

Projet jeu Darkest Dungeon

MONJOUX Hugo
RIVIÈRE Hadrien
Option IS TP1

18 Septembre 2018



FIGURE 1 – Aperçu du jeu "The Darkest Dungeon"

Table des matières

| | |
|---|-----------|
| 1 Présentation Générale | 3 |
| 1.1 Archéotype | 3 |
| 1.2 Règles du jeu | 3 |
| 1.3 Ressources | 3 |
| 2 Description et conception des états | 6 |
| 2.1 Description des états | 6 |
| 2.1.1 Etat éléments fixes | 6 |
| 2.1.2 Etat éléments mobiles | 6 |
| 2.1.3 Etats généraux | 7 |
| 2.2 Conception de logiciel | 7 |
| 3 Rendu : Stratégie et conception | 9 |
| 3.1 Stratégie de rendu d'un état | 9 |
| 3.2 Conception de logiciel | 9 |
| 4 Règles de changements d'états et moteur de jeu | 11 |
| 4.1 Changements extérieurs | 11 |
| 4.2 Changements autonomes | 11 |
| 4.3 Conception de logiciel | 11 |
| 5 Intelligence Artificielle | 13 |
| 5.1 Stratégies | 13 |
| 5.1.1 Intelligence aléatoire | 13 |
| 5.1.2 Intelligence basée sur des heuristiques | 13 |
| 5.1.3 Intelligence artificielle basé sur un algorithme d'optimisation de type MIN/MAX | 13 |
| 5.2 Conception de logiciel | 14 |
| 6 Modularisation | 16 |
| 6.1 Organisation des modules | 16 |
| 6.1.1 Implantation du multi-threading | 16 |
| 6.1.2 Répartition sur différentes machines : rassemblement des joueurs | 16 |
| 6.1.3 Répartition sur différentes machines : échange des commandes | 18 |
| 6.2 Conception de logiciel | 18 |

1 Présentation Générale

1.1 Archétype

Notre jeu s'inspire de Darkest Dungeon, qui est un RPG tour par tour, Rogue-like. Ce jeu est illustré dans les différentes figures exposées en fin du rapport.

1.2 Règles du jeu

Ce dernier permet au joueur d'évoluer soit en tant que héros, soit en tant qu'ennemis. L'objectif du joueur varie donc selon ce choix fait en début de partie. Si le joueur choisit de jouer en tant que héros, il doit arriver à la fin du niveau. Dans l'autre cas le joueur doit s'assurer que les héros n'arrive pas à la fin du niveau.

Le jeu se compose en trois vues :

- Vue d'accueil du joueur, qui pourra définir son équipe de héros, ainsi que l'inventaire de l'équipe, soit la répartition des ennemis sur les cartes de jeu
- Vue de déplacement dans le donjon
- Vue de combat lors de la confrontation des héros avec les ennemis

1.3 Ressources



FIGURE 2 – Premier élément de décor : couloir



FIGURE 3 – Deuxième élément de décor : salle



FIGURE 4 – Ecran d'accueil avec à l'intérieur du cadre, le menu d'accueil (New game, Options...)



FIGURE 5 – Exemple de design d'un personnage se transformant. Seulement les sprites encadrés seront utilisés pour alléger les ressources.

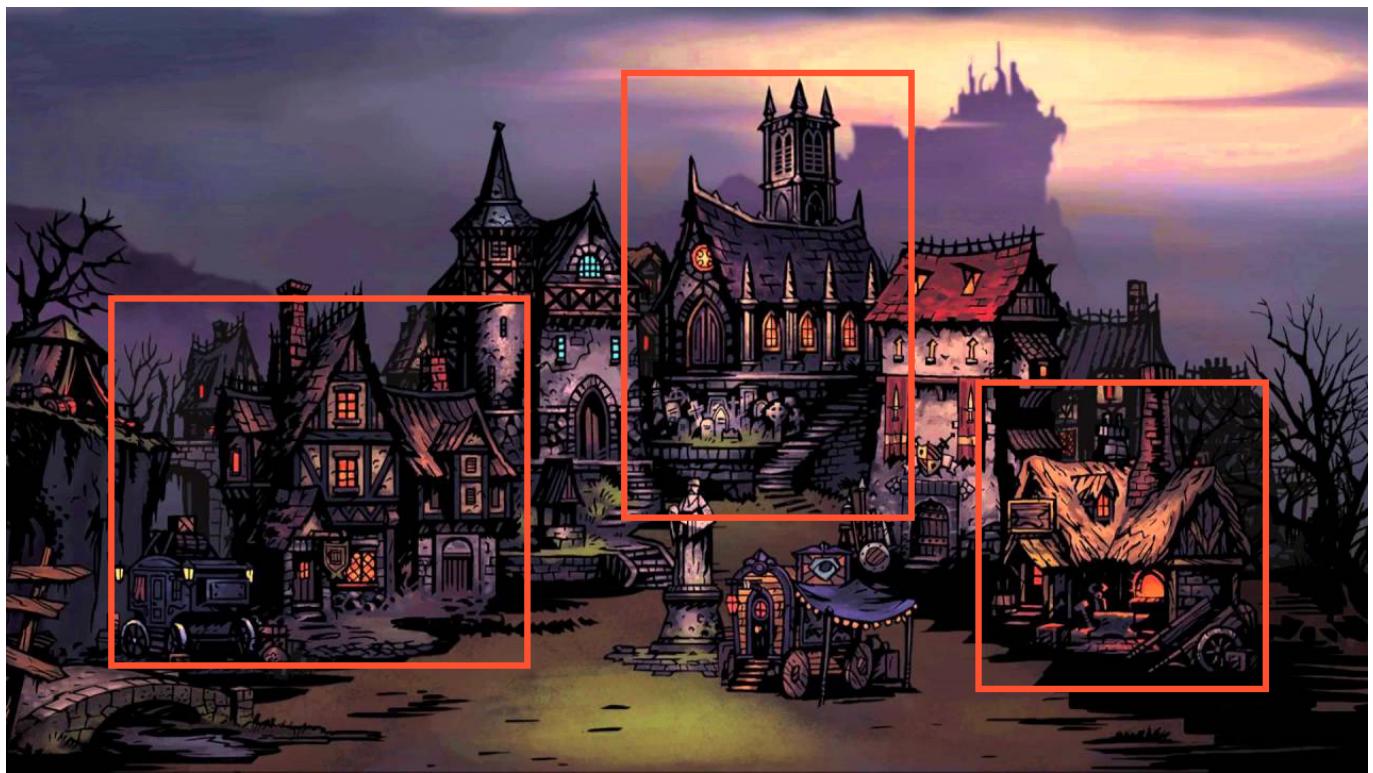


FIGURE 6 – Ville d'accueil du joueur avec de gauche à droite encadrés : Auberge de recrutement pour modifier l'équipe, Sauvegarde de la progression, et Magasin pour remplir l'inventaire de l'équipe.



FIGURE 7 – Ecran de jeu principal : encadrés en gris clair, les héros à gauche et les ennemis à droite avec leur barre de vie respectives situées en dessous | encadrés en orange, les différents outils à disposition du joueur (compétences, statistiques des personnages, équipements, carte, inventaire). Ces éléments seront adaptés aux contraintes de temps et de moyens de notre projet.

2 Description et conception des états

2.1 Description des états

La partie, une fois le jeu lancée le joueur est directement envoyé sur l'écran du village dans l'état actuel d'avancement. Ce dernier peut alors interagir avec les éléments fixes du Village évoqué ci-dessous

Puis une fois dans la partie ingame, l'état de jeu est formé par les éléments fixes (cartes, éléments de gameplay, ressources graphiques...) et des éléments mobiles (héros, ennemis, barres de vie, ordering lors des combats...). Les éléments mobiles dits mutables (héros et ennemis) sont transportés dans les éléments fixes, le donjon en cours d'exploration et les salles.

2.1.1 Etat éléments fixes

Les différentes cartes, à l'instar de celles trouvables en FIGURE 6 et en bas à droite de la FIGURE 7, sont des cartes statiques présentant des ressources graphiques majoritairement statiques. L'ensemble de ces éléments ne seront interactifs que par des effets de surbrillance ou d'ombre.

La première carte illustrée en FIGURE 6 caractérisant la classe Village sert d'accueil au joueur souhaitant jouer côté Héros, et présentent un carrefour entre l'Auberge (constitution de l'équipe), l'Eglise (Sauvegarde), le Magasin (acheter des items pour l'inventaire de l'équipe), et le Carrosse (partir à la conquête des donjons). Les classes constituant les éléments fixes sont :

- Le **Village** : qui donne accès à la fois au **Carrosse** ainsi qu'à l'**Eglise**.
- L'**Auberge** : qui donne accès à la customisation de l'équipe. Deux listes sont alors modifiées : une liste de personnage appartenant au joueur et l'autre représentant la banque des personnages disponibles.
- Le **Shop** : qui donne accès à un autre écran sur lequel le joueur peut acheter des items (Potions, Potions de vie et Antidote).
- Les **Item** : un listing des items obtenables par le **Shop** afin d'équiper le joueur au sein du **Donjon**, caractérisés par des méthodes singulières selon la nature de l'item. Ces derniers sont alors entreposés dans l'Inventaire du joueur qui pourra utiliser lors des combats. Ces derniers sous des classes filles de la présente classe : **PotionVie**, **PotionSoin** et **Antidote**.
- Le **Dongeon** : un listing de toutes les salles, ainsi que leur composition, ainsi que la position du joueur.
- Les **Room** : des instances qui réunissent à la fois la team ainsi que les différents ennemis, l'ordre des tours à venir. Dans la classe, un calcul de passage pour les membres présents dans la salle est fait à chaque initialisation.

La deuxième carte illustrée en FIGURE 7 en bas à droite permet au joueur de se déplacer dans le donjon dans lequel il progresse. Cette carte intègre une unique ressource graphique qui est faite à la main, représentant le plan du donjon. Les salles, indiquées par des carrés, seront associées à des phénomènes de surbrillance par exemple si le joueur s'y déplace.

2.1.2 Etat éléments mobiles

Les éléments mobiles possèdent une position de départ qui correspond à leur position sur l'écran de combat avant que l'attaque soit effectuée. Ils ont ensuite une vitesse de déplacement qui correspond à la vitesse à laquelle le sprite se déplace sur l'écran.

- **Character** (héros ou monstres) : Ces éléments sont contrôlés soit par un joueur soit par une IA. Ces derniers ont accès à une liste donnée de Skills auxquels ils ont accès lors du combat.
- Tout d'abord les **Monsters**, classe fille de **Character** qui elle-même à plusieurs classes filles : **Dragon**, **Blob**, **Sorcerer** et **DarkKnight**.
- Puis, les **Hero**, elle aussi classe fille de **Character** qui elle-même à plusieurs classes filles : **Range**, **Mage**, **Tank** et **Assassin**.

- Chaque personnage est alors caractérisé par ses statistiques définies statiquement, ainsi qu'une liste de skill accessibles et enfin un identifiant booléen pour savoir si le **Character** est un **Monsters** ou un **Hero**.

2.1.3 Etats généraux

L'ensemble de ces états sont complétés par un état général qui résume l'état du jeu comme défini ci-dessous à un instant t :

- Dans la classe **State**, une horloge permettant de rythmer les tours de jeu en ingame, ainsi que l'apparition et la désapparition des éléments in et outgame.
- Le Gameplay du Game + est basé sur l'édition du nombre de donjons finis, et du nombre de donjons finis successivement. Ainsi des conditionnelles sur ces chiffres peuvent être implémentées pour maximiser la durée de vie du jeu, ou simplement permettre de suivre le scoring du joueur sur sa partie.
- L'Etat de la **Team** : la santé in-dungeon, l'état du personnage (en forme, en état de choc, empoisonné, mort...)
- La classe **ElementTab** : la liste des salles, leur occupation (nombre de monstres, quels monstres), leur état (finies, non finies, où le héros se situe).
- L'**Inventory** : il s'agit de lister l'ensemble des item qui à un moment t appartiennent au joueur.
- Les **Skills** qui sont utilisés par les **Character** lors du combat. Il est à noter que la sauvegarde, comme évoqué plus haut en section 2.1.1, conserve les deux premiers tirets de ces états généraux.

2.2 Conception de logiciel

Le diagramme de classe est présenté ci-dessous en FIGURE 8 et nous pouvons y observer plusieurs groupes de classes :

- **Classes Elements** : ces classes sont représentées en jaune sur le diagramme pour les classes intermédiaires, les classes objets finales (qui n'ont pas de filles) sont représentées en bleue, et caractérisent des archétypes (d'item, de personnages ou d'ennemis). Afin d'identifier les natures des classes des méthodes sont définies telles que, notamment si l'objet créé est un élément mobile ou statique, par des méthodes `isStatic()`, ou au sein de l'architecture des classes statiques, pour savoir si l'objet créé est un bâtiment avec une méthode `isBuilding()`.
- **Conteneurs d'élément** : en vert, les classes `State`, `ElementTab`, `Inventaire` et `Team` qui permettent d'accumuler les éléments utiles à la partie du joueur. Les trois classes `Inventaire`, `Team` et `ElementTab` sont des conteneurs d'objets qui sont à dimensions finies (des tableaux de dimensions N éléments), et la classe `State` qui nous permet d'accéder à toutes les données de l'état du jeu à un instant donné.

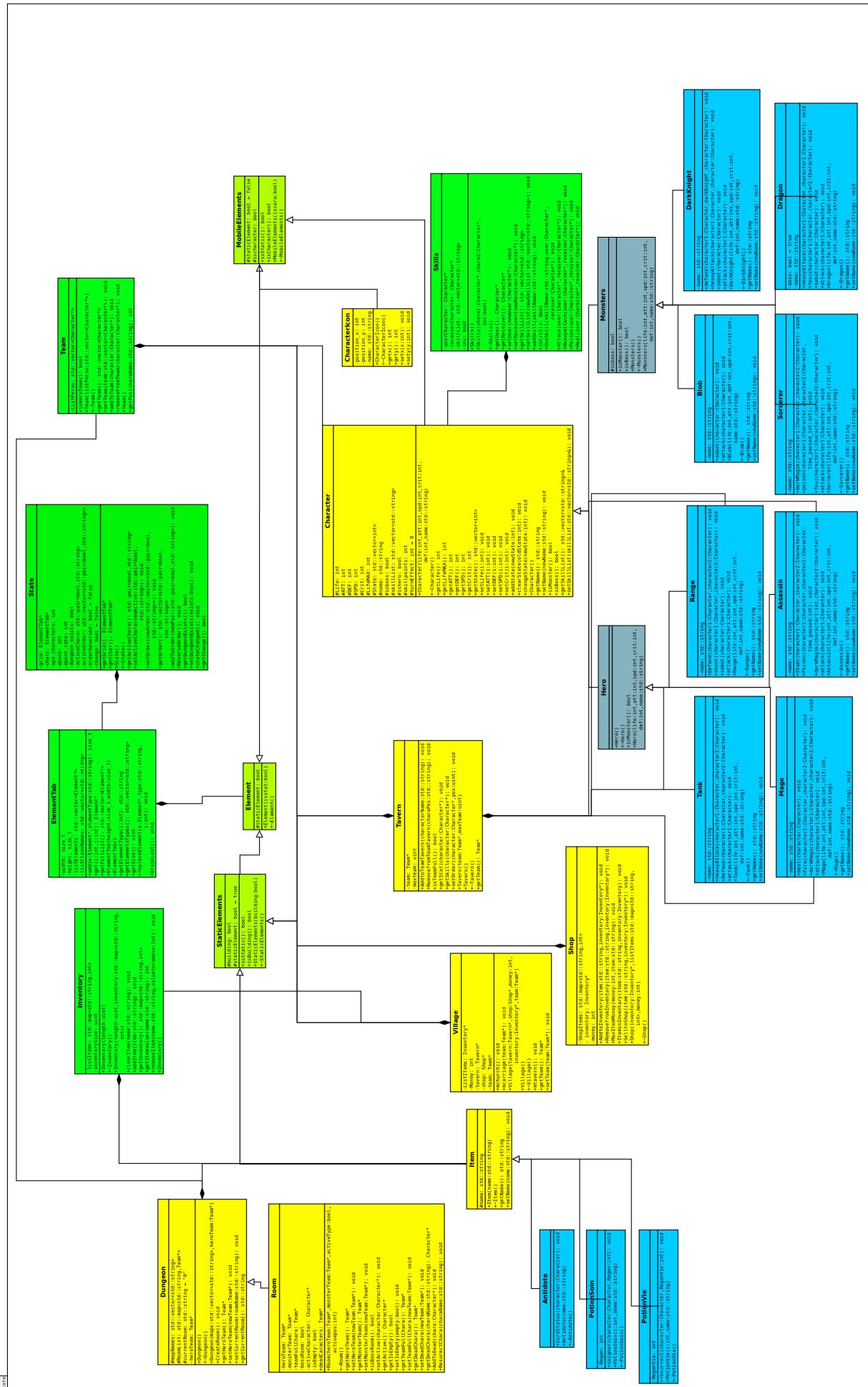


FIGURE 8 – Diagramme de classe associé au projet fait via DIA

3 Rendu : Stratégie et conception

3.1 Stratégie de rendu d'un état

La stratégie d'affichage du rendu se dirige vers un fonctionnement organisé en layers (calques) : ces derniers représentent chacun un plan.

Chacun des plans est lui-même constitué des éléments provenants du diagramme state.dia affiché ci-dessus. Afin d'ordonner ces plans de manière optimale, il est nécessaire de les organiser de la sorte :

1. Backgrounds : l'ensemble des éléments de décors intervenant sur l'arrière de l'écran (texture des salles, maps, ville, écran d'accueil...)
2. Mobiles objects : l'ensemble des objets qui sont assimilés à une interactivité avec le joueur ou l'horloge interne du jeu (sprites en mouvement, bouton cliquable...)
3. Effets : cela inclut tous les éléments graphiques ayant une signification à la fois dans l'aide de l'affichage, ou d'effet temporaires régis par une horloge (sprite attaques, halos lumineux ...)

Chacun de ses plans contient à la fois les éléments placés sur une matrice (de dimensions le nombre de pixels en largeur et hauteur, ce qui sera probablement 2130x1440px) qui modélise une grille placée sur notre fenêtre d'affichage, ainsi qu'une texture à afficher selon le plan concerné.

Les différentes couches de layers mettent notamment en évidence des changements d'états à la fois permanents, cycliques et temporaires. La plupart de ceux-ci sont donc gérés différemment mais temporiser par l'horloge interne définie.

- Si le changement d'état donne lieu à l'édition du rendu de manière permanente : la matrice de plan est éditée notamment les couches basses (backgrounds)
- Si le changement d'état donne lieu à l'édition du rendu de manière temporaire : la matrice de plan est éditée et régulièrement mise à jour notamment les couches hautes (Mobiles objects et Effets)

Concernant la synchronisation, il faut distinguer deux fréquences d'édition :

1. Une horloge étant associée à l'édition des états des éléments du jeu : celles-ci doit être relativement lente (de l'ordre de 0.1 s afin d'éditer plusieurs états par frame. Cependant elle doit être plus lente que l'horloge d'édition du rendu purement graphique.
2. Une horloge associée au rendu graphique (mouvement des sprites, déplacement des personnages, déplacements des animations...) : celle-ci à l'instar de beaucoup de jeux-vidéo actuels, doit à minima tenir une cadence de 30Hz voir 60Hz si le rendu n'est pas trop lourd.

Concernant la gestion des rendus associés aux différentes horloges :

- Certaines animations cycliques (mouvement des personnes, des ennemis, animations indiquant l'empoisonnement des personnages, halos lumineux sur des cases) disposeront de fréquences uniques leur étant associées, afin d'optimiser le rendu graphique.
- Lorsque le personnage joue il est augmenté en taille afin de le distinguer des autres personnages.

3.2 Conception de logiciel

Le diagramme de classe concernant le rendu global est présenté ci-dessous en FIGURE 9. Notre projet inclut des affichages statiques non composés de tuiles reproductibles, ce qui nous permet d'imaginer un diagramme de classe relativement simple, bien que concevoir de sorte à satisfaire plusieurs états d'affichages. Ce diagramme de classe est composé de :

- **Layers** : la classe Layers instancie les différents plans constituant le rendu global. Cela constitue l'ensemble des informations envoyées à la carte graphique. Cette classe communique avec les Surfaces d'un côté, et les définitions de layers, associés soit à l'état de jeu, soit aux éléments constituants les états à un instant précis.
- Une fois la texture chargée, il faut donc structurer l'affichage à l'écran, qui est géré par la méthode de setSpriteLocation(), et setSpriteTexture(list), qui dans notre cas nous permet de gérer l'affichage structurer de chaque état.
- **Surface** : Chaque surface est constituée d'une texture qui regroupe à elle-seule plusieurs sprites. Ces derniers représentent des types de ressources graphiques différentes (monstres, backgrounds, icônes, effets...) et sont concaténés dans cette texture. Chaque élément est placé selon le type d'état qui est défini dans l'état général, et les coordonnées de départ (en haut à gauche) et d'arrivée (en bas à droite) de chaque sprite.
- **XDisplay** : ces classes permettent de caractériser un display particulier associé à chacun des état du jeu. Nous en définissons ici 4 pour s'adapter à la majorité des états pour l'instant imaginés. Il est à noter que l'identité de l'état est récupéré par l'intermédiaire de state et nous permet d'identifier l'état à display. Par exemple pour l'état de combat défini par l'état Room, l'affichage et constant et demande N sprites avec des positions pré-définies dans les sous-classes évoquées.

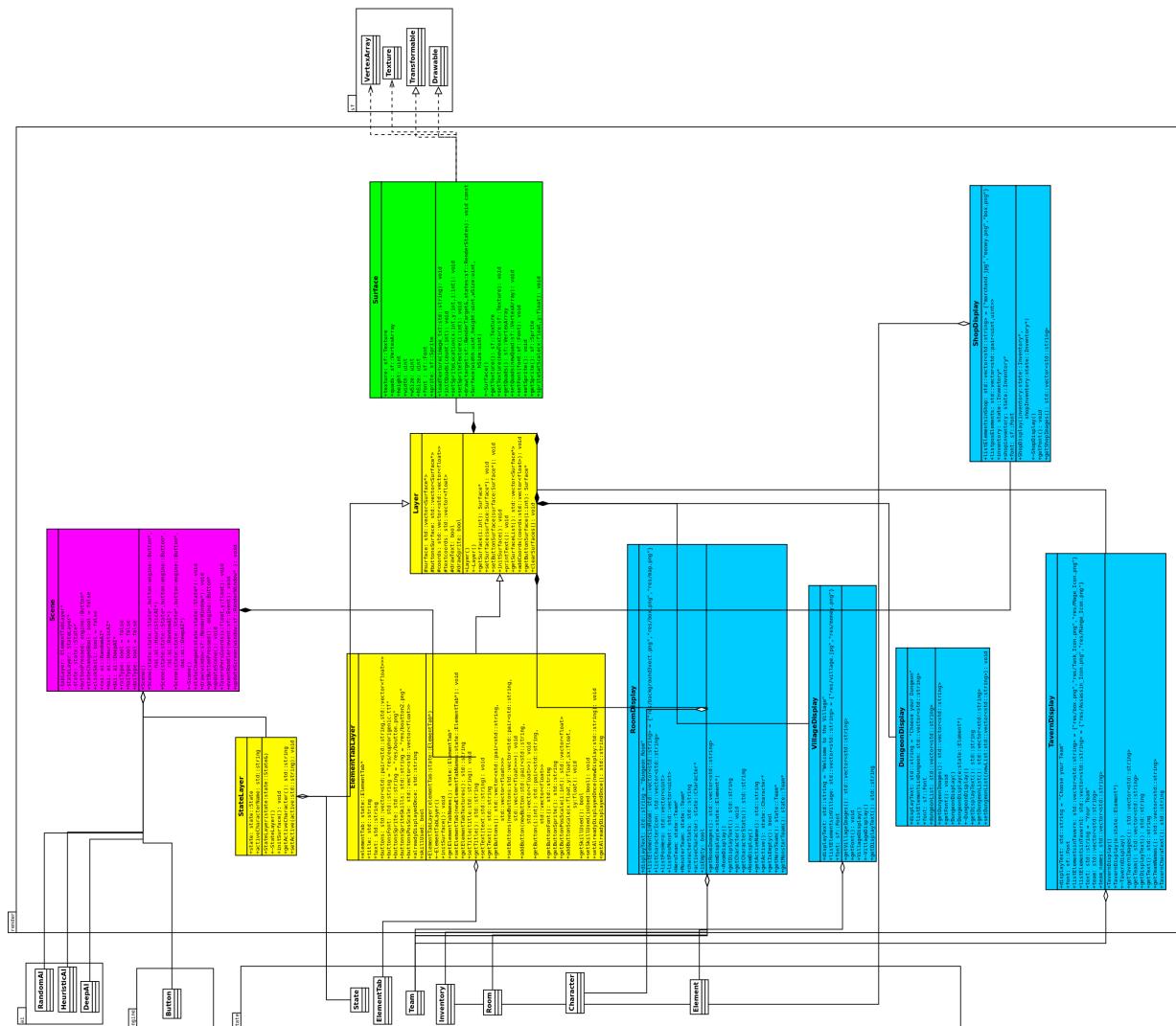


FIGURE 9 – Diagramme de classe associé au rendu global fait via DIA

4 Règles de changements d'états et moteur de jeu

4.1 Changements extérieurs

Ces changements sont provoqués par la pression sur une touche du clavier du joueur ou de l'utilisation de la souris (clic) :

- A partir du Village, le joueur est dirigé vers le choix du donjon
- Choix du donjon sur deux niveaux "EngineTest" et "The End".
- Des boutons de retour ou d'accès d'un état : "Donjon", "Back"
- Durant les combats le choix des différentes skills cliquables par le joueur
- A chaque écran, les boutons de retour à l'écran précédent
- Sur l'écran de Salle, le joueur peut choisir le personnage de l'équipe adverse à affronter/attaquer. Le choix est modélisé par le clic sur un bouton

4.2 Changements autonomes

Les changements autonomes sont exécutés en cascade à partir d'un changement extérieur.

- Une fois dans le room, la commande de calcul du personnage actif est effectuée automatiquement afin de déterminer l'ordre des personnages actifs qui joueront durant la partie
- Les affectations de la santé des personnages engagés dans le combat se fait naturellement.

4.3 Conception de logiciel

Le diagramme de classe du moteur est présenté en Figure 10, et nous présente l'implémentation que nous avons imaginé pour les différents changements précédemment décrits.

Classe **Command** : le rôle de ces différents classes listées ci-dessous, est de caractériser un comportement à l'origine d'un changement de nature autonome, extérieur, ou système. Grâce au **CommandTypeId**, nous allons identifier chaque instance de **Command** créée, afin d'y associer son comportement :

1. **BackCommand** : retour sur la scène précédente
2. **CalculateActiveCommand** : calcul du prochain personnage qui doit jouer lors du combat
3. **ChooseDungeonCommand** : choix du donjon par le joueur
4. **CreateDungeonCommand** : création du donjon, lorsque le joueur provient du Village
5. **CreateRoomCommand** : création de la salle de combat
6. **CreateVillageCommand** : création du Village
7. **UseSkillCommand** : utilisation d'une attaque par le joueur lors d'un combat

Classe **Engine** : cette classe est essentielle au bon fonctionnement de l'ensemble de la gestion des commandes. En effet, chaque commande est priorisée de manière unique. Cela permet à chaque époque, d'accumuler les différentes commandes à traiter dans un std : :map, et de les exécuter selon leur ordre de priorité. Une fois les commandes exécutées, elles sont supprimées.

Il reste à noter que lors de l'utilisation de l'**Engine**, ce dernier est désigné pour l'étape suivante qui est l'IA. Notre moteur reçoit et gère les Command qui lui sont envoyées à partir de la **Scene** dans laquelle l'IA intervient. Cela permet de simplifier grandement le traitement des commandes envoyées par l'IA et de la mettre dans une file d'attente auprès de **Engine**.



FIGURE 10 – Diagramme de classe associé au moteur de jeu fait via DIA

5 Intelligence Artificielle

5.1 Stratégies

5.1.1 Intelligence aléatoire

La stratégie d'implémentation de l'intelligence artificielle n'intervient pour l'instant que sur un écran : seul l'écran de combat est concerné car il expose une rencontre avec les ennemis. Ces derniers ont alors le choix pour l'instant soit d'attaquer un des héros , soit de passer leur tour. Cela donne donc 4 choix dans le rendu actuel du jeu.

Le choix de cette action se fait donc de manière aléatoire pour l'instant.

5.1.2 Intelligence basée sur des heuristiques

Cette intelligence artificielle est basée sur une anticipation comportementale de bas étage. En effet, l'IA analyse plusieurs facteurs et tente de mettre à mal le joueur avec deux principes :

1. L'ennemi essaie de minimiser le total de santé de l'équipe adverse (le total des vies doit être minimisé)
2. La cible prioritaire de l'ennemi est le membre de l'équipe adverse qui possède le moins de santé, ou qui a le moins de défense, ou les deux
3. La cible, dans le cas où les deux premiers points ne sont pas décisifs, devient le membre de l'équipe adverse apparaissant en premier dans la liste des joueurs à venir

Cette heuristique est donc mise en place par l'analyse des caractéristiques de l'équipe adverse, ou de leur position dans la liste des joueurs actifs futurs, ou encore l'état de la santé de l'équipe qui joue. Une stratégie est alors communiquée avec une combinaison de commande attribuée à l'équipe de monstre gérée par l'IA.

5.1.3 Intelligence artificielle basé sur un algorithme d'optimisation de type MIN/MAX

Cette intelligence artificielle a pour but de prendre en compte les différents éléments du jeu, de l'état actuel, et en tirer le meilleur parti. Cela implique donc une intelligence artificielle qui anticipe les mouvements, les meilleures actions que l'adverse pourrait jouer.

Pour modéliser cette intelligence, il faut tout d'abord modifier un élément qui nous permette de repérer un avantage pour une équipe ou l'autre. Nous avons donc décider d'implémenter et définir un score qui prend en paramètre de calcul les éléments stratégiques utilisés dans l'intelligence précédente, afin de leur attribuer un poids, et de mesurer un score pour chaque état possible.

A présent, il faut déterminer un procédé d'étude des maximas et minimas successifs pour les scores des deux joueurs. Ceci nous impose de créer un arbre, composé de **Noeud**. Ces derniers contiennent en attributs l'étage et un parent ainsi que des enfants ; ces trois données nous permettent de repérer à chaque moment notre position dans l'arbre. L'étage permet de repérer le nombre de tour que nous anticipons en se positionnant sur un noeud en particulier. Les parents nous rapprochent du présent et les enfants nous en éloignent. A chaque étage, selon si celui-ci est pair ou impair, nous tentons de maximiser ou minimiser le score de l'équipe exploitant l'intelligence artificielle.

Afin de nous déplacer correctement dans l'arbre, nous utilisons deux éléments : une liste d'actions qui nous permettent de calculer le score associé à l'action envisagée, ainsi qu'un clone de l'équipe qui de proche en proche est édité, pour calculer le score associé.

Une fois l'arbre rempli, il ne reste qu'à choisir le score le plus souhaitable et envoyé la première action à faire au moteur.

Lors de l'exécution du jeu avec la commande **Rollback**, il est possible de faire jouer le jeu par une IA en appuyant sur "R" une fois dans le donjon. Au bout d'un certain nombre de commandes exécutées, le rollback s'effectue en exécutant les commandes à l'envers.

5.2 Conception de logiciel

Le diagramme des classes pour l'intelligence artificielle est présenté en Figure 11. Classes AI. Les classes filles de la classe AI implantent différentes stratégies d'IA, que l'on puisse appliquer pour un ennemi :

- **RandomAI** : Intelligence aléatoire à laquelle on fournit une liste de string pour faire un choix hasardeux sur l'un d'eux ; nous fournissons dès à présent les entités Team et ActiveCharacter qui seront utiles lors des prochaines définitions d'IA.
- Pour les deux AI suivantes, nous utilisons une classe **Strategy** : celle-ci nous permet de triter les données issues de l'état actuel du jeu, et d'en prendre les caractéristiques qui permettent ensuite aux différentes IA de déterminer les meilleures actions à entreprendre. **Strategy** fournit donc des éléments de state qui influent le choix de l'IA selon son niveau d'interprétation (Heuristic ou Deep). On notera notamment plusieurs paramètres pris en compte :
 1. La santé totale de l'équipe adverse
 2. Le personnage ayant la défense et la vie la plus faible dans l'équipe adverse
 3. Le personnage actif lors de ce tour
 4. Le premier personnage de l'équipe adverse à jouer
- **Heuristic AI** : Intelligence artificielle comportementale de bas niveau. Celle-ci fournit à notre classe AI une liste de commande à envoyer au moteur, qui correspond à une stratégie définie plus bas dans l'architecture. La stratégie est définie selon les points évoqués dans la sous-partie précédente.
- **Deep AI** : Intelligence artificielle dite avancée, qui permet d'étudier et d'anticiper les choix faits par le joueur. Comme la précédente cette IA prendra en argument les éléments dits marquants et stratégiques concernant les deux équipes qui jouent. Ainsi ces paramètres influent sur le score prévisionnel sur plusieurs tours en avance, et un algorithme MIN/MAX détermine les actions les plus intéressantes, envoyées au moteur.

Il est à noter que pour l'instant l'IA avancée n'est pas fonctionnel.

Rollback : Le mode Rollback permet au joueur de rejouer les dernières actions faites en arrière. Cela demande une souplesse au niveau de l'utilisation des objets du code. Ainsi, nous avons défini un paramètre global nous permettant de savoir, si le joueur appuie sur la touche "R", modifier ce paramètre. Une fois ce dernier modifié, toutes les commandes qui ont été stockées, sont de nouveau exécuter mais en sens inverse (compris dans la définition de celles-ci).

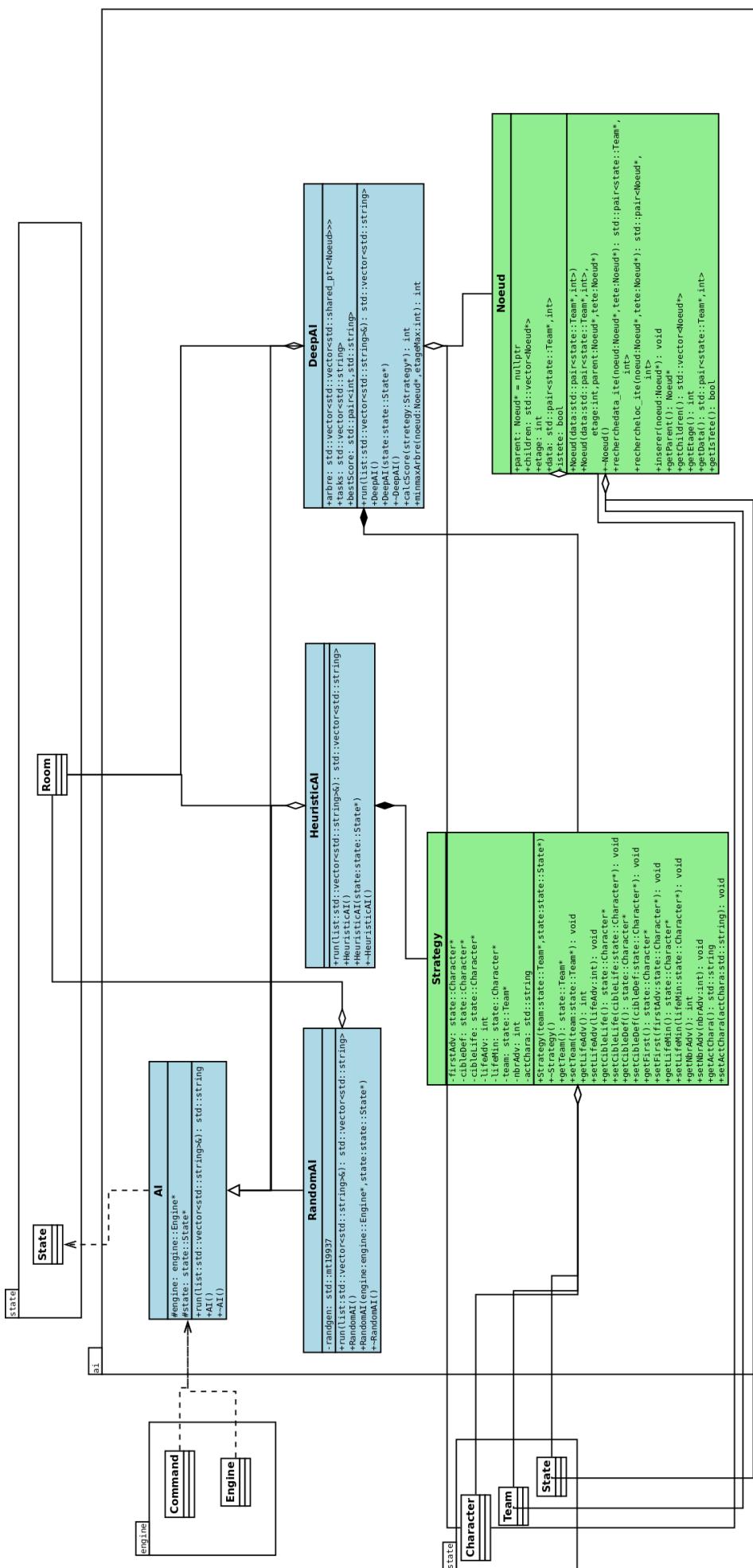


FIGURE 11 – Diagramme de classe associé à l'intelligence artificielle, fait via DIA

6 Modularisation

6.1 Organisation des modules

6.1.1 Implantation du multi-threading

Le multi-threading est ici essentiel afin de répartir les différentes actions selon des threads parallélisés. Cela permet de donner au jeu un aspect plus fluide et surtout d'alléger le temps de traitement de l'ensemble des opérations à réaliser pour un tic d'horloge.

Le multi-threading nous permet donc d'utiliser deux threads : un thread pour la gestion du moteur, un thread pour la gestion de l'IA.

Commandes

Un temps d'attente est initialisé dans le moteur afin de séquencer l'exécution des commandes lui parvenant. Cela permet également de traiter les actions et commandes émises par l'IA en parallèle, qui sont récupérées à la fin du cycle en synchronisant les deux threads dans le client.

Notification de rendu Ayant séquencé le fonctionnement du moteur, ce dernier exécute à présent une suite d'actions par ordre de déroulement chronologique. La Scène produit le rendu grâce à l'état de jeu qui a changé. Cela nous permet donc d'éditionner chronologiquement le rendu en fonction des commandes qui perviennent et sont exécutées dans l'engine.

6.1.2 Répartition sur différentes machines : rassemblement des joueurs

Client et Server Le Client possède un engine, une IA et lance le rendu. Lors du lancement du jeu le client envoie une requête au server pour voir si il peut s'enregistrer. Notre jeu peut être jouer au maximum par 2 joueurs, cela veut dire que quand le client essaie de se connecter et que deux joueurs sont déjà connectés il reçoit une réponse négative. Il ne peut pas s'enregistrer. Le serveur enregistre les différents utilisateurs (joueurs).

De plus, le serveur possède à présent une std::map qui associe à chaque commande un id. De plus la classe commande service possède aussi deux autres std::map nommés commandUser1/2. La première map permet d'enregistrer toutes les commandes de tous les joueurs. Quand un joueur envoie une commande grâce à SendCommand, exécutée par l'engine, elles sont stockées dans sa map (commandUser) puis elles sont ensuite déchargées dans commandMap où l'id correspond au joueur qui les a effectuées. Ensuite cette map est envoyée à tous les joueurs pour qu'ils puissent les exécuter et mettre à jour leur state. Si les joueurs ne reçoivent pas de message de passage de tour de la part du serveur, ils ne peuvent pas exécuter leurs commandes.

| | |
|-----------------------------|--|
| Requete GET /player/<id> | |
| Pas de donnees en entree | |
| Cas joueur <id>existe | Statut Ok Donnees sortie ; type : "objet", properties : {"name" : { type :string },}, required : ["name"] |
| Cas <id>négatif | Statut Ok Donnees sortie ; type : "object", properties : {"name" : { type :string },}, required : ["name"] |
| Cas joueur <id>n'existe pas | Statut NOT_FOUND Pas de données de sortie |

| | |
|---|--|
| Requête POST /player/<id> | |
| Donnees en entree | |
| type : "object", properties : { "name" : { type :string }, }, required : ["name"] | |
| Cas joueur <id>existe | Statut NO_CONTENT Pas de données de sortie |
| Cas joueur <id>n'existe pas | Donnees sortie : type ; "liste" properties : " "name" : { type :string }, |

| | |
|-----------------------------|---|
| Requête DELETE /player/<id> | |
| Pas de données en entrée | |
| Cas joueur <id>existe | Statut NO_CONTENT Pas de données de sortie |
| Cas joueur <id>n'existe pas | Statut NOT_FOUND Pas de données de sortie |

6.1.3 Répartition sur différentes machines : échange des commandes

Pour la gestion des commandes, tous les clients envoient leurs commandes moteur au serveur, qui sont stockées dans une map d'exécution update régulièrement. Il s'agit donc d'exécuter successivement les commandes par les deux joueurs, et la liste des dernières actions faites sont listées pour les deux joueurs.

6.2 Conception de logiciel

Client Le diagramme des classes associé au client est présenté en figure 12

La classe Client permet de structurer l'ensemble du jeu, c'est à dire, le moteur qui traite l'avancement chronologique de la scène en fonction des commandes exécutées, et d'autre part l'intelligence artificielle qui édite les actions avantageuses pour la machine, envoyées au moteur.

La fonction de record est implémentée pour sauvegarder l'état du jeu, c'est à dire les séquences de commandes ayant été exécutées depuis le lancement du jeu. Ces commandes sont alors sauvegardées dans un fichier .json et peuvent alors être rejouées à partir de la fonctionnalité de Play, la lecture d'un fichier .json qui est composé des commandes discutées plus tôt.

Lors du Record chaque commande est écrite dans le fichier de sauvegarde après avoir été exécutée. On écrit dans le fichier l'identifiant de la commande et les paramètres nécessaires pour son exécution. À chaque commande, le fichier est donc modifié. Le Replay quant à lui lit au début de la partie le fichier .json et sauvegarde les commandes dans une liste appartenant à la classe Engine. Par la suite, la fonction update de l'état n'attend plus de commande de l'extérieur mais exécute les commandes contenues dans la liste définie précédemment.

Server Le diagramme des classes associé au client est présenté en figure 12

L'objectif de cette partie est de mettre en place un service d'API Rest qui permet d'enregistrer des utilisateurs dans une base de données d'utilisateurs (std ::map associant l'id de l'utilisateur et un std ::unique vecteur le définissant). UserDB sert cette fonctionnalité. La classe UserService sert à définir les fonctions get et getAll qui permettent respectivement d'avoir un utilisateur par son id et d'avoir tous les utilisateurs. De plus c'est ici que les fonctions d'addition et de suppression des utilisateurs sont implémentées. Lors de l'envoi au serveur d'une commande (Get, Post, Delete), c'est la classe Service Manager qui permet l'appel des fonctions de User Service.

La classe Command Service sert à gérer les différentes commandes que reçoit le serveur. Les protocoles GET, POST et DELETE sont définis pour les commandes telles qu'il est possible de récupérer toutes les commandes ou les récupérer une par une. De même pour les DELETE on peut toutes les enlever d'un coup ou une par une avec leur id.

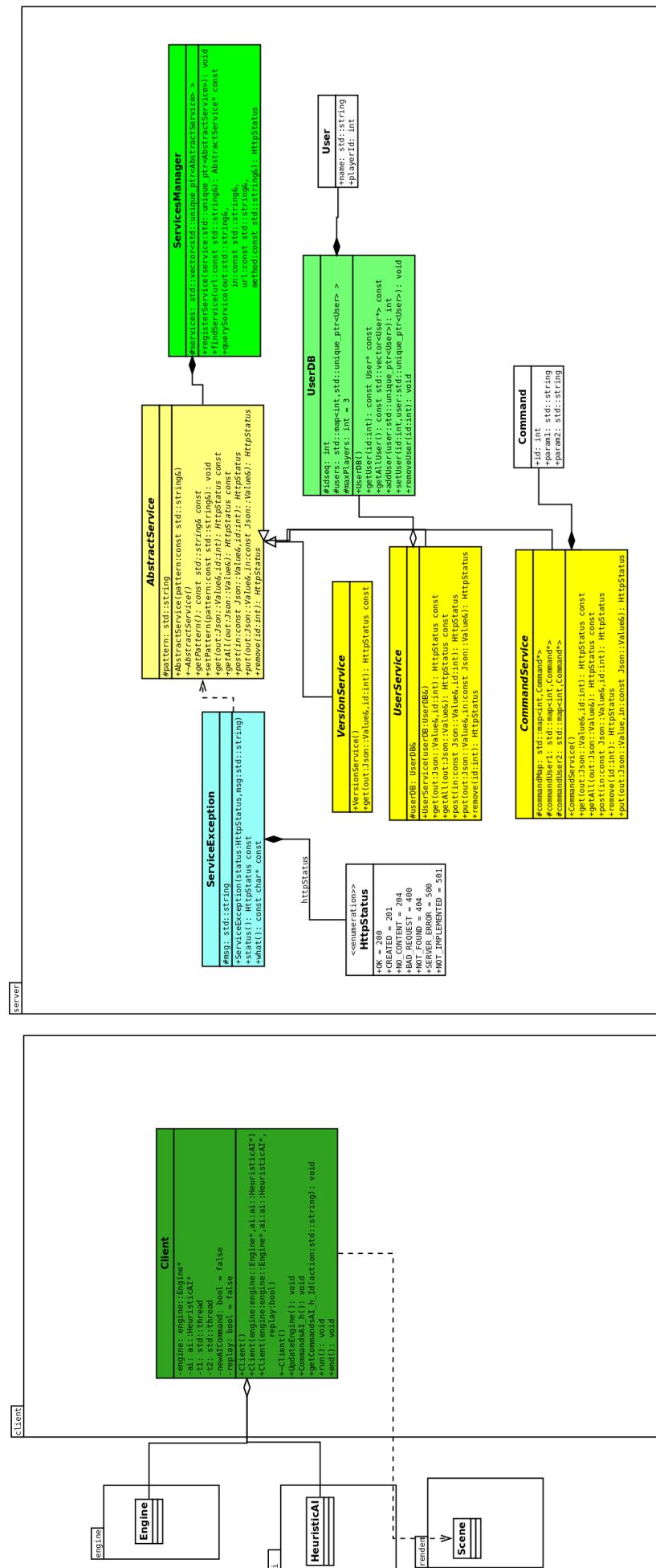


FIGURE 12 – Diagramme de classe associé à la modularisation, fait via DIA