Celine Ozkaya
Remy Théolier
Lise Leclef
Raphaël Chauvier
Sacha Malaterre

LO21
Projet Splendor Duel
-
Deuxième Compte Rendu

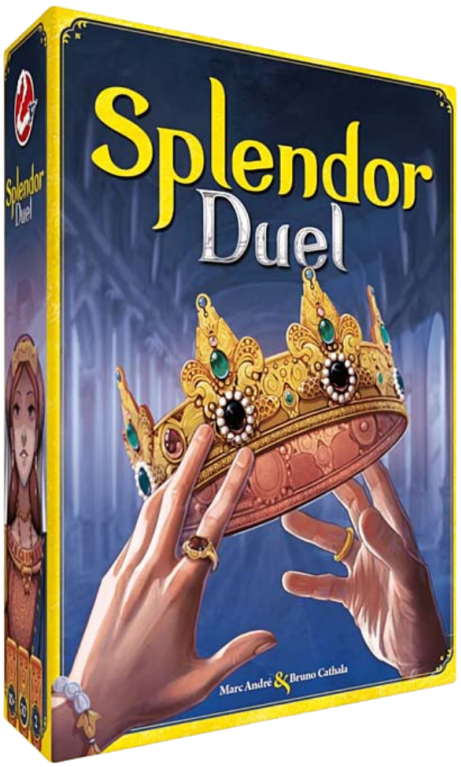


Table des matières :

I. Introduction

Nous avons changé plusieurs éléments de notre architecture de départ, en effet, comme détaillé dans la suite, les éléments liés au plateau/jetons, et le système de joueur à été revu pour être en adéquation avec l'interface QT et la gestion optimum des objets (future IA, gestion d'une partie, gestion des entrées/sorties). L'implémentation du jeu de base a bien débuté, pour le moment, seuls les éléments liés aux jetons au niveau du sac et du plateau , et aux privilèges au niveau du plateau fonctionnent (les classes et méthodes associées sont implémentées).
La cohésion de groupe jusqu'à maintenant est très bonne, le projet nous intéressant tous.
Le projet est disponible de l'adresse gitlab suivante:
https://gitlab.utc.fr/ozkayace/projet-lo21-splendor-duel

II. Nomenclature

Pour éviter toutes confusions de nommage et pour faciliter la compréhension du code de chacun, nous avons décidé de mettre en place le lexique suivant :

Variable	Signification
Classe Player	
name	nom du joueur
nbPrivilege	nombre de privilèges
type	type du joueur : humain ou IA
nbCrown	nombre de couronnes total parmis les cartes du joueur
prestigePoints	points de prestige totaux
jewelryCards	tableau représentant les cartes joailleries qu'un joueur a acheté
royalCards	tableau représentant les cartes royales (0, 1 ou 2) qu'un joueur a récupéré
reservedCards	tableau représentant les cartes réservées par un joueur
tokens	tableau comportant pour chaque couleur un tableau avec tous les jetons que le joueur possède
tokenSummary	Dictionnaire représentant le nombre de jetons par couleur du joueur
blueSummary (de même pour les autres couleurs de cartes)	Carte résumée des cartes bleues que possède un joueur
Classe Card	
Summarycard	Carte de résumé qui permettent une meilleure visibilité du jeu
JewelryCard	Carte de joaillerie
Bonus	Elément contenant la couleur et le nombre de bonus
RoyalCard	Cartes royales
Deck_level_x	Représente un des 3 decks du jeu

PyramidCards	Pyramide de cartes à 3 niveaux du jeu
Fichier Jeton (.h et .cpp)	
TokenColor	Couleur de Jeton
Token	Jeton
TotalToken	Ensemble contenant les instances de tous les jetons possible dans le jeu
Bag	Sac de jetons
TotalPrivilege	Ensemble contenant les instances de tous les privilèges
Board	Plateau de jeu contenant les jetons

III. Nouvelle décomposition des tâches et tâches déjà effectuées

A. Tâches effectuées:

- Révision et amélioration UML (V2)
- Architecture globale du logiciel déterminée: gestion de la position des jetons, de leur nombre, idem pour les privilèges.
- Implémentation initiale de certaines classes (en particulier sur les fichiers jetons)
- Première idée de la représentation graphique du jeu

B. Tâches restantes: (dans l'ordre d'exécution)

- Continuer l'implémentation initiale des classes
- Réfléchir à la gestion des classes concernant le déroulement d'une partie (déroulement d'une partie, règles de jeu, etc), fichiers partie.
- Implémenter certains design pattern sur des classes, en particulier les design pattern singleton sur les classes où une seule instance d'objet doit exister.
- Concevoir les graphismes du jeu
- Effectuer une première interface graphique du jeu sous QT

Le planning de Gantt est toujours valide, nous continuons de le suivre.

IV. Description des modules et des classes

1. Classe Joueur

La classe Joueur est implémentée par Lise et Céline. Nous avons mis à jour les attributs ainsi que les méthodes de la classe (ajout et suppression) du précédent rapport. Voici les différentes méthodes ainsi que leur fonctionnalité, à ce jour :

- getName, getPrivilege, getCrowns, getPrestige, getTokens et getType, qui renvoient respectivement le nom du joueur, son nombre de privilège, son nombre de couronnes, son nombre total de points de prestiges, son nombre de jetons et enfin le type du joueur.
- void removeToken(Token token) qui permet de supprimer un jeton de la liste de jetons et de mettre à jour le dictionnaire
- void addToken(Token token) qui permet à l'inverse d'ajouter un jeton à la liste et au dictionnaire
- void addCrowns() qui permet de rajouter une couronne au compteur global et de choisir une carte royale quand le joueur a atteint 3 ou 6 couronnes
- void addPrestige(int points, TokenColor color) qui permet de rajouter les points au compteur global (et dans la carte résumé de la couleur si une couleur est entrée), cette méthode permettra de simplifier la vérification des conditions de victoire
- void addPrivilege() et void removePrivilege qui permettent simplement d'incrémenter de 1 le compteur de privilège ou de réduire de 1 ce compteur
- void addJewelryCard(JewelryCard card) permet de rajouter une carte Joaillerie au tableau des cartes, d'utiliser la capacité de la carte et de mettre à jour la carte résumé correspondante
- void addRoyalCard(RoyalCard card) permet de rajouter une carte Royale au tableau correspondant, d'utiliser la capacité de la carte et de mettre à jour les points de prestiges du joueur
- void actionAddToken() qui permet de réaliser l'action de prendre les jetons sur le plateau
- void actionReserveCard() qui permet de réaliser l'action de réserver un carte et de récupérer un jeton Or, le choix de la carte et du jeton se font depuis la fonction qui ajoute ensuite la carte à la réserve et qui ajoute un or au joueur
- void actionBuyCard(Jewerlry &card) permet au joueur d'acheter une carte. On y vérifie que le joueur a les moyens d'acheter la carte sélectionnée puis retire et place dans le sac les jetons dépensés lors de l'achat, retire la carte de la pyramide et ajoute la carte à la liste des cartes possédées par le joueur.
- bool canBuyCard(const Jewerlry &card) verfie qu'un joueur à la moyen d'acheter une carte
- void spendRessources(const Jewerlry &card) effectue le paiement de la carte

En termes d'attributs, nous avons décidé d'ajouter des cartes résumées (comme détaillé à la fin de la section Classe Cartes) afin de manipuler les totaux de points de prestiges, bonus et couronnes par couleur. Chaque joueur aura une instance de cette classe par

couleur de carte. De plus, nous représentons les jetons d'un joueur avec un vector de vector tel que chaque élément du tableau présente la liste des jetons d'une couleur. Ex : tokens[0] = {redToken1, redToken4}, tokens[1] = {goldToken2}... Nous utilisons également un tableau de tuple (couleur du jeton, nombre de jeton de cette couleur) afin d'acceder au nombre de jetons plus facilement.

Pour ce qui est de la progression de cette classe, la grande majorité des méthodes sont déjà rédigées en pseudo-langage (afin d'être au clair avec leur structure générale afin de faciliter la "traduction" en C++) et beaucoup ont déjà commencé à être codées en C++. Nous avons également décidé de penser les méthodes en vue de leur manipulation avec l'interface.

Pour cette classe, il restera donc à finir les méthodes notamment en traduisant le pseudo-langage en C++, et il faudra également commencer l'interface.

2. Fichiers Jeton

Les classes primaires du Jeu de Splendor Duel permettant et utilisant la manipulation des Jetons sont définies dans les fichiers Jetons.h et Jetons.cpp. Grâce à ces implémentations terminées, on est capable désormais, parmi d'autres possibilités, de créer une instance de sac de jetons, de plateau de jeu, de piocher des jetons et de remplir un tableau.
Pour cela, nous avons instancié un type énuméré Couleur permettant de gérer les différentes couleurs possibles d'un jeton, puis nous avons défini une classe privilege et une classe jeton, qui permettent de créer des objets de ce type et d'obtenir des informations sur ceux-ci.

Ensuite, nous avons créé deux classes de gestion globale de ces classes: la classe JetonTotaux et la classe PrivilegesTotaux. Ces classes servent à instancier tous les jetons ainsi que tous les privileges du jeu.

Une classe d'exception pour la classe principale Jeton a aussi été instaurée, permettant de connaître l'origine des erreurs liés à celle-ci (aux jetons, au plateau, etc).

Pour la gestion du sac de jetons, grâce à la classe JetonTotaux, on peut initialiser le sac avec tous les jetons. Les méthodes comme piocherJeton et placerJeton seront très utiles pour le jeu.

Pour finir, la gestion du plateau de jetons est basée sur une matrice d'objet jeton, le plateau est initialisé avec tous les privilèges du jeu et le sac plein, avec lequel on remplit complètement le plateau. Cette classe est très utile pour prendre des jetons sur le plateau, connaissant les coordonnées du jeton à prendre.

Toutes les classes ont des méthodes de statut, permettant de savoir le nombre d'objets en elles (jetons ou privilèges), ou si elles sont vides. Cela permettra de mieux appréhender QT.
Le détail des fonctions est dans le git et un aperçu est sur l'UML. A l'avenir, pour encore mieux contrôler les entrées et sorties, le design pattern singleton pourra être utilisé pour éviter la duplication d'instance.

Le développement de cette partie est donc situé à 80%. Il reste donc encore une poignée d'heures de travail à passer dessus. Environ 15 heures de développement ont déjà été nécessaires (entre la compréhension du sujet et maintenant).

3. Fichiers Cartes

Les deux fichiers nommés Cards.h et Cards.cpp servent à définir, initialiser et gérer les instances des cartes joailleries, royales et la pyramide de cartes de niveaux différents. De plus, pour le stockage de l'ensemble des cartes, nous avons décidé de fonctionner avec une base de données en SQL (cf partie 5). Ces fichiers ainsi que la base de données ont été implémentés par Sacha et Rémy.
Le fichier Cards.h est composé des classes :
En premier lieu nous avons ajouté un enum Abilities qui comprend les différentes capacités des cartes. Ensuite nous avons créé plusieurs classes.

- JewelryCard

  Qui permet d'instancier les différentes cartes joailleries.
  Elle est composée,
  en privée :
  - unsigned int level
  - unsigned int cost;
  - unsigned int prestige_points;
  - unsigned int crowns;
  - Abilities ability;
  - TokenColor bonus;

  en public,
  - Constructeur
  - getCost() : qui renvoie le prix.
  - getPrestige() : qui renvoie le prestige.
  - getCrowns() : qui renvoie le nombre de couronnes.
  - getAbility() : qui renvoie la/les capacité.s de la carte.
  - getBonus() : qui renvoie le bonus que donne la carte.

- JewelryCardError

  Qui permet de gérer les erreurs et exceptions liées aux cartes joailleries. Elle est composée d'un attribut *reason* et d'un constructeur

- RoyalCard

  Qui permet d'instancier les différentes cartes royales.
  Elle est composée,
  en privée :
  - unsigned int prestige_points
  - Abilities ability

  en public,

- RoyalCard(unsigned int pp, Abilities a)
- getPrestige() : qui renvoie le prestige donné par la carte.
- getAbility() : qui renvoie la/les capacité.s de la carte.

- RoyalCardError

  Qui permet de gérer les erreurs et exceptions liées aux cartes royales. Elle est composée d'un attribut *reason* et d'un constructeur.

- Deck_level_one

  Qui permet de créer et de gérer une unique instance de pioche de carte de niveau 1. On rend cette amie de Pyramid_Cards pour pouvoir accéder à son attribut *pioche*

- Deck_level_two

  Qui permet de créer et de gérer une unique instance de pioche de carte de niveau 2. On rend cette amie de Pyramid_Cards pour pouvoir accéder à son attribut *pioche*

- Deck_level_three

  Qui permet de créer et de gérer une unique instance de pioche de carte de niveau 3. On rend cette amie de Pyramid_Cards pour pouvoir accéder à son attribut *pioche*

- Pyramid_Cards

  Qui permet de créer et de gérer la pyramide de carte. Pour ce faire nous avons déclaré 3 vecteurs un par niveau de cartes. Ces attributs fonctionnent séparément.
  Elle est composée de,
  en privée :
  - std::vector<JewelryCard *> row_level_one
  - std::vector<JewelryCard *> row_level_two
  - std::vector<JewelryCard *> row_level_three
  - static const unsigned int max_level_one = 5
  - static const unsigned int max_level_two = 4
  - static const unsigned int max_level_three = 3
    Les trois attributs ci dessus servent à définir le nombre max de cartes sur chaque vecteur (par niveau)

  en publique :
  - drawCard(unsigned int level) : qui vient remplacer une nouvelle carte à la fin du vecteur de la ligne du niveau correspondant au paramètre level.

- takeCard(unsigned int level, unsigned int position) : qui renvoie la carte choisie tout en la supprimant du terrain.

A la demande de Lise et Céline, nous allons créer une classe SummaryCard qui créera des instances de cartes résumées. Chaque instance de ces cartes représentera, pour chaque couleur (gemme), le nombre de points de prestige total, le nombre de couronnes et les bonus afin d'accéder aux informations relatives aux conditions de victoire des joueurs plus facilement. Au niveau de l'interface, ces cartes nous permettront de représenter les cartes que possèdent les joueurs en représentant une seule carte (carte résumée) par couleur plutôt que de devoir afficher individuellement toutes les cartes qui ont été achetées à l'écran.
On estime le temps restant au développement des fichiers cartes à 2-3 heures, plus les corrections post essais entre les différentes classes qui peuvent varier selon l'importance des modifications à faire.

4. Fichiers Partie

Ces fichiers contiendront l'implémentation des classes de gestion de la partie, ainsi que des règles du jeu.
On devra donc créer une classe Environnement, qui contiendra toutes les classes nécessaires à la bonne exécution d'une partie (Cartes, Pioches, Jetons, Plateau, Joueur, IA, …). En plus d'une classe Partie qui gèrera les tours du jeu, ainsi que les règles du jeu.

La fichier partie n'a pas encore été implémenté du tout, seulement une idée de l'architecture a été établie. Au moins une dizaine d'heures seront nécessaires.

5. Fichier cards

Ce fichier est le fichier contenant la base de données des cartes. Nous nous réservons la possibilité de le modifier ultérieurement mais pour le moment, celui-ci peut être complété par le fichier insert_cards.sql de la sorte :

- Depuis le terminal : sqlite3 cards.db
- Depuis le terminal sqlite dans le répertoire contenant la base :

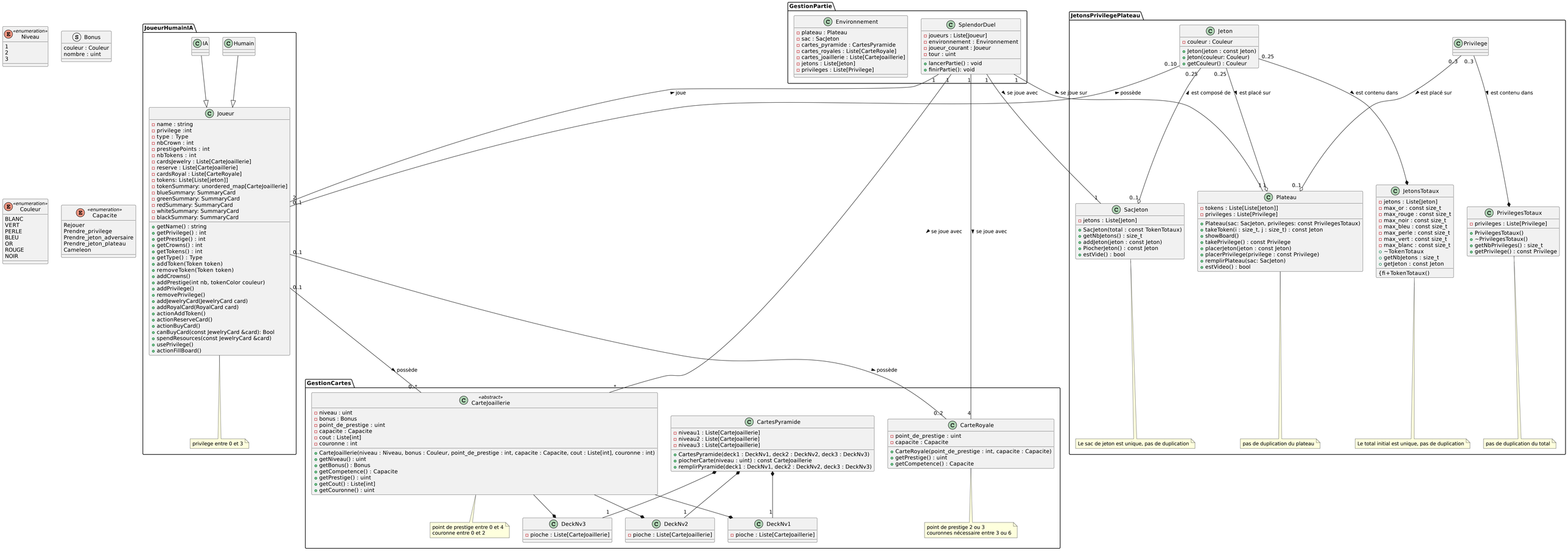
```
CREATE TABLE CardsTable (
   categorie TEXT,
   capacity TEXT,
   prestige INTEGER,
   crown INTEGER,
   cost_w INTEGER,
   cost_v INTEGER,
   cost_n INTEGER,
   cost_p INTEGER,
   color TEXT,
   lvl INTEGER
);


.read insert_cards.sql
```

La première commande permet de créer la table elle-même et la deuxième, de compléter la table à l'aide du fichier insert_cards.sql.

Comme le laisse penser la lecture de la table, Sacha et Rémy ont distingué les cartes "royales" et "joaillerie" par la "categorie" puis, ils ont intégré la capacité de la carte, son prestige, son coût, sa couleur et son niveau avec les champs après.
Pour créer une carte "joaillerie", le fichier insert_cards.sql contient par exemple :

```
INSERT INTO CardsTable
( categorie, prestige, crown, cost_w, cost_v, cost_n, cost_p, color,
lvl)
VALUES
( 'Jewerly', 3, 2, 3, 3, 5, 1, 'blue', 3 );
```

V. UML


«enumeration»
Niveau
1
2
3
Bonus
couleur : Couleur
nombre : uint
«enumeration»
Couleur
BLANC
VERT
PERLE
BLEU
OR
ROUGE
NOIR
«enumeration»
Capacite
Rejouer
Prendre_privilege
Prendre_jeton_adversaire
Prendre_jeton_plateau
Cameleon
JoueurHumainIA
IA
Humain
Joueur
- name : string
- privilege :int
- type : Type
- nbCrown : int
- prestigePoints : int
- nbTokens : int
- cardsJewelry : Liste[CarteJoaillerie]
- reserve : Liste[CarteJoaillerie]
- cardsRoyal : Liste[CarteRoyale]
- tokens: Liste[Liste[jeton]]
- tokenSummary: unordered_map[CarteJoaillerie]
- blueSummary: SummaryCard
- greenSummary: SummaryCard
- redSummary: SummaryCard
- whiteSummary: SummaryCard
- blackSummary: SummaryCard
+ getName() : string
+ getPrivilege() : int
+ getPrestige() : int
+ getCrowns() : int
+ getTokens() : int
+ getType() : Type
+ addToken(Token token)
+ removeToken(Token token)
+ addCrowns()
+ addPrestige(int nb, tokenColor couleur)
+ addPrivilege()
+ removePrivilege()
+ addJewelryCard(JewelryCard card)
+ addRoyalCard(RoyalCard card)
+ actionAddToken()
+ actionReserveCard()
+ actionBuyCard()
+ canBuyCard(const JewelryCard &card): Bool
+ spendResources(const JewelryCard &card)
+ usePrivilege()
+ actionFillBoard()
privilege entre 0 et 3
GestionPartie
Environnement
- plateau : Plateau
- sac : SacJeton
- cartes_pyramide : CartesPyramide
- cartes_royales : Liste[CarteRoyale]
- cartes_joaillerie : Liste[CarteJoaillerie]
- jetons : Liste[Jeton]
- privileges : Liste[Privilege]
SplendorDuel
- joueurs : Liste[Joueur]
- environnement : Environnement
- joueur_courant : Joueur
- tour : uint
+ lancerPartie() : void
+ finirPartie(): void
joue
se joue avec
se joue sur
possède
GestionCartes
«abstract»
CarteJoaillerie
- niveau : uint
- bonus : Bonus
- point_de_prestige : uint
- capacite : Capacite
- cout : Liste[int]
- couronne : int
+ CarteJoaillerie(niveau : Niveau, bonus : Couleur, point_de_prestige : int, capacite : Capacite, cout : Liste[int], couronne : int)
+ getNiveau() : uint
+ getBonus() : Bonus
+ getCompetence() : Capacite
+ getPrestige() : uint
+ getCout() : Liste[int]
+ getCouronne() : uint
point de prestige entre 0 et 4
couronne entre 0 et 2
CartesPyramide
- niveau1 : Liste[CarteJoaillerie]
- niveau2 : Liste[CarteJoaillerie]
- niveau3 : Liste[CarteJoaillerie]
+ CartesPyramide(deck1 : DeckNv1, deck2 : DeckNv2, deck3 : DeckNv3)
+ piocherCarte(niveau : uint) : const CarteJoaillerie
+ remplirPyramide(deck1 : DeckNv1, deck2 : DeckNv2, deck3 : DeckNv3)
CarteRoyale
- point_de_prestige : uint
- capacite : Capacite
+ CarteRoyale(point_de_prestige : int, capacite : Capacite)
+ getPrestige() : uint
+ getCompetence() : Capacite
point de prestige 2 ou 3
couronnes nécessaire entre 3 ou 6
DeckNv3
- pioche : Liste[CarteJoaillerie]
DeckNv2
- pioche : Liste[CarteJoaillerie]
DeckNv1
- pioche : Liste[CarteJoaillerie]
JetonsPrivilegePlateau
Jeton
- couleur : Couleur
+ Jeton(jeton : const Jeton)
+ Jeton(couleur: Couleur)
+ getCouleur() : Couleur
Privilege
est composé de
est placé sur
est contenu dans
est placé sur
est contenu dans
SacJeton
- jetons : Liste[Jeton]
+ SacJeton(total : const TokenTotaux)
+ getNbJetons() : size_t
+ addJeton(jeton : const Jeton)
+ PiocherJeton() : const Jeton
+ estVide() : bool
Le sac de jeton est unique, pas de duplication
Plateau
- tokens : Liste[Liste[Jeton]]
- privileges : Liste[Privilege]
+ Plateau(sac: SacJeton, privileges: const PrivilegesTotaux)
+ takeToken(i : size_t, j : size_t) : const Jeton
+ showBoard()
+ takePrivilege() : const Privilege
+ placerJeton(jeton : const Jeton)
+ placerPrivilege(privilege : const Privilege)
+ remplirPlateau(sac: SacJeton)
+ estVideo() : bool
pas de duplication du plateau
JetonsTotaux
- jetons : Liste[Jeton]
- max_or : const size_t
- max_rouge : const size_t
- max_noir : const size_t
- max_bleu : const size_t
- max_perle : const size_t
- max_vert : const size_t
- max_blanc : const size_t
+ ~TokenTotaux
+ getNbJetons : size_t
+ getJeton : const Jeton
{fi+TokenTotaux()
Le total initial est unique, pas de duplication
PrivilegesTotaux
- privileges : Liste[Privilege]
+ PrivilegesTotaux()
+ ~PrivilegesTotaux()
+ getNbPrivileges() : size_t
+ getPrivilege() : const Privilege
pas de duplication du total

Projet LO21

12 nov. 2023

http://

Chef de projet	
Dates du projet	14 sept. 2023 - 16 déc. 2023
Avancée	32%
Tâches	11
Ressources	5

Tâches

Nom	Date de début	Date de fin
Étude du Sujet	14/09/2023	07/10/2023
Analyser en profondeur les exigences du jeu Splendor Duel, les règles du jeu, et les fonctionnalités nécessaires.		
Shéma prévisionnel des classes et méthodes	21/09/2023	07/10/2023
Documenter les objets méthodes essentiels du projet, les fonctionnalités clés.		
Conception Initiale UML	28/09/2023	07/10/2023
Créer un diagramme UML initial pour représenter la structure et les relations des classes.		
UML 2	08/10/2023	19/10/2023
Réviser et améliorer le diagramme UML en tenant compte des commentaires de la première version.		
Architecture approfondie	08/10/2023	19/10/2023
Réfléchir à l'architecture globale du logiciel.		
Implémentation initial	20/10/2023	02/11/2023
Commencer/continuer à coder les classes et les fonctionnalités de base du jeu Splendor Duel en C++.		
Finalisation de la première implémentation	03/11/2023	16/11/2023
Travailler sur les fonctions et les mécanismes du jeu qui ne sont pas directement liés aux classes, améliorer les classes existantes avec les nouvelles connaissances acquises.		
Conception des images	03/11/2023	16/11/2023
Créer les graphismes et les images nécessaires pour le plateau, les cartes et les jetons du jeu.		
Conception interface graphique avec Qt	17/11/2023	23/11/2023
Créer une conception graphique pour l'interface utilisateur du jeu Splendor Duel.		
Implémentation interface	24/11/2023	15/12/2023
Intégrer l'interface utilisateur avec le code du jeu et s'assurer que tout fonctionne de manière fluide avec Qt, en C++.		
Finalisation	07/12/2023	15/12/2023
Finalisation du projet : batteries de tests en plus des tests existants, étude finale des fuites de mémoire, etc...		

Ressources

Nom	Rôle par défaut
Raphaël	Développeur
Céline	Développeur
Lise	Développeur
Sacha	Développeur
Remy	Développeur

Diagramme de Gantt

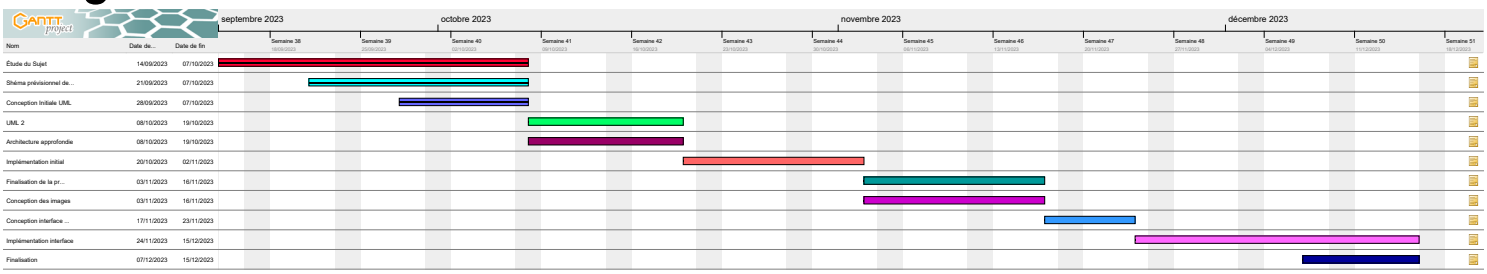


Diagramme des Ressources