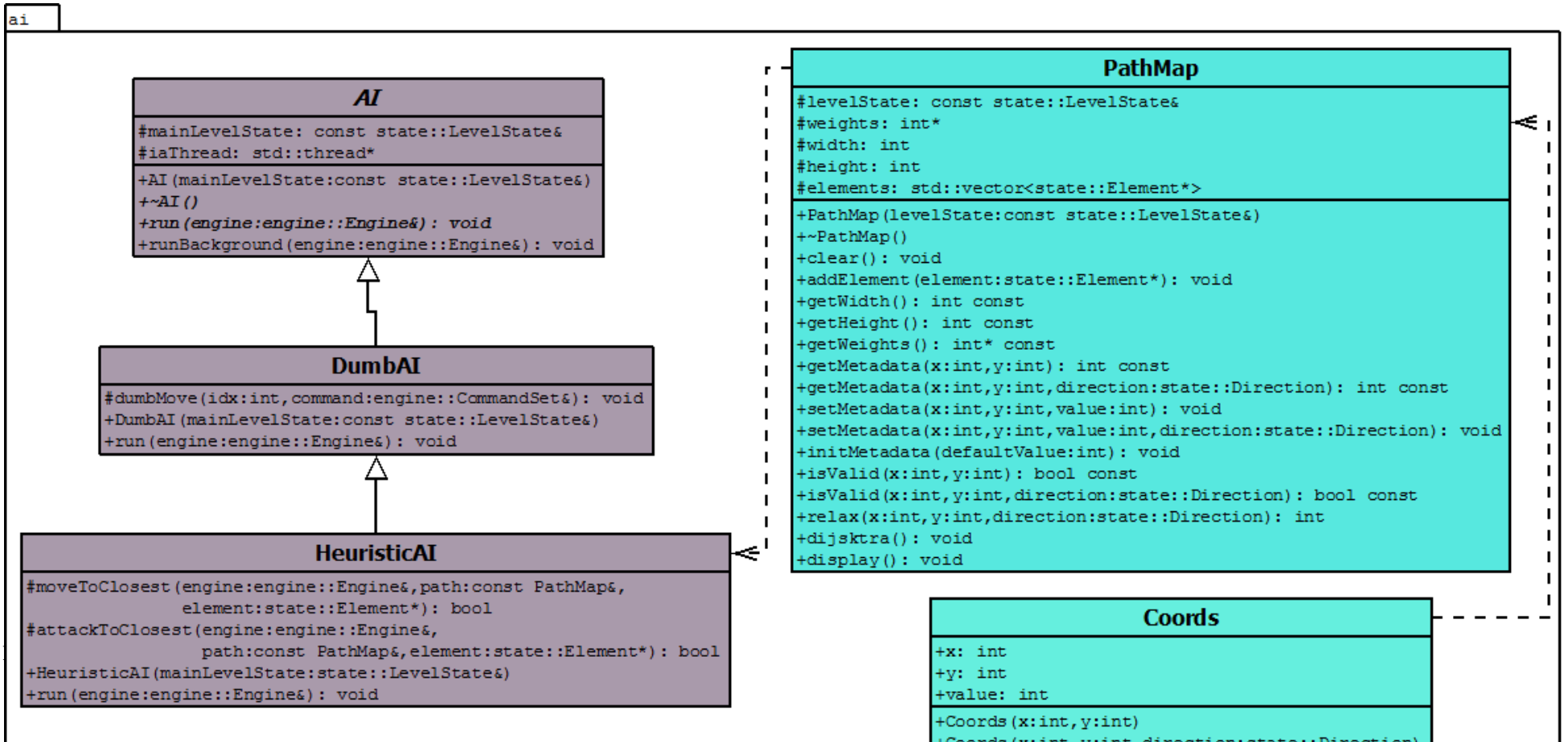
- Attaquer son adversaire si ce dernier est à portée et terminer son tour.

Tenter d'attaquer si possible sinon se déplacer et tenter d'attaquer à nouveau avant de valider son tour.

6.2 Conception logiciel

Classes AI. Toutes les formes d'intelligence artificielle implémentent la classe abstraite *AI*. Le rôle de ces classes est de fournir un ensemble de commandes à transmettre au moteur de jeu. Notons qu'il n'y a pas une instance par personnage, mais qu'une instance doit fournir les commandes pour tous les personnages. La classe *DumbAI* implante l'intelligence minimale.

PathMap. La classe *PathMap* permet de calculer une carte des distances à un ou plusieurs objectifs en utilisant l'algorithme de Dikjstra. Plus précisément, pour chaque case du niveau, on peut demander un poids qui représente la distance à ces objectifs allant de 0 (cible) à 999 (Obstacle ou personnage). Pour s'approcher d'un objectif lorsqu'on est sur une case, il suffit de choisir la case adjacente dans la limite du nombre de pas attribué au personnage qui a un plus petit poids. On peut également utiliser ces poids pour s'éloigner des objectifs, en choisissant une case avec un poids supérieur.



7 Modularisation

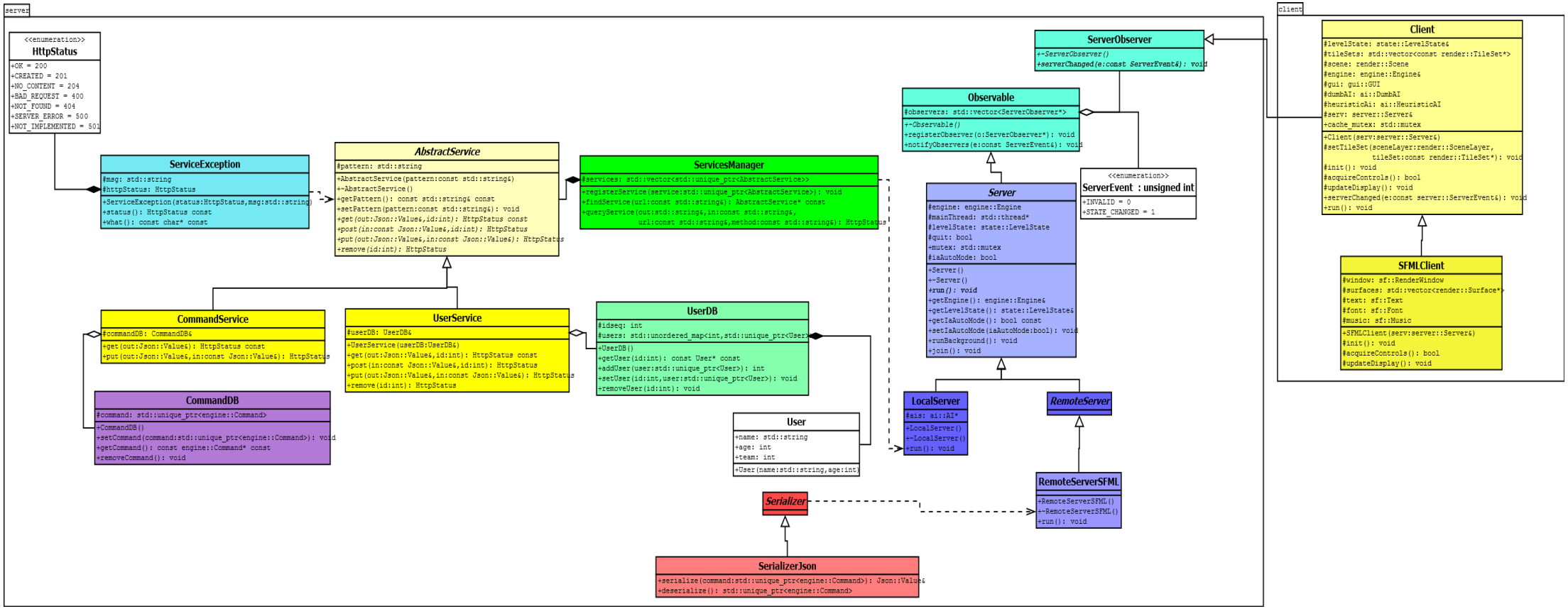


Illustration : 14- Diagramme des classes pour la modularisation

7.1 Organisation des modules

7.1.1 Répartition sur différents threads

Notre objectif ici est de placer le moteur de jeu sur un thread, puis le moteur de rendu sur un autre thread. Nous avons deux types d'information qui transitent d'un module à l'autre : les commandes et les notifications de rendu.
Commandes. Il s'agit ici des commandes de jeux, celles-ci peuvent aussi bien venir d'un appui clavier que des IA. Celles-ci peuvent arriver à l'importe quel moment, y compris lorsque l'état du jeu est mis à jour. Pour résoudre ce problème, nous utilisons un double tampon de commandes. Une liste de commande reçoit les commandes venues de tout part et à chaque nouvelle mise à jour de l'état du jeu, il y a permutation des deux tampons. Ainsi nous assurons la bonne réception des nouvelles commandes même lorsque nous sommes en train de traiter la liste des commandes.
Notifications de rendu. Nous avons l'objectif de nous tourner vers une solution avec recouvrement entre les deux modules, en proposant une approche qui le minimise autant que possible.
Pour ce faire, nous ajoutons un tampon qui va « absorber » toutes les notifications de changement qu'émet une mise à jour de l'état du jeu. Puis, lorsqu'une mise à jour est terminée, le moteur de jeu envoie un signal au moteur de rendu. Celui-ci, lorsqu'il s'apprête à envoyer ses données, regarde si ce signal a été émis. Si c'est le cas, il vide le tampon de notification pour modifier ses données graphiques avant d'effectuer ses tâches habituelles. Lors de cette étape, une mise à jour de l'état du jeu ne peut avoir lieu, puisque le moteur de rendu a besoin des données de de l'état pour mettre à jour les siennes. Nous avons donc ici un recouvrement entre les deux processus. Cependant, la quantité de mémoire et de processeur utilisée est très faible devant celles utilisées par la mise à jour de l'état du jeu et par le rendu.

7.2 Conception logiciel

Classes Server. La classe abstraite Server encapsule un moteur de jeu, et lance un thread en arrière-plan pour effectuer les mises à jours de l'état du jeu. L'implantation LocalServer est le cas où le moteur de jeu principal est sur la machine où est lancé le programme. Dans tous les cas, les instances de Server peuvent notifier certains événements grâce aux classes Observable, ServerObserver et ServerEvent qui forment un patron de conception Observer.

Classes Client. La classe abstraite Client contient tous les éléments permettant le rendu, comme la scène et les définitions des tuiles. Un maximum d'opération communes à tous les clients graphiques a été placés dans cette classe, afin de rendre la création d'un nouveau type de client plus rapide. La classe SFMLClient est un exemple de client graphique avec la librairie SFML.

CacheStateObserver. Cette implantation particulière de StateObserver nous permet de récupérer toutes les notifications émises par un moteur de jeu, dans le but de les appliquer plus tard, tel que décrit dans la section précédente. (Note : cette classe n'est pas encore implémentée, pour l'instant nous bloquons le rendu ou les modifications d'états à chaque traitement).

7.3 Extension réseau

7.3.1 : protocoles, formats et API

L'objectif ici est de placer le serveur et le client sur deux machines différentes. Pour cela nous utilisons les protocoles suivants : TCP/IP, HTTP et API REST.
L'IHM du client envoie les commandes au serveur à partir d'un service spécifique et celui-ci retourne une confirmation aux clients enregistrés. Les commandes seront sérialisées dans le format de données JSON.

Notre JSON pour la commande est composé d'un objet dont le 1er champ spécifie la commande.
Il y a ensuite un objet pour chaque type de commandes existant (LoadCommand, ModeCommand, MoveCommand, AttackCommand, cf partie Moteur). Les membres de ces objets sont les attributs de la classe correspondant.

```
{
"type": "TYPECOMMANDE",
  "LoadCommand" : {
    "file_name": "nom_fichier"
  }
  "ModeCommand" : {
    "mode": "mode"
  }
  "MoveCommand" : {
    "x": -1
    "y": -1
  }
  "AttackCommand" : {
    "attacker": adress
    "target": adress
  }
}
```

Le JSON pour les données utilisateur a le format suivant :

```
{
"team": 1,
"name": "Kusan"
}
```

Le serveur propose les services suivants :

- UserService : gestion des utilisateurs
  - get (out : Json ::Value&, id :int) : renvoie les données de l'user Id
  - post (int :const Json ::Value&, id :int) : Permet de modifier un user
  - put (out :Json :Value&, int :const Json::Value&) : Rajoute un user
  - remove(id :int) : permet de supprimer un user
- CommandService : gestion des commandes
  - Get(out :Json ::Value&) : Renvoie la dernière commande non validé
  - Put(out :Json ::Value&, in :const Json ::Value&) : Rajout d'une commande

7.3.2 Conception logiciel

Classe AbstractService. Toute la hiérarchie des classes filles d'*AbstractService* permettent de représenter les différentes catégories de services. La classe implémente les méthodes correspondent aux quatre opérations CRUD usuelles ie GET, POST, PUT et DELETE.

Classes UserService et UserDB. Ces classes représentent un service dédié à la gestion de clients et permettent de simuler une petite base de données contenant des informations sur chaque client.

Classes CommandService et CommandDB. Ces classes représentent un service dédié à la gestion de commande provenant d'un client. Les commandes sont stockées et sont envoyées au client lorsque celui-ci en fait la demande si la commande a été préalablement validée.

Classe ServiceManager. Cette classe représente un gestionnaire de service, ie sélectionne le bon service et la bonne opération à exécuter en fonction de l'url et de la méthode http qui lui ont été communiquée.

Classe ServiceExeption. Cette classe gère la levée d'exception à tout moment pour interrompre l'exécution du service en indiquant le code HTTP.