ENSEA
Ecole Nationale
Supérieure
de l'Electronique
et de ses Applications

Projet Logiciel Transversal

Anaïs COLIN – Vincent LAMBRETCHS

DuelMaster

ENSEA
Ecole Nationale Supérieure de l'Electronique et de ses Applications

Table des matières

1 Objectif 3
1.1 Présentation générale 3
1.2 Règles du jeu 3
2 Description et conception des états 5
2.1 Description des états 5
2.1.1 Etat éléments fixes 5
2.1.2 Etat éléments mobiles 6
2.1.3 Etat globale 6
2.2 Ressources 7
3 Rendu : Stratégie et Conception 11
3.1 Stratégie de rendu d'un état 11
3.2 Conception logiciel 12
3.3 Conception logiciel : extension pour les animations 12
4 Règles de changement d'états et moteur de jeu 14
4.1 Conception logiciel 14
5 Intelligence Artificielle 16
5.1 Stratégies 16
5.1.1 Intelligence minimale 18
5.1.2 Intelligence basée sur des heuristiques 18
5.2 Conception logiciel 10
6 Modularisation 19
6.1 Organisation des modules 20
6.1.1 Répartition sur différents threads 20
6.1.2 Répartition sur différentes machines 20
6.2 Conception logiciel 23

1 Objectif

1.1 Présentation générale

Présenter ici une description générale du projet. On peut s'appuyer sur des schémas ou croquis pour illustrer cette présentation. Éventuellement, proposer des projets existants et/ou captures d'écrans permettant de rapidement comprendre la nature du projet.

L'objectif de ce projet est la réalisation d'un jeu de cartes à collectionner de type « HearthStone » sur la base d'un jeu de rôle de type « Pokémon ». Le joueur incarne un personnage qui évolue dans un monde (partie Pokémon) divisé en 4 royaumes, représentant chacun un élément, et affronte des personnages par l'intermédiaire d'un jeu de carte (partie HearthStone) pour conquérir le monde.

1.2 Règles du jeu

Présenter ici une description des principales règles du jeu. Il doit y avoir suffisamment d'éléments pour pouvoir former entièrement le jeu, sans pour autant entrer dans les détails. Notez que c'est une description en « français » qui est demandé, il n'est pas question d'informatique et de programmation dans cette section.

❖ *But du jeu*

Le joueur incarne un personnage avec un deck et un pouvoir héroïque donnés. Son but est de conquérir les 4 royaumes et de devenir le « Maitre des cartes ». Il devra pour cela battre les 4 princes au DuelMaster, un jeu de carte à collectionner. Sur son chemin, plusieurs chevaliers viendront le défier.

Il existe 3 modes de jeu :

- **Mode Observation :** un joueur artificiel évolue tout seul et montre les fonctionnalités du jeu.
- **Mode Découverte :** (mode classique) le joueur évolue sans contrainte et seul sur la carte.
- **Mode Rival :** un joueur artificiel évolue en même temps que le joueur. Le premier à devenir « Maître des cartes » a gagné.

❖ *Univers du jeu*

Le jeu se décompose en 2 sous-jeux :

➢ *Partie Exploration*

Le joueur se déplace sur une carte en 2D fixe vue du dessus contenant des montagnes, des plans d'eau, des plaines ainsi que des maisons et un château par royaume. Ces décors n'influencent en aucun cas la façon de jouer : ce ne sont que des obstacles. Le monde est divisé en 4 royaumes représentant chacun un élément (feu, etc.) et régit par un prince expert en DuelMaster. Toutefois, il existe des personnages non joueur qui pourront affronter le joueur ou simplement interagir avec lui.

➢ ***Partie Duel***

Les duels se déroulent sous forme d'un jeu de cartes tour à tour appelé DuelMaster : chacun des personnages détient un deck préparé au préalable et un nombre de points de vie (=PV). Le but du combat est de réduire les PV de l'adversaire à 0.

Pour cela, les joueurs utilisent les cartes de leur deck : chacune de ces cartes dispose de caractéristiques propres à elle-même et un coût en points de combat (=PC).

Il existe 3 zones de jeu contenant des cartes :

- **La main :** chaque joueur dispose d'un ensemble de cartes visibles et jouables lors d'un tour
- **La pioche :** il s'agit d'un tas de cartes non visibles. A chaque début de tour, le joueur pioche une carte dans **sa** pioche. Une fois la pioche épuisée, le joueur à qui elle appartient perd des PV à la place de piocher.
- **Le plateau :** pour prendre effet, une carte doit être posée sur le plateau de jeu

Il existe une banque de PC qui autorise une ou plusieurs actions. A chaque début de tour, un PC supplémentaire est attribué jusqu'à un maximum.

❖ ***Evolution***

Tout au long du jeu, le joueur peut améliorer son deck. Pour cela, il existe plusieurs façons d'obtenir de nouvelles cartes :

1) Se battre contre des joueurs ou des princes
2) Gagner de l'expérience, celle-ci s'incrémente au fur et à mesure des cartes jouées
3) Remplir des quêtes

Une fois les 4 royaumes conquis, le joueur est nommé « Maître des cartes » , gagne la partie et débloque un nouveau personnage détenant un nouveau deck et un nouveau pouvoir héroïque.

2 Description et conception des états

L'objectif de cette section est une description très fine des états dans le projet. Plusieurs niveaux de descriptions sont attendus. Le premier doit être général, afin que le lecteur puisse comprendre les éléments et principes en jeux. Le niveau suivant est celui de la conception logiciel. Pour ce faire, on présente à la fois un diagramme des classes, ainsi qu'un commentaire détaillé de ce diagramme. Indiquer l'utilisation de patron de conception sera très apprécié. Notez bien que les règles de changement d'état ne sont pas attendues dans cette section, même s'il n'est pas interdit d'illustrer de temps à autre des états par leur possibles changements.

2.1 Description des états

Partie Exploration

Un état du jeu est formé par un ensemble d'éléments fixes et un ensemble d'éléments mobiles. Tous les éléments possèdent les propriétés suivantes :

- Coordonnées (x,y) dans la grille
- Identifiant de type d'élément : ce nombre indique la nature de l'élément (ie classe)

Partie Duel

Tous les éléments possèdent les propriétés suivantes :

- **Coût :** ce nombre indique le prix en Points de Combat à ce tour pour utiliser cet élément.
- **Attaque :** ce nombre indique le nombre de Points d'Attaque que possède cet élément.
- **Utilisation :** ce booléen est utilisé par les méthodes pour activer les effets.
- **Identifiant de type d'élément :** ce nombre indique la nature de l'élément (ie classe)

2.1.1 Etats éléments fixes

Partie Exploration

La carte est formée par une grille d'éléments nommés « cases ». La taille de cette grille est fixe.
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 « cadeaux », qui contiennent des cartes à récupérer

Partie Duel

Le plateau ne contient qu'un seul élément fixe nommés Case « Pouvoir ». Une Case « Pouvoir » contient le pouvoir du héros, qui est la personnification du joueur.

2.1.2 Etats éléments mobiles

Partie Exploration

Les éléments mobiles possèdent une direction (aucune, gauche, droite, haut ou bas), une vitesse, et 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.
Les éléments sont :

- **Eléments mobiles « Personnage » :** Cet élément est dirigé par le joueur qui commande la propriété de direction qu'elle provienne d'un humain ou d'une IA.
  Un personnage supplémentaire est déblocable lorsqu'une partie est gagnée

- **Eléments mobiles « Villageois » :** Cet élément est également commandé par la propriété de direction. Ils possèdent une propriété particulière appelée « status » :

  - Status « normal », cas où le villageois peut interagir avec le personnage
  - Status « combat », cas où le villageois propose au personnage un combat

Partie Duel

- **Eléments mobiles « Carte » :** Cet élément est utilisable par le joueur ou l'IA. Il possède un attribut position qui définit dans quel tableau il est contenu. Il existe quatre types de tableaux et chaque joueur possède un exemplaire de ces quatre types :

  - **Tableau « pioche »** : ce tableau contient les cartes non jouées et non jouables. Sa taille maximale est de 20.
  - **Tableau « main » :** ce tableau contient les cartes jouables et non jouées. Sa taille maximale est de 10.
  - **Tableau « terrain » :** ce tableau contient les cartes jouées. Sa taille maximale est de 7.
  - **Tableau « cimetière » :** ce tableau contient les cartes jouées qui ne sont plus présentes sur le terrain et inutilisables. Sa taille maximale est de 20.

  Il existe 3 types de cartes :

  - **Soldat :** un soldat détient un nombre de PV et peut avoir un effet particulier et se pose sur le terrain
  - **Ordre :** un ordre détient uniquement un effet particulier et ne se pose pas sur le terrain
  - **Arme :** une arme détient une durabilité et peut avoir un effet particulier et se pose sur une case particulière du terrain

2.1.3 Etat général

A l'ensemble des éléments statiques et mobiles, nous rajoutons les propriétés suivantes :

Partie Exploration

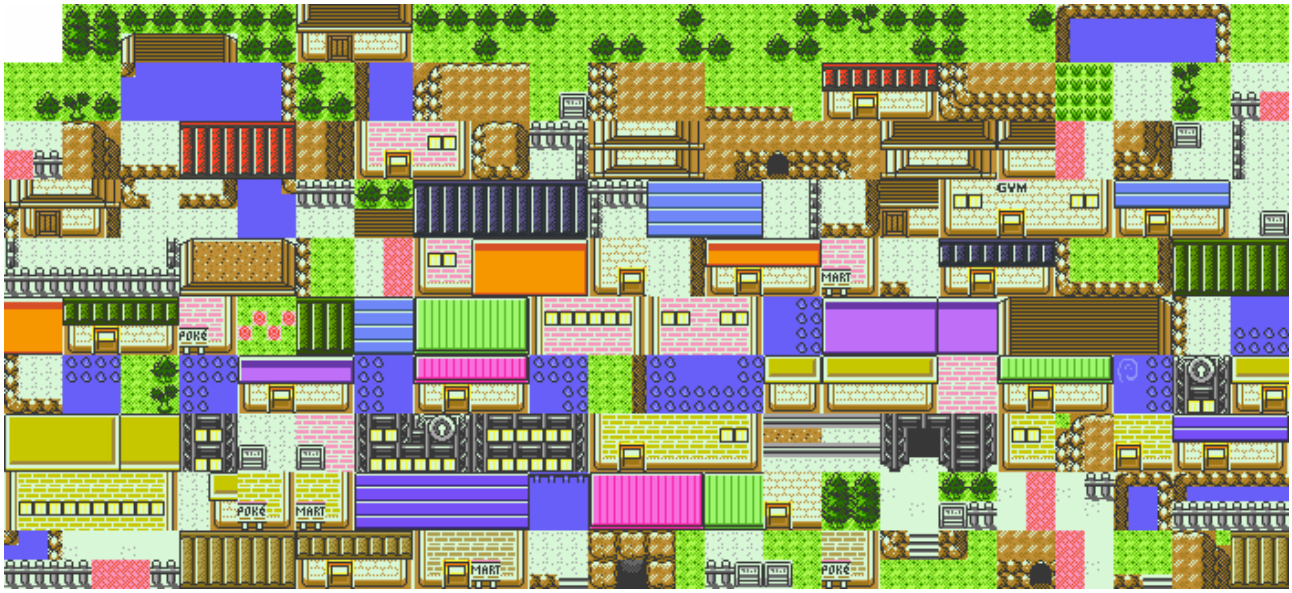
- **Epoque :** représente « l'heure » correspondant à l'état, ie c'est le nombre de « tic » de l'horloge globale depuis le depuis de la partie.
- **Vitesse :** le nombre d'époque par seconde, ie la vitesse à laquelle l'état du jeu est mis à jour
- **Compteur de victoire sur prince :** le nombre de victoires contre les princes (détermine l'avancée du jeu car il faut battre les 4 princes pour gagner une partie)

Partie Duel

- **Compteur Tour :** le nombre de tours joués depuis le début du duel
- **Compteur PV :** le nombre de points de vie (PV) restants (PV=30-dégâts)
- **Compteur Pioche :** le nombre de cartes restantes dans la pioche (Pioche=20-cartesdistribuées)
- **Compteur PC :** le nombre de points de combat restants (disponibles) lors d'un tour (CompteurPC=PCdutour-CoûtsCartesJouéespendantletour)
- **Liste des cartes :** liste complète des cartes contenues dans le jeu. Cette liste comprend les cartes sur le terrain, les cartes dans la main des joueurs, les cartes dans leur pioche respective et les cartes dans les cimetières.

2.2 Ressources

Nous utilisons pour la partie Exploration le TileSet suivant :



Et voici, le rendu de notre carte Exploration :

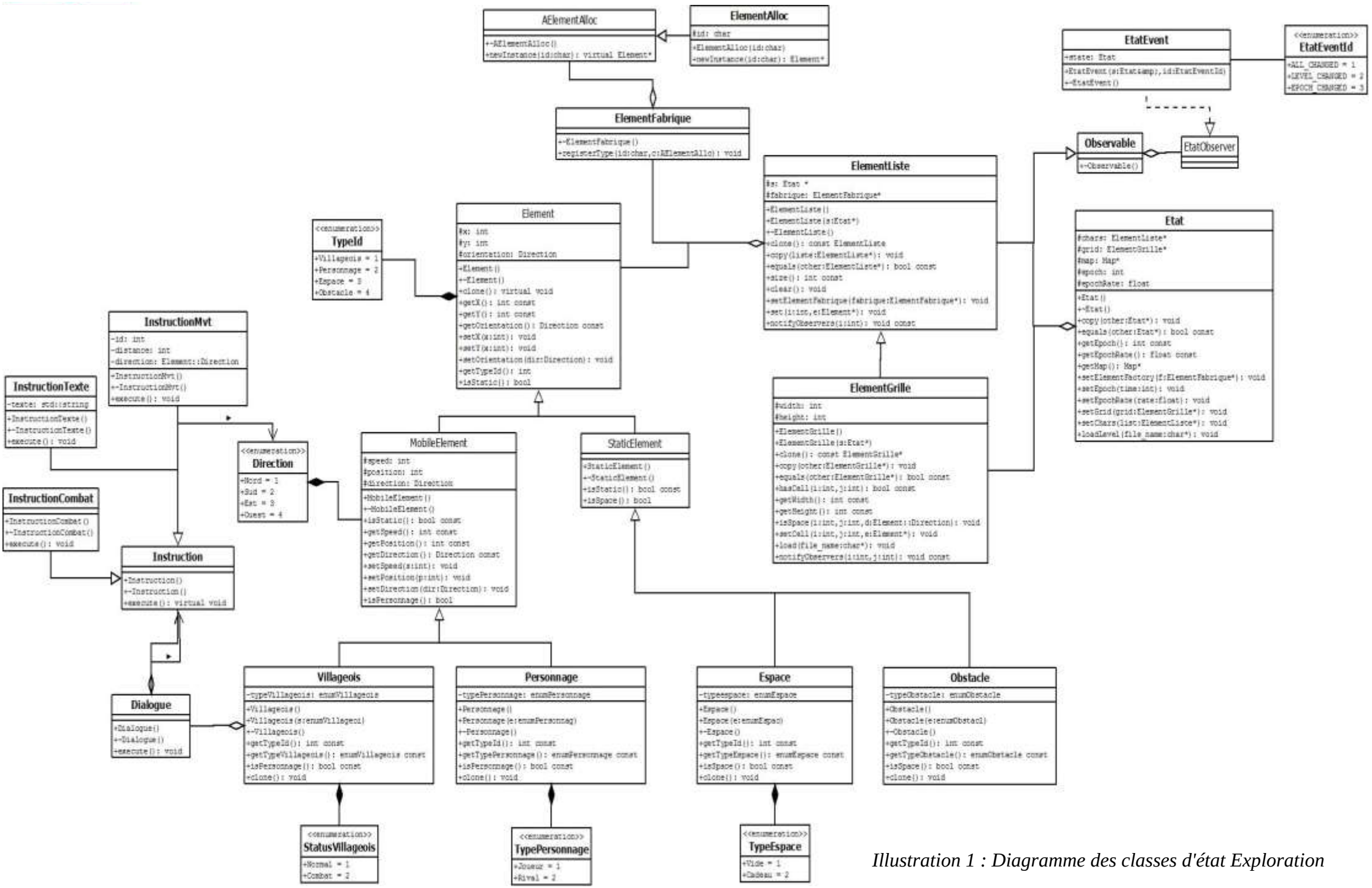


Illustration 1 : Diagramme des classes d'état Exploration

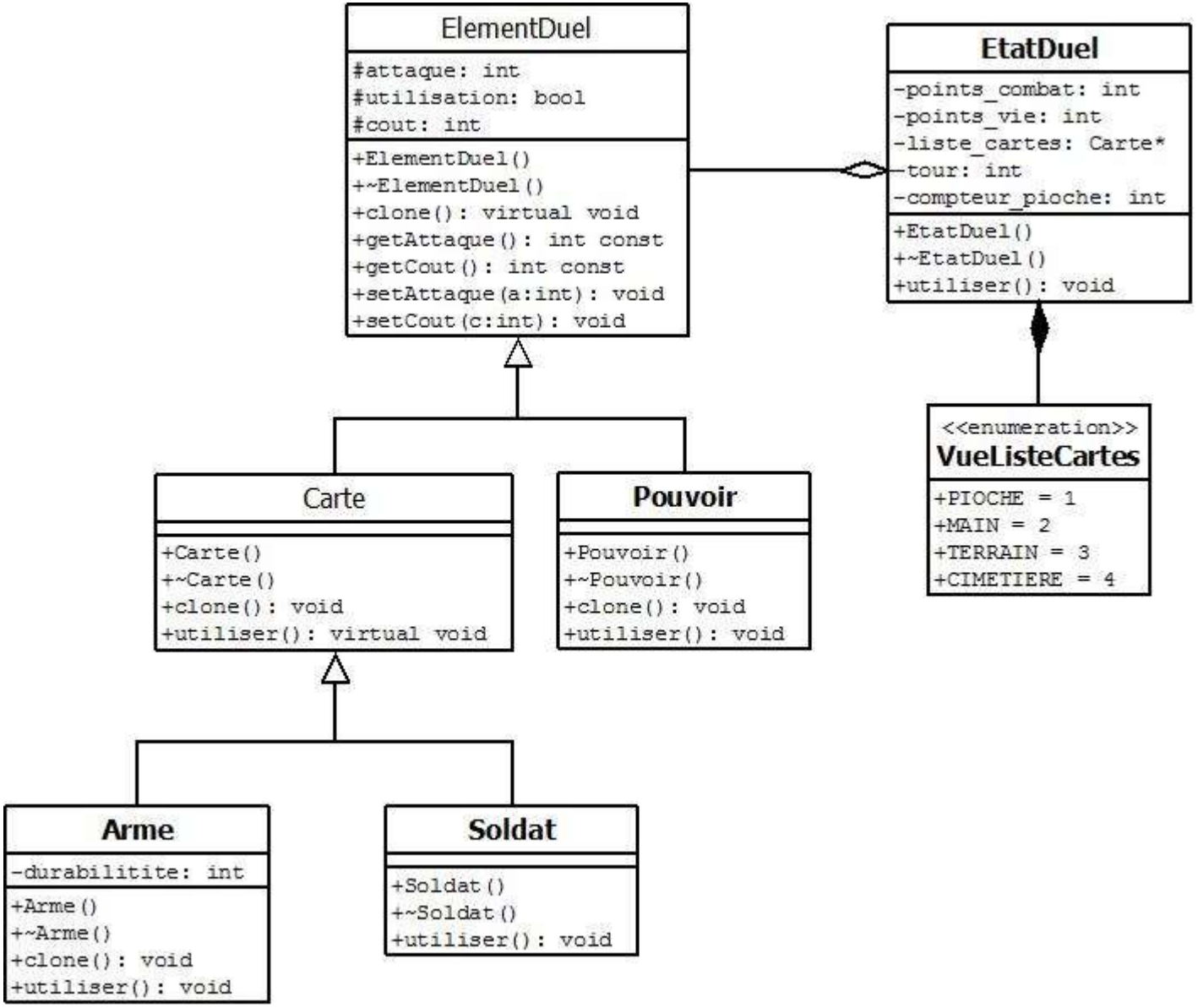


Illustration 2: Diagramme des classes d'état Duel

3 Rendu : Stratégie et Conception

Présentez ici la stratégie générale que vous comptez suivre pour rendre un état. Cela doit tenir compte des problématiques de synchronisation entre les changements d'états et la vitesse d'affichage à l'écran. Puis, lorsque vous serez rendu à la partie client/serveur, expliquez comment vous aller gérer les problèmes liés à la latence. Après cette description, présentez la conception logicielle. Pour celle-ci, il est fortement recommandé de former une première partie indépendante de toute librairie graphique, puis de présenter d'autres parties qui l'implémentent pour une librairie particulière. Enfin, toutes les classes de la première partie doivent avoir pour unique dépendance les classes d'état de la section précédente.

3.1 Stratégie de rendu d'un état

Pour le rendu d'un état, nous avons opté pour une stratégie bas-niveau. Nous découpons la scène en plusieurs plans :

Partie Exploration

- Un plan pour les éléments fixes (murs, maison, eau, etc.)
- Un plan pour les éléments mobiles (personnage, villageois, etc.)
- Un plan pour les informations (parties gagnées, princes battus, etc.)

Partie Duel

- Un plan pour le fond
- Un plan pour les éléments mobiles (cartes)
- Un plan pour les informations (vies, coûts des cartes, ATQ, DEF, etc.)

Chaque plan contiendra 2 informations bas-niveau qui seront transmises à la carte graphique : une unique texture contenant les tuiles et une unique matrice avec la position des éléments et les coordonnées dans la texture. Ainsi 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 correspondant. Sinon, comme pour les animations et/ou les éléments mobiles, nous tenons à jour une liste d'éléments visuels à mettre à jour (=modifier la matrice du plan) automatiquement à chaque rendu d'une nouvelle frame.

En ce qui concerne les aspects de synchronisation, nous avons 2 horloges (celle des changements d'états et celle de la mise à jour du rendu à l'écran). Chaque horloge pourra tenir le rythme qui lui convient avec pour seule hypothèse que l'horloge des changements d'états sera plus lente 4-12Hz) que celle de la mise à jour du rendu à l'écran (30-60Hz). En conséquence, il faut interpoler entre 2 changements d'états pour pouvoir obtenir un rendu lisse :

- *Pour les animations qui n'ont aucun lien avec l'état :* on définit une fréquence pour chaque animation. A partir de cette information, les animations tournent en boucle, sans corrélation avec les 2 horloges.
- *Pour les animations qui ont un lien avec l'état :* on produit un rendu équivalent à un sous-état fictif, produit d'une horloge fictive des changements d'états synchronisée avec celle du rendu.

3.2 Conception logiciel

Le diagramme des classes pour le rendu général, indépendante de toute librairie graphique, est présenté en Illustration 3.

- **Plans et Surfaces** : le cœur du rendu réside dans le groupe autour de la classe Plan.
  Le principal objectif des instances de Plan est de donner les informations basiques pour former les éléments bas-niveau à transmettre à la carte graphique. Ces informations sont données à une implantation de Surface. Cette implantation non représentée dans le diagramme, dépendra de la librairie graphique que nous avons choisie.

  L'ensemble classe Surface avec ses implantations suivent donc un patron de conception de type Adapter.
  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 éléments graphiques seront indexés)*.

  La classe Plan est le cas général, que l'on peut spécialiser avec des instances de StatePlan et ElementPlan, chacun étant capable de réagir à des notifications de changement d'état via le mécanisme d'Observer.

- **Scène** : Tous les plans sont regroupés au sein d'une instance de Scene.
  Les instances de cette classe seront liées (« bind ») à un état particulier. Une implantation pour une librairie graphique particulière fournira des surfaces via les méthodes setXXXSurface().

- **Tuiles** : La classe Tuile 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 StaticTuile stocke les coordonnées d'une unique tuile, et la classe AnimatedTuile stocke un ensemble de tuiles. Etant donné qu'AnimatedTuile est à la fois une classe fille de Tuile et un conteneur de celle-ci, nous sommes donc face à un patron de conception de type Composite. Enfin, les implantations de la classe TuileSet stocke l'ensemble des tuiles et animations possibles pour un plan donné. Notons que nous ne présentons pas dans le diagramme des implantations, uniquement la classe abstraite TuileSet. En outre, on peut voir ces classes comme un moyen de définir un thème graphique : lors de la création d'instance de Plan, on pourra choisir n'importe quel jeu de tuile, pourvu qu'ils contiennent des tuiles cohérentes avec le plan considéré.

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, ou sont d'une fréquence différente.
  Les animations de mouvement, par exemple Personnage qui 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.

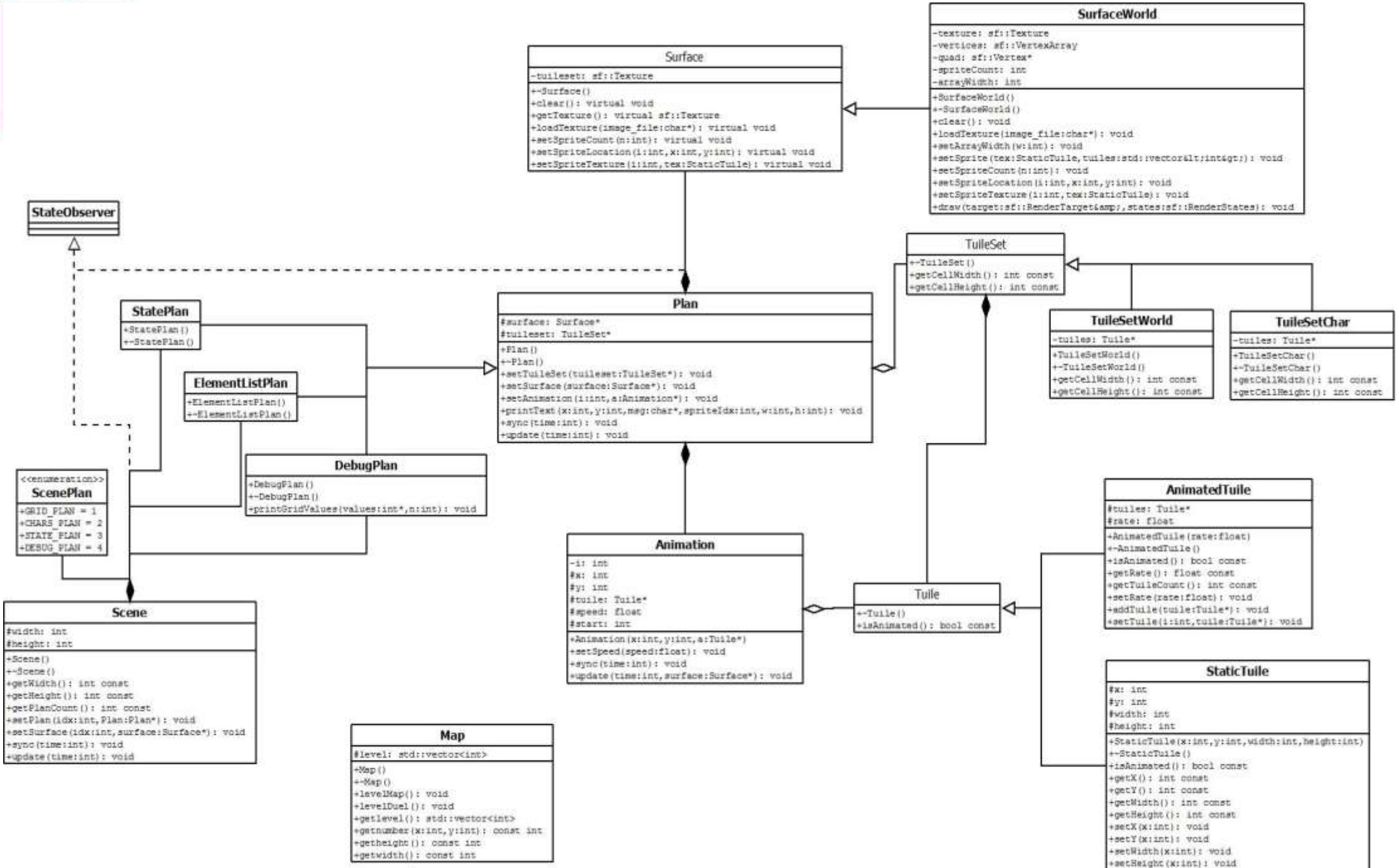


Illustration 3: Diagramme de classes pour le rendu

4 Règles de changement d'états et moteur de jeu

Dans cette section, il faut présenter les événements qui peuvent faire passer d'un état à un autre. Il faut également décrire les aspects lié au temps, comme la chronologie des événements et les aspects de synchronisation. Une fois ceci présenté, on propose une conception logicielle pour pouvoir mettre en œuvre ces règles, autrement dit le moteur de jeu.

4.1 Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en Illustration 4. L'ensemble du moteur de jeu repose sur un patron de conception de type Commande, et a pour but la mise en œuvre différée de commandes extérieures sur l'état du jeu.

Classes Commande. Le rôle de ces classes est de représenter une commande extérieure, provenant par exemple d'une touche au clavier (ou tout autre source). Notons bien que ces classes ne gèrent absolument pas l'origine des commandes, ce sont d'autres éléments en dehors du moteur de jeu qui fabriqueront les instances de ces classes. A ces classes, on a défini un type de commande avec CommandeTypeId pour identifier précisément la classe d'une instance. En outre, on a défini une catégorie de commande, dont le but est d'assurer que certaines commandes sont exclusives. Par exemple, toutes les commandes de direction pour un personnage sont exclusives : on ne peut pas demander d'aller à la fois à gauche et à droite. Pour l'assurer, toutes ces commandes ont la même catégorie, et par la suite, on ne prendra toujours qu'une seule commande par catégorie (la plus récente).

Moteur. C'est le cœur du moteur. Elle stocke les commandes dans une instance de CommandeSet. Lorsqu'une nouvelle époque démarre, ie lorsqu'on a appelé la méthode update() après un temps suffisant, le principal travail du moteur est de transmettre les commandes à une instance de Ruler. C'est cette classe qui applique les règles du jeu. Plus précisément, et en fonction des commandes ou de règles de mises à jour automatiques, elle construit une liste d'actions. Ces actions transforment l'état courant pour le faire évoluer vers l'état suivant.

Action. Le rôle de ces classes est de représenter une modification particulière d'un état du jeu. Conception logiciel : extension pour l'IA.

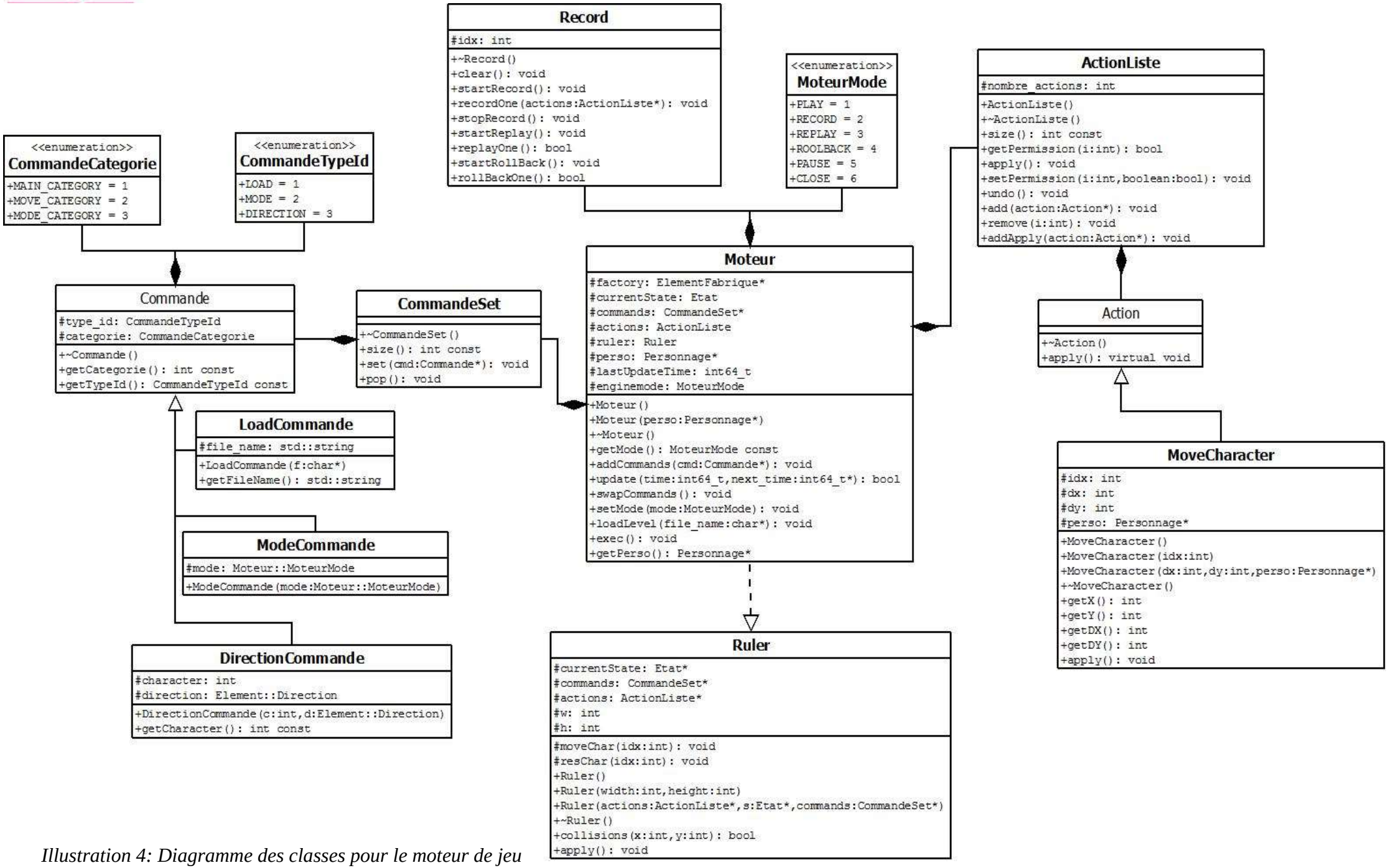


Illustration 4: Diagramme des classes pour le moteur de jeu

5 Intelligence Artificielle

Cette section est dédiée aux stratégies et outils développés pour créer un joueur artificiel. Ce robot doit utiliser les mêmes commandes qu'un joueur humain, ie utiliser les mêmes actions/ordres que ceux produit par le clavier ou la souris. Le robot ne doit pas avoir accès à plus information qu'un joueur humain. Comme pour les autres sections, commencez par présenter la stratégie, puis la conception logicielle.

5.1 Stratégies

L'ensemble de notre stratégie d'intelligence artificielle repose sur une complexification croissante de la difficulté et sur un tableau de comportements. De même il existera au sein du jeu deux types d'intelligence artificielle :

- la première est « le rival », c'est l'intelligence artificielle capable de faire les même actions que le joueur dans son ensemble. La complexité de cette intelligence artificielle déterminera la difficulté du jeu.

- la deuxième est « le duelliste », c'est l'intelligence artificielle limitée aux actions dans le mode duel, autrement dit aux parties de cartes. Sa complexité est indépendante du rival.

Ainsi chaque type d'IA pourra utiliser une liste de comportements différents. Par exemple, l'IA complexe « rival » aura un comportement de déplacement complexe et un comportement de combat complexe, tandis que l'IA complexe « duelliste » aura un comportement de combat complexe et un comportement de déplacement simple.

Cette modularité pourra permettre également de personnaliser différentes IAs.

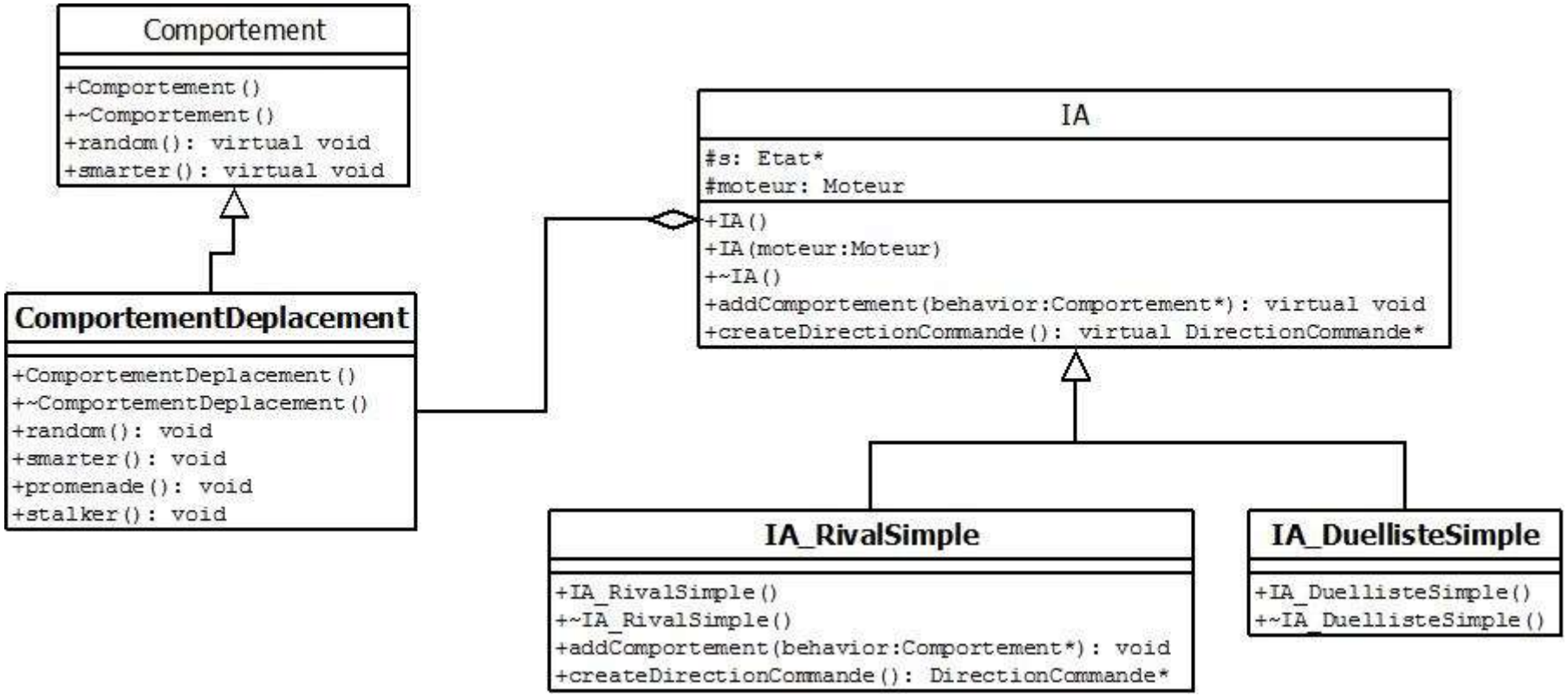


Illustration 4: Diagramme des classes pour l'IA

5.1.1 Intelligence minimale

Dans le but d'avoir une difficulté très facile et des intelligences artificielles « duelliste » accessible à tous, nous proposons une méthode dumb() de ComportementCombat très simple, basée sur les principes suivants :

- Tant que c'est possible on joue la carte la plus chère en point de combat.
- Tant que c'est possible on attaque les soldats adverses que l'on peut tuer avec nos soldats, sinon on attaque le personnage adverse.

En ce qui concerne les déplacements sur la carte d'exploration, on divise les comportements simples en deux catégories : promenade() et stalker().

L'intelligence artificielle « duelliste » utilise la méthode promenade() de ComportementDéplacement. Elle se déplace aléatoirement dans une zone et cueille des champignons.

L'intelligence artificielle « rivale » utilise la méthode stalker() de ComportementDéplacement qui agit comme suit :

- Dès que le personnage joueur avance, on cherche à se rapprocher de lui.
- Lorsqu'on atteint le joueur, on demande à lancer un combat. Ceci invite le joueur à éviter le plus possible son rival si il ne veut pas perdre de temps avec un véritable colle forte.

Dans le cas où l'intelligence « rival » est appliquée au joueur, le joueur suivra le rival qui se déplace de manière aléatoire et revient sur ses pas s’il rencontre un obstacle.

5.1.2 Intelligence basée sur des heuristiques

Nous proposons ensuite un ensemble d'heuristiques pour offrir un comportement meilleur que le hasard, et avec une chance notable de résoudre le problème complet (ie battre le joueur en duel ou finir le jeu avant lui dans le cas de l'IA « rival »).

Dans le cas de ComportementDeplacement, l'IA « rival » peut utiliser la méthode smarter() :

- Si le rival est proche du joueur, on s'en rapproche le plus possible et on tente de lancer un combat.
- Sinon on se dirige vers le « duelliste » non battu le plus proche et on lance un combat.

Ces heuristiques proposées sont mises en oeuvre en utilisant des cartes de distance vers un ou plusieurs objectifs. Pour calculer ces cartes, nous utilisons l'algorithme de Dijkstra.

Dans le cas de ComportementCombat, les IA « rival » et « duelliste » peuvent utiliser la méthode smarter() (par soucis pratique on nomme duelliste l'intelligence artificielle en combat) :

- Si le duelliste peut achever le joueur avec ses cartes sur le terrain et dans sa main, il les utilise toutes pour vaincre le joueur.
- Si le duelliste risque de se prendre des dégâts à ce tour, il joue si possible une carte de provocation. Sinon il tente de tuer les soldats sur le terrain du joueur avec ses cartes sur le terrain d'abord, puis ses cartes dans sa main.
- Si le duelliste peut jouer une carte, il joue sa carte la plus chère en point de combat tant qu'il en dispose.
- Si le duelliste peut attaquer les soldats adverses avec ses soldats il le fait en perdant le moins de soldats possible, sinon il attaque le personnage adverse.

Ces heuristiques proposées sont mises en œuvre grâce à un algorithme alpha beta.

5.2 Conception logiciel

Classes IA. Toutes les formes d'intelligence artificielle implantent 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 une instance par personnage, ce qui permet de « personnaliser » chaque personnage. Les classes IA_***Simple implantent l'intelligence minimale, telle que présentée ci-dessus. De même, les classes IA_***Moyenne implantent la version améliorée.

Classes Comportement. Toutes les formes d'intelligence artificielle contiennent un tableau de comportements qui vont dicter les commandes à envoyer. Toutes les classes de comportement héritent d'une classe abstraite Comportement. Ces classes de comportements contiennent des méthodes qui définissent le « comportement » de l'intelligence artificielle associée

Chemin. La classe Chemin permet de calculer une carte des distances à un ou plusieurs objectifs. Plus précisément, pour chaque case « espace » du niveau, on peut demander un poids qui représente la distance à ces objectifs. Pour s'approcher d'un objectif lorsqu'on est sur une case, il suffit de choisir la case adjacente qui a un plus petit poids. Même si cela n'est pas optimal, on peut également utiliser ces poids pour s'éloigner des objectifs, en choisissant une case avec un poids supérieur. L'interface CibleChemin nous permet de fabriquer des cartes de distances pour de nombreux types d'objectifs (patron de type Strategy). Nous avons implanté un certain nombre, qui ne sont pas présentés sur le diagramme :

- CibleDuelliste : Objectif duelliste
- CibleCadeau : Objectif cadeau

6 Modularisation

Cette section se concentre sur la répartition des différents modules du jeu dans différents processus. Deux niveaux doivent être considérés. Le premier est la répartition des modules sur différents threads. Notons bien que ce qui est attendu est une parallélisation maximale des traitements: il faut bien démontrer que l'intersection des processus communs ou bloquant est minimale. Le deuxième niveau est la répartition des modules sur différentes machines, via une interface réseau. Dans tous les cas, motivez vos choix, et indiquez également les latences qui en résulte.

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

Notre objectif ici est de placer le moteur de jeu sur un thread et le moteur de rendu sur un autre thread. Nous avons deux types d'information qui peuvent transiter d'un module à l'autre : les commandes et les notifications de rendu.
Commandes. Il s'agit ici des touches du clavier pressées par l'utilisateur. Celles-ci peuvent arriver à n'importe quel moment, y compris lorsque l'état du jeu est mis à jour. Pour résoudre ce problème, actuellement traitées par une mise à jour de l'état du jeu, l'autre accueillera les nouvelles commandes. A chaque nouvelle mise à jour de l'état du jeu, on permute les deux buffers : celui qui contenait les commandes traités devient celui accueillant les nouvelles commandes, et l'autre devient disponible pour accueillir les futures commandes. Ainsi, il existe toujours un buffer capable de recevoir des commandes, et cela sans aucun blocage. Il en résulte une parfaite répartition des traitements, ainsi qu'une latence au plus égale au temps entre deux époques de jeu.
Notifications de rendu. Ce cas est problématique, car il n'est pas raisonnable d'adopter la même approche. En effet, cela implique un doublement de l'état du jeu (un en cours de rendu, l'autre en cours de mise à jour), ce qui augmente de manière significative l'utilisation mémoire et processeurs, ainsi que la latence du jeu. Nous nous sommes donc tournés 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 buffer 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 à la carte graphique, 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.

6.1.2 Répartition sur différentes machines

- **Format des messages**

JSON (JavaScript Object Notation) est un format léger d'échange de données. Il est facile à lire/écrire et est facilement analysable par des machines. JSON est un format texte complètement indépendant de tout langage. Ces propriétés font de JSON un langage d'échange de données idéal.

Un document JSON ne comprend que deux éléments structurels :

- des ensembles de paires nom / valeur
- des listes ordonnées de valeurs

Ces mêmes éléments représentent 3 types de données :

- des objets
- des tableaux
- des valeurs génériques de type tableau, objet, booléen, nombre, chaîne ou NULL

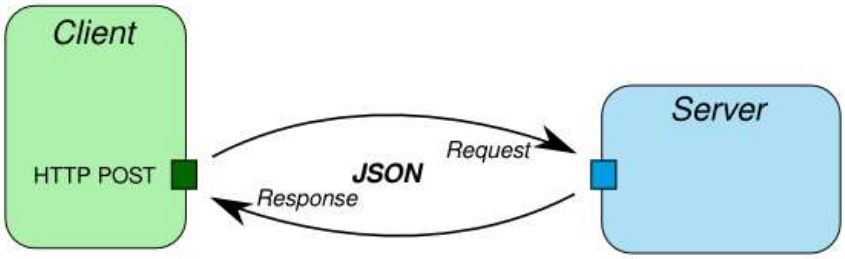


Illustration 5: Requête/Réponse Json

Avantages :
Le principal avantage de l'utilisation de JSON, dans notre application, est qu'il est simple à mettre en œuvre. Mais également,

- ✓ Facile à apprendre, car sa syntaxe est réduite et non-extensible
- ✓ Ses types de données sont connus et simples à décrire
- ✓ Peu verbeux et léger, au contraire du langage XML qui est très verbeux

• Web Service

Un Web Service est un programme informatique permettant la communication et l'échange de données entre applications et systèmes hétérogènes dans des environnements distribués. Il s'agit donc d'un ensemble de fonctionnalités exposées sur internet ou sur un intranet, par et pour des applications ou machines, sans intervention humaine.

Il existe deux grandes familles de Web Services :

- Les services web de type SOAP
- Les services web de type REST

1) SOAP

SOAP (Simple Object Access Protocol) est un protocole RPC orienté objet bâti sur XML ce qui le rend indépendant de tout système d'exploitation et de tout langage de programmation, il permet de faire des appels de procédures entre objets distants physiquement situés sur un autre serveur.

SOAP est défini pour être indépendant du protocole de transport utilisé pour véhiculer le message. Cependant, le protocole le plus utilisé avec SOAP est HTTP car c'est devenu un standard de fait. Son utilisation avec SOAP permet de rendre les services web interopérables.

Le protocole SOAP est composé de 2 parties :

- une enveloppe, contenant des informations sur le message lui-même afin de permettre son acheminement et son traitement
- un modèle de données, définissant le format du message, c'est-à-dire les informations à transmettre

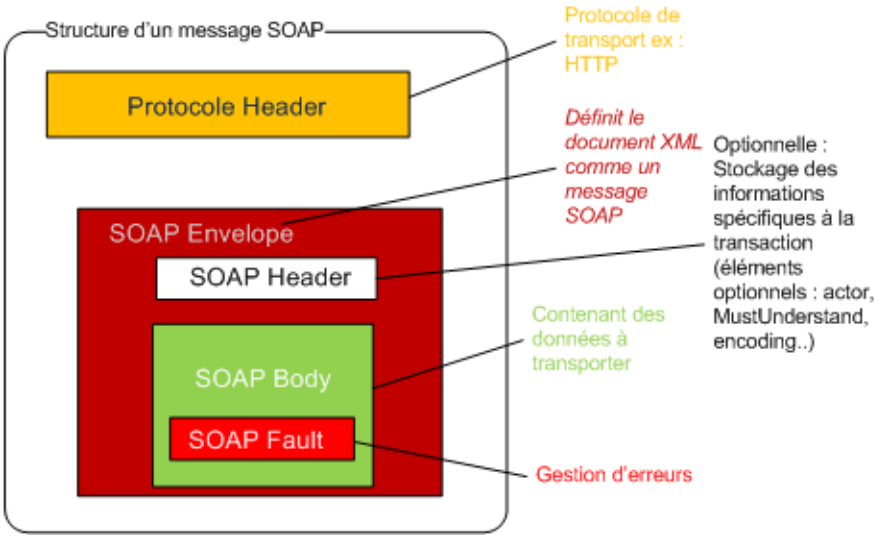


Illustration 6: Structure d'une enveloppe (source Openclassroom)

2) REST

REST (Representational State Transfer) est une manière de concevoir des applications ou des Web Services. Il n'est pas un protocole ou un format, c'est un style d'architecture sans état (stateless).

Bien que REST ne soit pas un standard, il utilise des standards. En particulier,

- l'URI: connaître l'URI doit suffire pour nommer et identifier une ressource
- HTTP: fournit toutes les opérations nécessaires (GET, POST, PUT et DELETE)

Avantages :
L'absence de gestion d'état du client sur le serveur conduit à une consommation de mémoire inférieure, une plus grande simplicité et donc à une capacité plus grande de répondre à un grand nombre de requêtes simultanées.

Notre choix : REST
En effet, SOAP redéfinit un protocole via une enveloppe, des entêtes et un document, à l'intérieur du protocole réseau l'hébergeant (la plupart du temps HTTP). SOAP ne bénéficie donc pas des requêtes HTTP, gérée par l'infrastructure réseau, et est donc plus lent. Ainsi REST est plus rapide, ce dont nous avons besoin pour un jeu : exécution des commandes.

6.2 Conception logiciel

Nous présentons en *Illustration7* le diagramme de classes pour la modularisation du projet. Deux packages sont présentés : le package *server* regroupe les opérations relatives au moteur de jeu, et le package *client* les opérations relative au moteur de rendu.

Classes Server. La classe abstraite Server encapsule un moteur de jeu, et lance un thread en arrière-plan pour effectuer les mises à jour 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'opérations communes à tous les clients graphiques ont été placés dans cette classe, afin de rendre la création d'un nouveau type de client plus rapide.

L'*Illustration8* présente le diagramme des classes utilisées. On peut distinguer plusieurs parties :

— Classes Commande (déjà créée) et CommandeDB : permettent de simuler une petite base (ie, dans un cas plus réel, ce serait l'interface vers une base SQL ou noSQL). Les différentes méthodes parlent d'elles-mêmes, et correspondent aux quatre opérations CRUD usuelles.
— Classe AbstractService : classe abstraite pour tout service REST de notre jeu
— Classe CommandeServices : implémente les 4 opérations CRUD (GET/POST./PUT/DELETE)

- GET

Le but de cette opération est de renvoyer une commande à partir de son identifiant. Par exemple, si on place l'url suivante dans un navigateur : http://localhost:8080/commande/1

On souhaite voir apparaître le résultat suivant :
{
« name » = moveup ;
« id » = 1 ;
}

Pour ce faire, on décode l'identifiant dans l'url, de format <pattern service>/<identifiant>. Par exemple, pour le service Commande, le format est /commande/<identifiant>.
La méthode CommandeService::get() pour qu'elle peuple l'objet JSON en argument avec les valeurs de la commande trouvée.

- POST

Le but de cette opération est de modifier les informations d'une commande existante. Par exemple si on exécute la commande curl suivante dans un terminal :
curl -X POST --data '{"id":3}' http://localhost:8080/commande/3

Cela va modifier l'id de la commande d'identifiant 3. Une opération POST ne renvoie pas de données (uniquement le status HTTP).

La méthode CommandeService::post() modifie la commande désignée. On récupère une copie de cette commande, on modifie les attributs indiqués, puis on remplace la commande existante avec cette copie.

- PUT

Le but de cette opération est d'ajouter une commande existante. Par exemple si on exécute la commande curl suivante dans un terminal :
curl -X PUT --data '{"commande":"movedown","id":4}' http://localhost:8080/commande

Cela va créer une nouvelle commande avec les valeurs indiquées, puis renvoyer son identifiant :
{
"id" : 4
}

La méthode CommandeService::put() pour qu'elle ajoute une commande, puis peuple l'objet JSON retour avec son identifiant.

- DELETE

Son but est uniquement de supprimer la commande désignée par l'identifiant dans l'URL.

La méthode CommandeService::remove() supprime la commande indiquée.

— Classe ServicesManager : le gestionnaire de service, ie qui sélectionne le bon service et la bonne opération à exécuter en fonction de l'url et de la méthode HTTP.
— Classe ServiceException : permet de jeter une exception à tout moment pour interrompre l'exécution du service, de la manière suivante :

```
throw new ServiceException(<code status HTTP>,<message>);
```

Le code status HTTP est une des valeurs de l'enum HttpStatus, et le message une chaîne de caractères. Par exemple :

```
throw ServiceException(HttpStatus::NOT_FOUND,"Invalid_user_id");
```

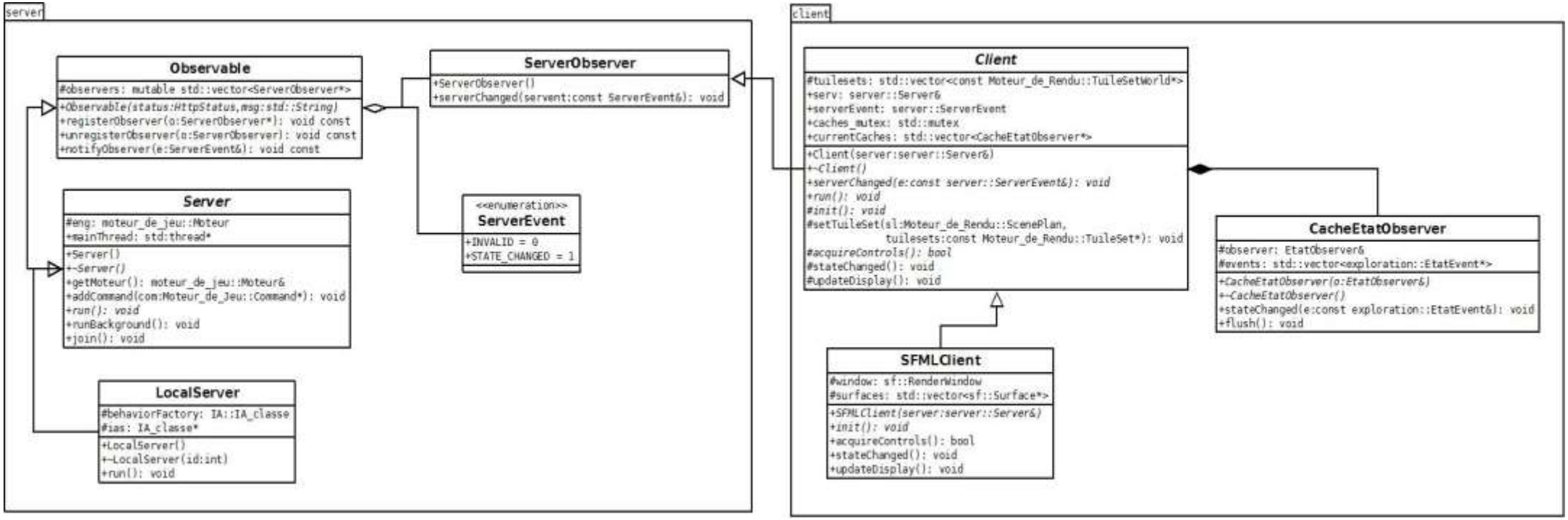


Illustration 7: Diagramme de classes pour la modularisation

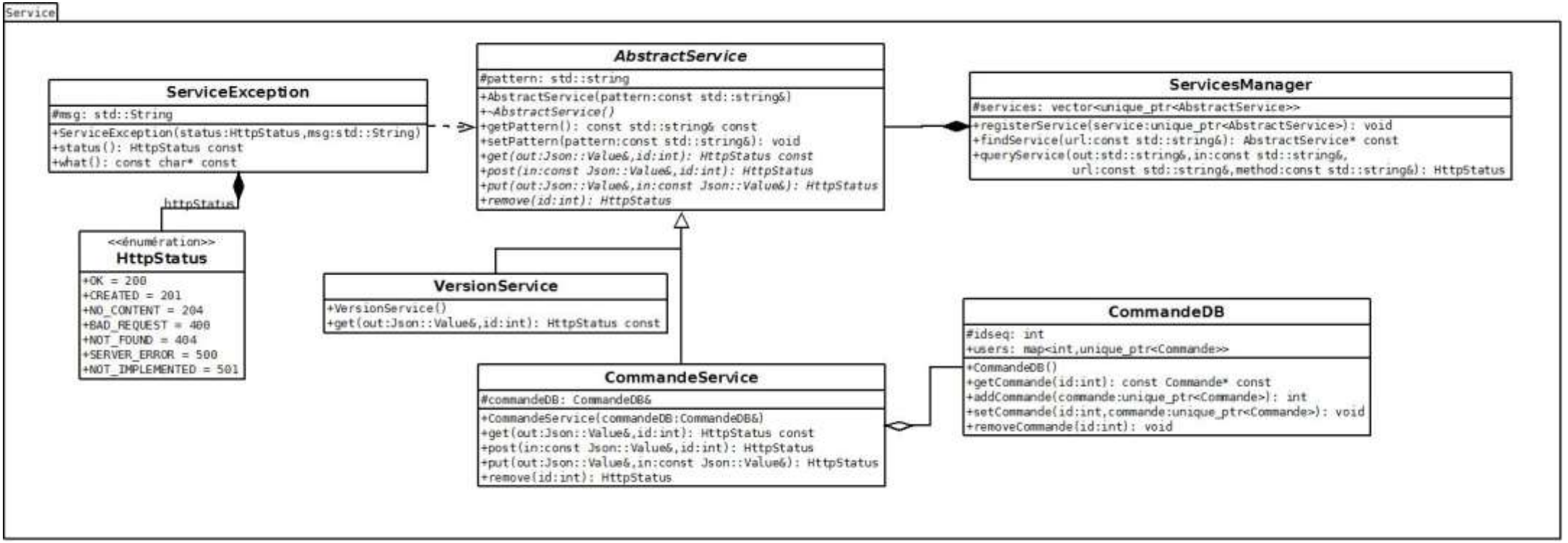


Illustration 8: Diagramme de classes pour la modularisation